Università degli Studi di Padova

Dipartimento di Matematica
"Tullio Levi-Civita"

Master's Degree in Computer Science

Physics Informed Neural Networks in Temporal Graphs

Master's Thesis

Supervisors

Prof. Nicolò Navarin
Paolo Frazzetto, PhD

Author

Filippo Visentin

Academic Year 2022-2023

Abstract

With the recent outbreak of COVID-19, researchers worldwide have come together in a collaborative effort to model and forecast the disease, in order to prepare, understand, and control it.
Epidemiological models offer a means to address uncertainties regarding the spread of the virus by combining available information from experimental studies and the opinion of experts to gain insight into the dynamics of infection and disease control.

Machine learning can help with epidemiological modelling, when little expertise is available or the spread is conditioned by the human behaviour, so it is difficult to capture using only theory.
The problem with classical machine learning techniques is that they require a lot of data, they are not interpretable and it is not easy to integrate available domain knowledge.
In order to get better models, usable for forecasting, machine learning can be enhanced with human knowledge by means of physics informed machine learning. In fact the virus spread can be interpreted as a physical phenomenon of which many properties are known and for which there already exist analysis tools.
Usually this phenomenon is modelled using probabilities that evolve in time. This evolution can be treated as a kind of dynamics and described using differential equations, that can be learned from data.

The goal of this thesis is to develop new interpretable physics informed machine learning techniques for finding epidemiological models, and compare these techniques to existing ones. These models should help understanding the disease and forecasting it.
Different types of physics information are included in the learned models: the structure of the social network where the infection spreads, some constraints on the states reached by the dynamical system and a restriction on which terms can be present in the differential equations.

To accomplish this, a technique called Sindy, used for dynamics identification, has been adapted to work on graphs, in this way it is able to extract spatial features and learn how they evolve in time. This evolution is encoded with symbolic differential equations that are interpretable.
To improve the generalization ability and plausibility of the models, we added some constraints in the learned probabilities dynamics. Because of the temporal evolution, the models are defined recurrently and this makes enforcing these

constraints challenging. Several techniques are proposed to tackle the problem, some are designed exploiting the characteristics of epidemiological modelling and some are applicable also in other contexts.
We also faced the problem of dealing with aggregated information. Because knowing the evolution in time of the infection of every single person is impossible, only statistics stemming from large populations can be used, often without knowing how the pandemic spreads spatially. For solving this we used a series of improving estimates of spread with single person granularity.

The developed techniques are tested to evaluate the forecasting performance, using datasets generated from simulations, and compared to a type of Spatial-Temporal Graph Neural Networks.
We find that our technique, Sindy Graph, has lower generalization errors than STGNNs, because more physics information can be integrated into the learned models. Moreover Sindy's models are more interpretable and simpler to train.

Structure

This thesis is structured in the following way:
in chapter Background the concepts required for understanding the problem and the methods from literature used to solve it are introduced and discussed.
Then in chapter The problem, Epidemiological modelling is introduced together with the problem tackled in this thesis.
Then Related works are discussed to give a literature overview on epidemiological forecasting techniques.
In the Contributions chapter, our techniques are introduced in a formal way, then implementation is discussed.
In Experimental Results we discuss how our method, called Sindy graph with aggregated supervision and constraints, performs in the task of epidemiological forecasting and compare it to an instance of existing techniques called Spatial-Temporal Graph Neural Networks.
Finally we give Conclusions supported by the experiments.

Contents

Notation

Notations	Descriptions
x	Scalar value
X	Set
G	Graph
$f(x)$, $F(x)$	Function of scalar input
$L(m)$	Loss function where m is a model
$\boldsymbol{x}$	Vector
$\boldsymbol{A}$	Tensor of ≥ 2 dimensions
$\boldsymbol{c}$	Vector of learnable parameters
$\boldsymbol{x}(t)$	Signal with multiple features that evolve in time continuously
$\boldsymbol{x}_t$	Signal with multiple features that evolve in time discretely at timestep t
$\tilde{\boldsymbol{x}}(t)$	Approximation of the signal x
σ	Generic activation function

Indexing

Tensors are indexed following the PyTorch convention. For example $\boldsymbol{x} = \boldsymbol{A}_{:,i}$ represents a tensor named $\boldsymbol{A}$ of 2 dimensions (a matrix) from which a vector $\boldsymbol{x}$ is extracted, corresponding to the i^{th} column of $\boldsymbol{A}$.
Functions may return a tensor that can be indexed directly, for example $\boldsymbol{a} = f(x)_{:,i}$ represents a function f of input x that returns a matrix, of which only the i^{th} column is taken to form a vector.

Chapter 1

Background

In this chapter the concepts required for understanding the problem and the methods from literature used to solve it are introduced and discussed. The literature studied is cited to support the theory and to provide useful references for further insight into the arguments.
At the beginning an high level introduction on the topic of Physics Informed Machine Learning is given, because our methods rely on physics information to obtain superior generalization capabilities than traditional techniques. To give a more general overview on the topic, some useful physics informed techniques will then be discussed.
Constrained Optimization is an important example of how to include physics information into a model, and is discussed in a separate section.
Then in Learning to Simulate Physics, the concept of *physics simulator* will be introduced, this will give a more clear idea of what a model that solves the problem should do. The definitions in this section will be then expanded in later chapters.
Graph Neural Networks follow, as they are a tool useful for modelling viral dynamics between people, and for this reason are used by us as a foundation on which building our techniques.

1.1 Physics Informed Machine Learning

Physics Informed Machine Learning (PIML) is the combination of machine learning techniques with prior knowledge about physics, so high level abstractions long thought by humans to explain natural phenomenons. While powerful techniques to learn models by data alone exist and are successful, the introduction of physics has emerged as an effective way to deal with **shortage of training data**, to increase the model's **capability to generalize**, to ensure the physical **plausibility** and **explainability** of results.
In this section an high level introduction about this area of research will be done following the categorization introduced in the survey [36] and the many references provided.

1.1.1 Motivations of PIML

Machine learning techniques have archived incredible success in domains where large amount of data is available using highly expressive architectures such as neural networks. The research has since started to explore ways of using these techniques to advance scientific discovery.
While deep neural networks are really good at finding relations between inputs and outputs if a lot of data is provided, the optimization process is not simple and can lead to sub optimal results. Moreover if not enough data is available, the model tends to overfit. Prior knowledge can help solve these issues by limiting the search space, providing information not contained in the data and improving the plausibility of the resulting models.
Physics prior knowledge is particularly powerful as it stems from a long history of observation and abstraction done by humans with scientific rigor, and has been validated both empirically and theoretically. Compared to other forms of prior knowledge such as logic rules, physics information requires specific ways to be integrated in machine learning.

In the following, some more motivations are given by introducing some applications in which PIML is used.

Physics models enhanced by data driven methods

Physics problems often involve analyzing huge amounts of data. Because machine learning methods are great at finding useful meaning in the data, interest has grown in apply these techniques for scientific discovery in physics. Advantages are more flexibility, generalizability and less computational cost.

Simulation

Traditional physics simulations are complex to implement, require specific domain knowledge and are heavy computationally. Many *surrogate* models use physics information and data to create a simulator without requiring **specific knowledge** on the system. For example in [24] a simulator is proposed that can learn to simulate a wide variety of challenging physical domains, involving fluids, rigid solids, and deformable materials. Using physics information in the form of model's architecture, it is possible to simulate many different scenarios and account for situations difficult to model explicitly (e.g external forces, boundary conditions), making this techniques more **flexible**.
A surrogate model made to be physically plausible can also be used to **extrapolate** dynamics not seen in the training set, making it able to generalize.
Moreover if the model is constructed to be explainable, meaningful scientific **discovery** can be made just by processing data [3].

Downscaling

Many numerical problems in physics require solving partial differential equations (PDEs) to then measure certain quantities of the system. In order to solve them numerically discretization in space and time is required, and for certain problems this is so fine grained to require huge computational power. Machine learning

techniques such as neural networks have been used to interpolate information at a coarser scale, given their ability to represent non linear relationships between variables and using data to learn a low-resolution to high-resolution mapping [8].

Parameterization

When modeling complex systems in physics a common technique is to start from a simple process which has free parameters and find them based on observations, minimizing the discrepancy with model's predictions. To accomplish this machine learning techniques are used increasingly often, by combining physics information to classical ML architectures [5].

Reduced order models

Often complex systems are difficult to describe in the usual observations space. Many techniques reduce the complexity or order of the system by using dimensionality reduction techniques based on machine learning and then model the system in the reduced space, making it more interpretable and controllable (in control theory) [27].
In fact by doing a projection of the input in a smaller space, a simpler, more general dynamics is uncovered and this can be captured using standard physics dynamics discovery. This works well when inputs are noisy, high dimensional or a change of coordinates is needed to simplify the dynamics.

Counterfactual Analysis of Physical Dynamics

Reasoning about the physical world requires understanding causes and effects of mechanical systems. The goal of this field of research is predict an outcome starting from some intervention on the initial conditions of the system [16]. The modeling of relationships between causes and effects and the extraction of information from an input can be done using ML techniques, informed about the physics involved in the system studied.

Improvement of Data-Driven Models from External Knowledge

Data driven methods are highly successful nowadays thanks to the expressive power of Neural Networks and the wealth of data available. The optimization process is hard in practice though and data can be limited in some problems, leading to suboptimal models. Moreover optimizing without **constraints** can lead to models that do not generalize well and violate physical laws and logic rules.
To account for this, physics prior information has been added to standard machine learning techniques that learn only from data.

Object-centric data

When describing the real world many systems are composed by many discrete objects, that change their properties with time. Graph Neural Networks (GNNs) are particularly good at modeling many-objects systems and their relationships

as their inductive bias matches that of data. They are also surprisingly good at generalize when the structure of the system is perturbed [11].

Spatio-Temporal data

Many physical phenomenons are represented as a multitude of values of interest evolving in time, for example a system evolving in space. Some techniques constrain in multiple ways the learned dynamics to be physically plausible, or inform the system using known PDEs to guide the optimization process.

1.1.2 Physics knowledge

There are several ways to introduce physical knowledge in machine learning. Some categories of knowledge are general enough to have inspired many useful techniques.

Lagrangian mechanics

In physics there are several ways to express the *motion of a system*. The Lagrangian mechanics expresses the dynamics of a system from an energy perspective, and through the *principle of least action* enables the enforcing of some physical constraints, such as the **conservation** of some quantities, by construction.
The **state** (or configuration) of a system can be defined by its position q and velocity q'. The **trajectory** of a system is a function of the state with respect to time, that describes how an initial state evolves in time changing its position and velocity.

From the *machine learning* perspective, the goal is to search for a function $l(q, q', t) = T(q, q', t) - V(q, t)$ where T is the kinetic energy and V is the potential energy of the system.
l represents the system from the energy perspective and completely captures the dynamics of it. l is the model to be learned and is represented for example by a neural network.

In the study of mechanics the **principle of least action** holds. The action of a trajectory $q(t)$ is expressed as:

$$\mathcal{L}[q(t)] = \int_{t_1}^{t_2} l(q(t), q'(t), t)\, dt$$

The principle states that the trajectory taken by the system starting from an initial condition, from time t_1 to time t_2, minimizes the action taken, so every physically plausible trajectory must minimize $\mathcal{L}$.
This implies the Euler-Lagrange equation:

$$\frac{\partial l}{\partial q} = \frac{d}{dt}\frac{\partial l}{\partial q'}$$

Using this equation, one can impose some specific constraints that translate in conserved quantities. For example by imposing $\frac{\partial l}{\partial t} = 0$, that is l does not depend on time, the energy of the system is conserved as the trajectory unfolds.

So by having no time variable and satisfying the Euler-Lagrange equation, the **energy is conserved by construction** [19].
Other quantities that can be constrained to conserve, include the linear momentum and the angular momentum.

Symmetries

A symmetry defined on a system is a transformation that keeps certain properties *equivariant* [25]. Symmetries are often used in physics to describe regularities and can be incorporated in machine learning techniques to inform them. Often these symmetries are satisfied by construction of the technique, so all the hypothesis in the hypothesis space satisfy them.
For example special neural network layers have been conceived such that they satisfy known symmetries of the domain of study.
Convolutional layers are known to be equivariant under **translation** of the input image. Graph neural networks are equivariant under the **permutation** of nodes, because often only the relations between objects are relevant, not their order, e.g. for particle systems. Spherical convolutions are invariant over **rotations** in the group SO(3), and are used in astrophysics for spherical observations and chemistry when modelling molecules that exhibit the same symmetry. LSTMs are invariant over **time warping**, and that is desirable to model correctly long-range dependencies on the input sequence.

Koopman theory

Methods based on Koopman theory are used when dealing with dynamics that are describable using nonlinear differential equations [9]. The theory guarantees the possibility of moving from the state (or input) space to a measures (or latent) space where the nonlinear dynamics become linear(describable with linear differential equations), albeit infinitely dimensional. The linearity of the reduced dynamics renders the study of the system more simple and the integration and control of it possible using closed form solutions.

As an example of application of this theory we consider the **Koopman Operator** and explain how it is used in the measures space, to advance the dynamics of a modeled system, without dealing with nonlinear differential equations.
If x_t is the state of the system at time t, one can advance to the next state by using its characteristic nonlinear function f that map the state of the system forward in time:

$$\boldsymbol{x}_{t+1} = f(\boldsymbol{x}_t)$$

The Koopman theory says that it is possible to map the state to the measures space using a nonlinear function φ, where the measure $\boldsymbol{m}_t$ is infinitely dimensional:

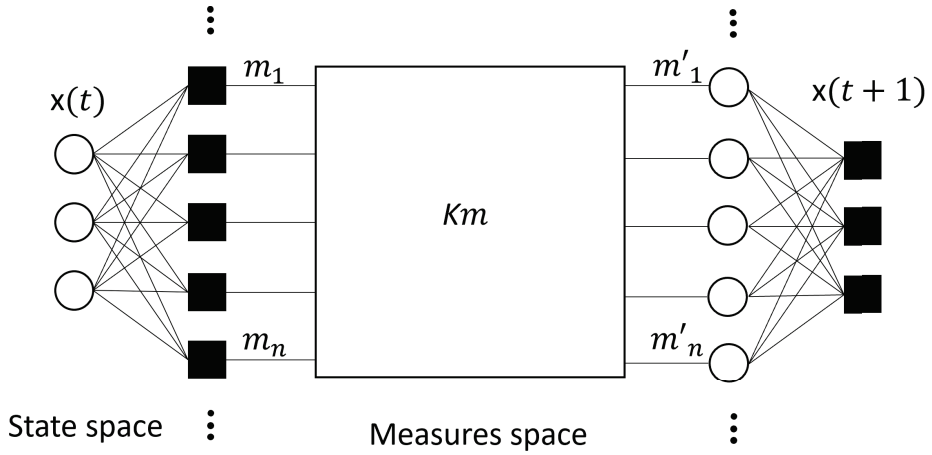
$$\boldsymbol{m}_t = \varphi(\boldsymbol{x}_t)$$

The Koopman operator $\mathcal{K}$ is an infinite dimensional matrix that advances the measure in time:

$$\boldsymbol{m}_{t+1} = \mathcal{K}\boldsymbol{m}_t$$

To perform this mapping, an infinite series of measuring functions $\varphi_n(\boldsymbol{x}_t)$ is used, each outputting a component m_n of the measure $\boldsymbol{m}$, and they must be

invertible to return to state space. In practice though the number of them is finite and the mapping only approximate.
If the measuring functions are chosen wisely, each captures a meaningful component of the dynamics that can be interpretable [1].



Physics-Informed Computation Graph

There are several classic physical knowledge based methods that have been enhanced by neural networks, including some already discussed. These techniques use computational graphs that mimic the behavior of classical methods but substituting some fixed parameters with ones learned from data, or replace variables difficult to estimate with neural networks.
These techniques enforce physics by construction, because of this, the prediction performance is better and less data is required than non physically informed techniques. Moreover general physical knowledge can be used, to enable less architectures to be conceived and leave the burden of specializing the model to the specific problem to the learnable parameters.
This hybridization has been introduced for example in classical numerical methods for solving Partial Differential Equations (PDEs), such as the Finite Difference Method, The Finite Volume Method and the Finite Element Method.
Physics based methods and deep learning can be fused also at higher level: by creating some modules that are entirely deep and some that employ pure physics methods, exposing only inputs and outputs to each other.

Physics-Informed Optimization

Prior physical knowledge can be integrated in machine learning techniques that use optimization to learn a model. This can be obtained by adding a term in the **loss function** or by other forms of **regularization**. The goal is to reshape the optimization space in order to encourage the training process to converge to a physically plausible solution.
Because the goal of optimization in ML is to find a model that best solve a particular task and the measure of its goodness is given by a loss function, by changing it, it is possible to change the task to penalize the non-physical be-

havior of the model.
This is a form of regularization, others include adding **constraints** that limit the hypothesis space eliminating non-physical models, or making the model as simple as possible, where the notion of simple is derived from physical notions.

Because Physics-Informed Optimization is a form of physics information used in our work, an entire section that follows is dedicated to *constrained optimization* 1.2.

1.1.3 Physics Informed Neural Networks

PINNs are a recent contribution to the deep learning field [7]. Their goal is to learn physically plausible solutions to differential problems, by exploiting the expressive power of neural networks that are trained to solve supervised learning tasks while respecting any given law of physics.
This method tries to solve 2 problems: *data driven solutions of partial differential equations* and *data-driven discovery of partial differential equations.*

The **Data driven solution** of a partial differential equation

$$f := u_t + \mathcal{N}[u] = 0 \tag{1.1}$$

where $\mathcal{N}$ is a *nonlinear differential operator*, and u can be thought as the solution function $u(t, x)$ where $x \in \Omega$ is a possible state of the system, $t \in [0, T]$.
A nonlinear differential operator is one that takes in a function, differentiates it and applies some nonlinear function to it. The equation above describes a relation between u and its derivatives.
The insight is that the function u_t can be replaced by a neural network, that is a universal function approximator, and can be trained from data minimizing the loss $L_u = ||u' - z||$ where z is the training data represented as an array of states that evolve in time and u' are the learned states, and together minimizing $L_f = ||f||$, to make the *surrogate* solution u respect the governing equation 1.1 as much as possible.
L_f is differentiable because the neural network is, and also $\mathcal{N}$, because even if it is a differential operator, automatic differentiation can be used to make the gradient pass through it.
The solution u to the equation 1.1 can be found even with no data available, by optimizing only L_f, making the technique an effective PDEs solver.

The **Data-driven discovery** of the governing equations of a system is done by first parameterizing them using coefficients λ,

$$f := u_t + \mathcal{N}[u, \lambda] = 0$$

and then learning those coefficients from data. So the overall structure of the equations serve as physics information, but its parameters are learned from data. At the same time u is learned as before.

1.2 Constrained Optimization

A way to enforce prior physical knowledge is to constrain the optimized model to satisfy certain properties.

A classical method for doing that is to introduce a term in the loss function multiplied by a coefficient λ that decides the tradeoff between the discrepancy between the model's prediction and target f and the satisfaction of the constraints g. Because when the optimization process concludes, the constraints can be only partially enforced, they are called **soft constraints**.

$$L(x) = f(x) + \lambda g(x)$$

The problem with soft constraints is that the choice of λ is critical. Too small and the constraint will not be satisfied completely, too large and the optimization cannot pass through unfeasible regions of the search space that often provide a shortcut to good regions that also satisfy constraints. Moreover a large λ creates problems for gradient based methods, where the gradient becomes very large when into unfeasible regions, making the optimization process jump, possibly missing good minima.

A solution would be to enforce constraints directly into the design of the architecture, creating an **hard constraint**, which guarantees that at convergence the model satisfies the constraints.
This requires designing a new architecture for each new constraint considered and that can be difficult and time consuming.

There are instead ways, known for a long time in the optimization community, that make constraints **simple to enforce and hard**. An example of hard constrained optimization in the context of PINNs is found in [30]. There the authors conduct a study on the performance of different optimization techniques to enforce hard constraints on the model. They found empirically that the best technique is that of the augmented Lagrangian.
A similar technique will be now discussed.

Method of Lagrange Multipliers

The goal is to find a λ that is large enough to push the gradient outside the unfeasible region, but small enough to balance exactly the tendency to move inside the unfeasible region when on the barrier between feasible/unfeasible regions. The key insight is that it is possible optimize at the same time the model and λ, by defining the *Lagrangian function*:

$$\mathcal{L}(x, \lambda) = f(x) - \lambda g(x)$$

Then the objective becomes:

$$min_x max_\lambda \mathcal{L}(x, \lambda)$$

This can be thought as a two-player zero-sum game [38], where one player tries to minimize $f(x)$ and the other tries to enforce the constraint, making this a 2 objective optimization.

It can be shown that stationary points of $\mathcal{L}$ are local minima of $f(x)$ that **satisfy** the constraints. But because the stationary points are saddle points, classic minima seeking gradient based methods do not work.

Moreover handling a multi-objective optimization is challenging as one gradient can point in an opposite direction from the other, making the optimization's objective oscillate and converge very slowly.
This has been noticed in [21], where an extensive study of this problem in the context of PINNs is conducted, showing how the generalization capabilities of the technique are penalized as the physical constraint is not respected. They solve the issue by introducing a new technique that modifies the gradients dynamically.
There are many other techniques for improving the convergence of this kind of multi-objective optimization, as discussed in [13].

1.3 Learning to Simulate Physics

For many tasks in which the goal is related to understanding the physics of a system from observed data, maybe because of prediction or system modeling, ultimately a physics simulator is learned.
Often a system is observed making *measures* $\boldsymbol{x}_t$ of some properties of it, as they evolve in time. The ordered sequence of snapshots of the measures of the system is called **trajectory** of states $\boldsymbol{x}_{t_{0:k}} = (\boldsymbol{x}_{t_0}, ..., \boldsymbol{x}_{t_k})$.
A *simulator* $s : \mathcal{X} \to \mathcal{X}$ models the dynamics by mapping preceding states to causally consequent future states [24]. A simulated trajectory $\tilde{\boldsymbol{x}}_{t_{0:k}} = (\tilde{\boldsymbol{x}}_{t_0}, ..., \tilde{\boldsymbol{x}}_{t_k})$ is calculated sequentially starting from the first state and applying s iteratively: $\tilde{\boldsymbol{x}}_{t_{k+1}} = s(\tilde{\boldsymbol{x}}_{t_k})$.
Simulators use dynamics information contained in the current state of the system to update it and get a future state. The assumption is that no previous information is needed. Simulators view the dynamics in a differential way, e.g. a numerical differential equation solver uses these equations to get dynamics information (*time derivatives*) used to update the current state (by *integration*).

A learnable simulator $s_{\boldsymbol{c}}$ uses a parameterized function approximator $d_{\boldsymbol{c}}$ to compute the dynamics information, where $\boldsymbol{c}$ are the parameters that can be learned. The **physics information** here is given by how restricted is the space of functions that can be generated by the approximator, and how these functions respect the a priori information.
The update mechanism can be seen as a function which takes the $\tilde{\boldsymbol{x}}_{t_k}$, and uses $d_{\boldsymbol{c}}$ to predict the next state, $\tilde{\boldsymbol{x}}_{t_{k+1}} = \text{Update}(\tilde{\boldsymbol{x}}_{t_k}, d_{\boldsymbol{c}})$.
For example in learning to simulate a physical dynamics, $d_{\boldsymbol{c}}$ can represent an approximation of differential equations that govern the system, and Update() is a numerical integrator.

1.4 Sparse Identification of Nonlinear Dynamics

System Identification field is concerned with finding mathematical models that capture physical systems from data. Using differential equations leads to models that **generalize** well and are concise and **interpretable**.
The Sindy algorithm [3] finds a concise description of a nonlinear dynamics by searching for a set of sparse **symbolic** differential equations that capture it.
It essentially learns a *simulator* where $d_{\boldsymbol{C}}$ represents a set of differential equa-

tions, that are integrated then by the update mechanism.
It finds one differential equation for each variable $\boldsymbol{y}_i$ of the state of the system $\boldsymbol{y}$, and each equation is a sum of terms:

$$d_{\boldsymbol{C}}(\boldsymbol{y})_i = \frac{d\boldsymbol{y}_i}{dt} = \sum_j \boldsymbol{C}_{ji} f_j(\boldsymbol{y})$$

Each term is a coefficient $\boldsymbol{C}_{ji}$ learned from data, that multiplies a fixed function $f_j(\boldsymbol{y})$. All functions are selected by the user, to form a **dictionary**. That is the principal source of physical information. If the system is understood quite well, the dictionary can contain only few functions as candidates to describe the system, restricting the hypothesis space. If the system is not understood well, or the level of expertise needed to select appropriate functions is lacking, it is possible to let the algorithm decide the right combination of terms needed to describe the system, provided enough data is available.
In order to learn the right coefficients to put in front of the dictionary's functions, every input dynamics $\boldsymbol{X}_{t,i}$ of the training set is differentiated with respect to time, and a regression is performed to fit that.

The other form of physical information is the **sparsity** of coefficients. It has been noticed that systems described simply tend to generalize better, if data is scarce. This is why Sindy tries to zero out the least *important* coefficients, to try to fit the data but without introducing terms that might be there as a result of random fluctuations or chosen by the optimization algorithm arbitrarily as many models may fit the data well if it is scarce.
The notion of importance of coefficients has not just a single definition, there are many ways to select the right terms to eliminate. Those have been studied by the authors of the technique.
This is in essence a form of *regularization*.

1.4.1 Change of coordinates

A mapping from input states $\boldsymbol{y}$ of physical coordinates, to hidden variables $\boldsymbol{h}$ in which the same dynamics can still be described is a change of coordinates. This can be helpful to get a simpler equations that model the system in a more suitable space. This is useful to linearize **nonlinear** dynamics or when simpler dynamics is hidden behind an **high dimensional** input (e.g. images, Partial Differential Equations, particles systems).

For PDEs for example, to discover a good change of coordinates one way is to use a **PCA** (alternatively Singular Value Decomposition or Proper Orthogonal Decomposition), to get eigenfunctions of the dynamics in space, whose eigenvalues encode the most important features of it. Those eigenvalues evolve in time and become the hidden variables. The PCA can be actually seen as a linear, shallow **autoencoder** [12]. This suggests that a full fledged autoencoder can be more powerful as it can generate nonlinear changes of coordinates.

To do that, in [12] an autoencoder is constructed with the Sindy regression built in the latent layer. The encoder learns how to get to the hidden space,

then Sindy is used on the hidden features $\boldsymbol{h}$ to learn $\frac{d\boldsymbol{h}}{\partial t}$, then the hidden dynamics is integrated and the decoder retrieves physical states $\boldsymbol{y}$. The loss is created to account for the reconstruction error, the regularization(sparsity) on the model's parameters, the error between real and calculated $\frac{d\boldsymbol{h}}{\partial t}$.
If a linear latent dynamics is desired, it is sufficient to only have linear terms in the dictionary, doing so is like approximating the Koopman operator.

1.5 Graph Neural Networks

Deep learning has been used with success in a wide range of tasks, that have the characteristic of being represented in *Euclidean space*. There is though an increasing number of applications in which data is generated from non-Euclidean domains that can be represented as graphs.
This is because graphs can model naturally complex relationships and interdependency between objects that doesn't exhibit the classic grid like structure of Euclidean domains.

Deep neural networks can leverage statistical properties of the data such as stationarity and compositionality through local statistics [4]. For example, in computer vision, these properties are exploited by convolutional architectures, so far fewer learnable parameters are needed and *priors* are imposed, decreasing the need for high quantities of data.
There are applications in which the data is inherently non-Euclidean, for instance in social networks, in sensor networks, in neuroscience and in computer graphics. In this cases there is no common system of coordinates or shift-invariance to be exploited, so the key ingredients that make deep methods successful have to be adapted to this *geometric* domain.
A classification of Graph neural networks will be given following the taxonomy and frameworks identified in [35].

1.5.1 Definitions

Graph

A graph is represented as $G = (V, E)$ where V is the set of vertices or nodes and E is the set of edges. An edge from node v_i to v_j is denoted as $e_{ij} = (v_i, v_j) \in E$. The neighbourhood of a node v is defines as $N(v) = \{u \in V | (u, v) \in E\}$. A graph can have d attributes associated with each of its $|V|$ nodes. These attributes are defined as the matrix $\boldsymbol{X} \in \mathbb{R}^{|V| \times d}$ with nodes in rows and attributes in columns.
The graphs considered here are *undirected*, so edges represent only a link between nodes without a direction.
The graph may also have edge attributes $\boldsymbol{X}^e \in \mathbb{R}^{|E| \times c}$ where m is the number of edges and c the number of attributes each.

Spatial-temporal Graph

A Spatial-temporal graph is a graph in which its node's features $\boldsymbol{X}_{t,:} \in \mathbb{R}^{|V| \times d}$ change with time t, describing a *graph signal*.

Graph embedding

A Graph embedding is a representation of a graph in a latent space. The graph is transformed into a low dimensional vector trying to preserve important properties. Two graphs can be mapped into the latent space and their respective vectors' distance tell how similar they are with respect to these properties.

Node embedding

Every node of a graph is mapped to a latent space trying to preserve important properties. Notably the structure of the graph can be preserved, so that the proximity of these mapped points recovers the connectivity of the original graph.

Graph Neural Networks

A Graph Neural Network is an optimizable transformation that can operate on all the attributes of a graph (node, edge, global features) that is invariant to nodes' permutations and produces another graph with the same connectivity but updated embeddings [34].
Several GNN layers are often stacked on top of each other, while latent features are *pooled* (aggregated) to represent higher graph-level features, then predictions are made.

Message Passing Neural Networks

Several Graph neural networks propagate information between nodes, to build hidden representations, reflecting nodes' features, edge features and the *graph connectivity*.
To abstract commonalities of many neural models for graph structured data, the Message Passing Neural Networks framework has been proposed [6].

This framework describes 2 phases: the **message passing** phase and the **readout** or pooling phase.
The message passing phase is repeated for many timesteps T and hidden states $\boldsymbol{h}_v^t$ of node v are updated using the function U_t based on messages $\boldsymbol{m}_v^{t+1}$ coming from the direct neighbours of v according to:

$$\boldsymbol{m}_v^{t+1} = \sum_{w \in N(v)} M_t(\boldsymbol{h}_v^t, \boldsymbol{h}_w^t, e_{vw})$$

$$\boldsymbol{h}_v^{t+1} = U_t(\boldsymbol{h}_v^t, \boldsymbol{m}_v^{t+1})$$

So node's features are first transformed according to M_t, then sent to neighbours, aggregated and used to update the current hidden state.
Because this phase is repeated many times, information from far nodes can be gathered.
The readout phase computes a feature vector for the whole graph using some readout function R according to:

$$\boldsymbol{y} = R(\boldsymbol{h}_v^T | v \in G)$$

M, U and R are learnable functions.
Here the *aggregation function* is the sum over messages, but different ones can

be defined as long as they are *permutation-invariant*. This is because graphs are symmetric under node permutation and a change in nodes order should not change how information is propagated.

1.5.2 Categorization of GNNs

Frameworks

With the features contained in nodes and edges, and the structure of the graph itself as input, GNNs can solve a variety of tasks, that can be categorized depending on the mechanism:

- **Node-level** In this case the regression or classification tasks are performed on each node. The input features are propagated between nodes to build an higher level (hidden) representation of them, then the task is performed on this representation.

- **Edge-level** Given the hidden representation of the 2 nodes connected by an edge the task is to predict the label or connection strength of it.

- **Graph-level** Here a single set of features is extracted for the whole graph, by means of information propagation between nodes and pooling techniques that create a compact representation of the graph. The pooling techniques extract sub structures of the graph and by repeating information propagation and contractions, the graph can be reduced to a single set of features that preserve the properties of interest. This features are then used for regression or classification of the whole graph.

Training Frameworks

GNNs can be trained in different ways depending on the task and availability of data:

- **Semi-supervised learning for node-level classification** For this kind of task there exist techniques that can be supervised only for some nodes (e.g. ConvGNNs) and be robust enough to do the classification correctly for every node. This depends on how information between nodes propagates.

- **Supervised learning for graph-level classification** Here propagation phases are alternated to contraction phases to obtain a compact set of features fed into a multi-layer perceptron and a softmax layer to classify the graph.

- **Unsupervised learning for graph embedding** If no labels are available a graph embedding can be learned in unsupervised fashion. The task is to represent the graph using a low dimensional vector that preserves topological information and important properties. This embedding simplifies then the analysis of the data on the graph with other techniques.

Taxonomy of GNNs

- **Recurrent graph neural networks** The RecGNNs are pioneer works on GNNs. They extract high-level node representations by applying the same function recurrently over nodes in a graph.
  They diffuse nodes information, by updating nodes' state based on that of neighbours. This updated state is then propagated many other times and states are updated accordingly until a stable equilibrium is reached. This *update function* has parameters that can be learned and is constructed so that this procedure converges.

- **Graph Autoencoders** GAEs are neural architectures that map nodes from an input graph to a latent space and then decode this representation to obtain graph information. They can be used for graph embedding or graph generation. GAEs try to embed the nodes so that the structure of the graph is preserved (as described by the adjacency matrix), moreover they also encode node feature information. To do so they use a couple of activated Graph Convolution Network layers in the encoding step, then decode the adjacency information by calculating the similarity of couples of node embeddings.

1.5.3 Convolutional graph neural networks

It is a special kind of message passing neural network where each node's hidden features are a linear combination of neighbours' and its own features, with coefficients that depend on the *structure* of the graph.
GCNNs are a generalization of classic Convolutional neural networks, that instead operate on Euclidean domains which posses special properties that not present on graph domains. In particular a special definition of **convolution** on graphs must be given:
Graphs have less structure than grid-like domains such as images, where instead the number of neighbours of a pixel is constant and they can be ordered, so a specific coefficient can be associated to a neighbour based on its *direction*. The nodes in a graph are not assumed to be provided in any specific order, so the latent representations of 2 isomorphic graphs with the same features should be identical. An implication of this is that graph convolutional filters are oblivious of direction or *isotropic* [25].

Spatial-based ConvGNNs

In this approach the convolution operation is generalized on graphs by combining the neighbouring features of each node [26]:

$$\boldsymbol{m}_v^{t+1} = \bigoplus_{w \in N(v)} c_{vw} M_t(\boldsymbol{h}_w^t)$$

$$\boldsymbol{h}_v^{t+1} = U_t(\boldsymbol{h}_v^t, \boldsymbol{m}_v^{t+1})$$

where $\bigoplus$ is a aggregation operator that is invariant to the permutation of nodes. The coefficients c_{vw} are often derived directly from the adjacency matrix, so they are not parametric. In some works (e.g in Graph Attention Networks) they are

learned to express the importance of the link between v and w.
The message passing is more general than convolution, and considers both the features of the node and of the neighbour to compute each message in the aggregation.
Intuitively, one convolution step smooths the hidden representations locally along the edges of the graph and ultimately encourages similar predictions among locally connected nodes.

Spectral-based ConvGNNs

Spectral-based methods are mathematically well founded in graph signal theory. They exploit the *convolution theorem* to define the convolution, after the concept of *graph Fourier transform* has been established.
This transform derives from the eigen decomposition of the graph's Laplacian matrix that carries information about the aggregated difference of each node's features and its neighbours' features.
The convolution theorem then states that

$$\boldsymbol{x} *_G \boldsymbol{c} = \mathcal{F}^{-1}(\mathcal{F}(\boldsymbol{x}) \odot \mathcal{F}(\boldsymbol{c}))$$

where $\boldsymbol{x} \in \mathbb{R}^n$ represents a feature of each node, $\boldsymbol{c}$ is the learnable convolution kernel, $\mathcal{F}$ is the graph Fourier trasform and $\odot$ is pointwise multiplication.
This definition of convolution is very sensible to the graph's structure, so a learned kernel cannot applied to a graph with different structure and obtain similar results. Moreover the computation of the graph Fourier transform is quite slow.
To improve on this several techniques have been proposed.
ChebNet approximate the kernel using the Chebyshev polynomials, in this way it becomes localized in space.
Graph Convolutional Network is an approximation of Chebnet where only the first order neighbours are considered when calculating the value of the convolution for a node:

$$h_v = \sigma(\boldsymbol{c}_0 \boldsymbol{h}_u + \boldsymbol{c}_1 \sum_{u \in \{N(v) \cup v\}} \bar{\boldsymbol{A}}_{v,u} \boldsymbol{h}_u)$$

where $\bar{\boldsymbol{A}}$ is the normalized adjacency matrix. Here the multiplication by the coefficients is done after the aggregation, so every neighbour contributes equally to a node's hidden state.

1.5.4 Spatial-temporal graph neural networks

If the features of a graph evolve in time they describe a *graph signal*. Such a signal depends both on time and on the graph's structure, as a node's dynamics may depend on that of connected neighbours. The task of STGNNs can be the *prediction* of future features value or label.
For example a road network can be described as a graph while the dynamic traffic conditions can be captured by a graph signal that describes the congestion at each arc. These conditions depend on the road structure and evolve in time following past congestion.

Most architectures for STGNNs model the spatial dependency using graph convolutions and computing a hidden representation that evolves in time using either **Convolutional Neural Networks** or **Recurrent Neural Networks**.
A graph signal on nodes can be represented by a matrix that changes with time $\boldsymbol{X}_{t,:,:} \in \mathbb{R}^{|V| \times d}$ where $|V|$ is the number of nodes in the graph, d the number of features for each node.
Using **CNNs**, for each timestep t a feature propagation is performed using a graph convolutional layer that operates on each graph temporal *snapshot* $\boldsymbol{X}_t$, then a 1-D convolution is performed in time independently for each node, in this way past hidden representations are aggregated together to predict one snapshot in the future.

Recurrent neural networks are used in the machine learning community to analyze sequences and doing prediction or classification. These tasks are performed on extracted hidden states $\boldsymbol{h}$ that are found by progressively accumulating the sequence, by means of a learned function. A simple example of a RNN, operating on a signal $\boldsymbol{x}_t$ can be:

$$\boldsymbol{h}_t = \sigma(\boldsymbol{C}\boldsymbol{x}_t + \boldsymbol{D}\boldsymbol{h}_{t-1} + \boldsymbol{b})$$

where $\boldsymbol{C}$, $\boldsymbol{D}$, $\boldsymbol{b}$ are learnable parameters.
A RNN can be modified to use graph spatial features instead, extracted using a graph convolution technique:

$$\boldsymbol{H}_t = \sigma(GConv(\boldsymbol{C}; \boldsymbol{X}_t, \boldsymbol{A}) + GConv(\boldsymbol{D}; \boldsymbol{H}_{t-1}, \boldsymbol{A}) + \boldsymbol{b})$$

where $\boldsymbol{A}$ is the adjacency matrix. Here instead of doing a matrix multiplication between signal features and learned parameters, graph convolution is used. This is more appropriate, as in this way spatial features are propagated following the graph's structure.

An example of Recurrent Graph Neural Network is the Graph Convolutional Recurrent Network (GCRN) that combines a LSTM network with ChebNet.
The Diffusion Convolutional Recurrent Neural Network (DCRNN) instead works using a different type of convolution. The task is to learn a function $f(.)$ that maps T' snapshots of the graph signal in the past to T in the future

$$(\boldsymbol{X}_{t-T'}, ..., \boldsymbol{X}_t) \xrightarrow{f(.,G)} (\boldsymbol{X}_{t+1}, ..., \boldsymbol{X}_{t+T})$$

In order to do that the idea is to consider a **Gated Recurrent Unit** to encode the temporal information of the signal from the graph, and use a **Diffusion Convolutional Layer** to learn spatial information.
A DC layer learns a probability $P_{i,j}$ for each arc, that represents the likelihood of diffusion from node i to node j, based on graph random walks theory. The convolution is then a sum of neighbours' features (each feature separately) weighted by the diffusion probability.

RNNs and physics

RNNs are more problematic than CNNs when dealing with sequences, as they are less parallel. In fact the value of the hidden state $\boldsymbol{h}_t$ depends on $\boldsymbol{h}_{t-1}$ so the

computation must follow the sequential order of the sequence and this causes a performance hit. Moreover a small error on the learned parameters is propagated recurrently until the last hidden state, growing exponentially. This causes the *exploding gradient* problem and makes this type of networks difficult to train. Even with these shortcomings, recurrent neural networks provide a useful blueprint for learning *physical dynamics*. In fact a physical system that evolves in time is usually described by differential equations that can be used to map a state into the next one, just like RNNs can learn how to map previous signal's states into the next. In this case the RNNs **learns implicitly the differential equations** of the system, and becomes a *physics simulator* such that

$$\tilde{\boldsymbol{x}}_t = \boldsymbol{h}_t = s(\boldsymbol{x}_{t-1})$$

This analogy is particularly interesting when dealing with physical dynamics where the states evolution is not completely *observable*. If there are unobserved states, the supervision is not available for the whole sequence. In this case they become *hidden* states that should be reconstructed following a learned recurrent relation between each other. For example if $\boldsymbol{x}_{t-k}$ is the last observed state

$$\tilde{\boldsymbol{x}}_t = \boldsymbol{h}_t = s^k(\boldsymbol{x}_{t-k})$$

with s^k meaning s has been composed with itself k times, having a sequence $\boldsymbol{h}_{t-k+1:t-1}$ of intermediate results.

Chapter 2

The problem

In this chapter Epidemiological modelling is introduced as well as the famous SIR model. Then the problem tackled in this thesis can be introduced, using the concepts explained in the Background. A more precise definition of Diffusion dynamics on graphs is then provided, to better state the problem in terms of Physics simulator 1.3 that works on graphs. At the end the datasets used to do experiments are presented under NDlib.

2.1 Epidemiological modelling

Epidemiology is the study of the distribution and causes of health and disease conditions in specified populations. [39]
Decisions regarding the control of infectious diseases often have to be made despite an imperfect understanding of how a disease spreads and develops. Epidemiological models offer a means to address these uncertainties by combining available information from experimental studies and the opinion of experts to gain insight into the dynamics of infection and disease control [2].

A **model** is a representation of a physical process that is designed to increase the understanding of that system. Models can be used to understand how external influences change the system's behaviour and make predictions about its dynamics.
Epidemiological models **predict** patterns of spread under different conditions and give information on how to control the disease, especially when there is limited practical experience with it. Moreover they can be used to study the factors linked to the disease and how they affect it.

The approach for modelling changes depending on how well the epidemiology of a **disease is understood**, the amount and **quality of data available** and the **skill of the modellers** involved.
Models can be *deterministic* if they generate only an expected outcome, *stochastic* if they generate a range of possible outcomes with some probability distribution. Stochastic models are more difficult to construct but they can be used to investigate the likelihood of different outcomes.
Data quality and quantity is particularly important. The time precision of

a model depends on how often the disease is measured. If spatial information is available, locations and distances are taken into account in disease transmission computations. Modelling invariably involves tradeoffs in terms of the complexity versus the availability of data. So if small datasets are available or they are taken from specific populations, the model should be simple to be able to generalize and an expert should provide a priori information to be built into the model.

Traditionally models assume simple population structures with homogenous mixing of the people and simplified transmission parameters, they do not necessarily account for spatial or social dimensions. Researches have shown [2] that spatial effects, population heterogeneity and social behaviour can affect the dynamics of the disease.
Recent advances on remote sensing and data analysis methods make it possible to simulate the effects of a disease on a much smaller scale. For example **social networks** can model the interactions between people and capture complex patterns that underlie disease transmission. Another technique is to build a large scale agent-based model where each entity behaves following a predefined set of rules.

2.1.1 The SIR model

Compartmental models are a modelling technique that divides the population into compartments bases on states in which a person can be. The evolution of a disease in time makes people change state, so the number of individuals in a compartment varies. These models use deterministic or probabilistic approaches to predict how many people will be in every compartment in a future time.
A common deterministic approach is to describe the dynamics of each compartment using **differential equations**. In this case the system can be regarded as *physical*, and its governing equations follow the biological laws of the infection.

The SIR model is useful to analyse pandemic diseases that spread from person to person. It divides the population into 3 compartments:

- **S**: Susceptible individuals. When a susceptible person enters in contact with an infected person, the susceptible individual becomes also infected.
- **I**: Infected individuals. These individuals can infect other people and grow the spread of the pandemic.
- **R**: Recovered individuals. People who have contracted the disease and are recovered or died.

The number of individuals is assumed to be *constant*, so no people are born, and those who die do so because of the disease. Moreover a person who recovers can never infect again, this is true only for a limited number of diseases and for a limited amount of time.
There are different compartment based models, such as the SIS where a person that recovers can be susceptible again, that is useful for some diseases such as the influenza. Another is the SIRD that distinguishes people recovered and

people that died as a result of the infection. The SEIR model considers that for many important infections, there is a significant latency period during which individuals have been infected but are not yet infectious themselves. During this period the individual is in compartment E (for exposed). The SIR model is chosen as the subject of study for this thesis as it is one of the simplest to analyse.

The 3 SIR variables vary over time as the infection progresses, so they can be regarded as functions of time: $S(t)$, $I(t)$, $R(t)$. The change of each variable can be expressed as only dependent on the state of the system, e.g. if the number of infected people is high, also the rate at which people recover will be high, this disregarding time. For this reason the evolution of the variables can be expressed by a system of ordinary differential equations:

$$\begin{cases} \frac{dS}{dt} = -\beta \frac{IS}{N} \\ \frac{dI}{dt} = \beta \frac{IS}{N} - \gamma I \\ \frac{dR}{dt} = \gamma I \end{cases} \tag{2.1}$$

where N is the total number of people in the population, β is the infection probability, γ is the recovery removal probability.

An interesting **constraint** that can be derived from this equations is that:

$$\frac{dS}{dt} + \frac{dI}{dt} + \frac{dR}{dt} = 0$$

by integrating both parts it can be seen that:

$$S(t) + I(t) + R(t) = N \qquad \forall t \geq 0 \tag{2.2}$$

the sum of the variables for each time instant is **constant**. Moreover the initial value of each variable is assumed to be positive or equal to 0. Using this fact together with the previous constraint it can be shown that:

$$S(t), I(t), R(t) \in [0, N] \qquad \forall t \geq 0 \tag{2.3}$$

These constraints are general enough to be present in a wide range of dynamics, that means that their enforcement is useful for many problems in which the task is to find a model for such dynamics, as will be discussed later.

2.2 Problem formulation

Machine learning can be used to help with epidemiological modelling, where a disease is not fully understood or it is difficult to capture using mathematical means.
A model can be learned using ML techniques. For example a **simulator can be learned** that evolves some properties of the system in time. To learn a SIR model the three important variables S,I,R can be extracted from data and their evolution used to train a simulator that captures their fundamental dynamics, so that the simulation can be brought forward in time to **make predictions**.

Many space based models used in epidemiology, follow a predefined set of rules valid on a small scale, to simulate the infection on a much larger scale, where the dynamics are less understood. For example in a social network, represented with a **graph**, the dynamics of interaction between near people can be known, as well as the structure of the network, then, by using a simulation that exploits this knowledge, the large scale effects can be studied. Once a precise simulation is available, specific properties of it can be extracted (e.g. the S,I,R features) without loss of information.
Graph neural networks can be used to learn these small scale interactions and *simulate* their effects on a larger scale.

In this thesis, the problem of *epidemiological forecasting* is attacked by means of *physics informed machine learning*. The goal is to learn an epidemiological model on graph that forecasts the probability for every person of being in a compartment (S,I,R). This model is trained on a *limited* set of data and should be able to forecast the spread of COVID-19 forward in time. Physics information is enforced into the model by the choice of the architecture and the application of constraints. This is useful for dealing with the scarce data available and make physically plausible predictions. A special focus is given to the *interpretability* of the model, that enables insight into the data and encourages a more informed research on the topic.

The topic of physics informed machine learning will now be introduced and the main ideas discussed, citing also the works that explored them. This next section is useful for explaining why physics information is important and to introduce then the methods used to solve the problem.

2.3 Diffusion dynamics on graphs

Graphs are a very powerful tool for reducing complex phenomena to a common analytical framework whose basic components are nodes and their relationships. A special case of modelling is that of *dynamic* phenomena, meaning realities in which the relationships between agents as well as their status change with time. We are interested in the diffusion dynamics on graphs of a viral disease, where the nodes represent people and the edges their possible contacts, so the virus spreads between neighbouring nodes. There are several elements that determine the patterns of spreading through a group of people: the properties of the virus (its contagiousness, its severity and infection speed), the structure of the social network and the mobility patterns of people.
Here the graph is supposed to be static, so no people can be added or removed and their relationships remain the same through time. A person is characterized by a compartmental state and can be either in the **S**(Susceptible), **I**(Infected) or **R**(Recovered) state. This is encoded by 3 features that can be either 0 or 1. The dynamics of the infection is then a graph signal $\boldsymbol{X}_t \in \mathbb{R}^{|V|\times 3}$ where $|V|$ is the number of nodes(people) in the graph. An **initial state** of the infection is $\boldsymbol{X}_0$, this describes the compartment in which each person is at the beginning of the study of the dynamics.
We suppose that $dt = 1$ so that each timestep advances by 1 time unit. Each successive state of the infection is a graph *snapshot* $\boldsymbol{X}_{t+1}$ that depends on the

previous one in time $\boldsymbol{X}_t$, this dependency can be deterministic or not, in which case it depends also on chance.
Our task is to learn a *simulator* s that advances the infection on graph forward in time $\tilde{\boldsymbol{X}}_{t+1} = s(\tilde{\boldsymbol{X}}_t)$ by *diffusing* it from each person to its neighbours.

2.4 NDlib

To train the simulator a source of data is required. In this case a graph simulation library called NDlib is used [10]. This library is capable of handling a number of different epidemic models among which the SIR, that will be used.
The first snapshot of the simulation is instantiated by having a small fraction of people infected and the rest susceptible. At each simulation step the neighbours of each person are considered and those infected can infect also the person with probability β. If a person is already infected he/she can be removed with a probability γ. A person cannot transition from the susceptible state to the removed one directly. A static graph in which the simulation takes place can be provided.
If a large enough number of people is considered, and the number of individuals for each compartment are aggregated, a dynamics emerges that can be described by the classic SIR equations (2.1) with the same β and γ used in the simulation.

Chapter 3

Related works

Several techniques have been studied to forecast COVID-19 [33].

An **autoregressive** model operates on a time series trying to forecast the next value based on a linear combination of previous values plus a stochastic term. This type of model has been applied to the problem in different forms, starting from a simple moving average [17], to the more advanced autoregressive integrated moving average [22], that is well known in statistics and economics to treat time series with *seasonality*.

Regression is a statistical tool used to model a dependent variable given a set of features or explanatory variables. This technique is useful for forecasting if a set of interesting predictors is available that can characterize the future progress of the pandemic [15].

Several models based on **differential equations** have been discussed in literature including the logistic growth model and the deterministic compartmental models (such as SIR, SEIR and SIRD). They model the change in the number of cases and in the number of people present in a compartment respectively as a function of the current state of the infection. These models contain parameters that affect the dynamics of the infection that can be learned from even a small amount of data. For this reason they can predict the pandemic well in the first phase on the infection, when data is still scarce [14].

Genetic Programming has been used to construct equations to describe the dynamics of the virus [23]. These techniques work by combining base formulas together to form complex enough models to describe the infection dynamics. The coefficients of such models are learned too during the search process. Genetic Programming has been found to be reliable for prediction and learns models that are *interpretable*.
This method is similar to Sindy, but instead of having a fixed number of functions to be chosen and summed together, Genetic programming can also compose them, making equations more powerful. The learning process is much slower though and relies on many hyperparameters.

Several **machine learning** techniques have been employed for forecasting COVID-

19. They have the advantage of not making many assumptions on the dynamics and instead learn directly the model from data, if enough is available.
For example [18] use an LSTM recurrent neural network to make predictions based on the time history of cases. In another work [31] neural networks are used in conjunction with autoregressive models and special optimization techniques.

Graph neural networks use graphs to reason on social relationships between people and simulate how the virus spreads on social networks, learning more detailed models.
In the work [32] a dataset is used that counts the number of cases per region, and the interconnections between regions are encoded by a graph. Message passing layers plus an LSTM encode the spatial-temporal information about the pandemic to then predict the future number of cases. Moreover they employ *transfer learning*, after noticing that past information of the evolution of the pandemic on another country may share patterns with the county under analysis. So the model is initialized with parameters learnt from the other country.
Instead [20] takes into account also the mobility of people across regions because this represents a good indication on how the virus spreads. The flow of people is modelled with a weight on the edges that changes with time for inter-region mobility and a weight on the nodes for mobility into the region. Instead of using a recurrent neural network to capture the temporal dependency of features, a simple concatenation in time plus activation is used. So a fixed number of flow snapshots are used to predict the next one.
In [28] mobility data is combined to epidemiology data, by fusing the features coming from a graph neural network and a probabilistic model respectively, making it possible to use different forms of data together to improve the predictions.

The closest work to our own is [29]. It tries to predict more than one time step in the future by using graph attention networks together with a **physics informed** loss. This loss is composed of 2 terms: a short term penalty and a long term one. The short term error is calculated with respect to the target graph infection, the long term error is calculated from the SIR model by measuring how close the prediction is to a physically correct one.
This work explores the use of a loss term added to regularize, whereas by using sindy the regularization is given by the sparsity of coefficients, we use the term in the loss to enforce constraints instead. The SIR model is used in this work to determinate how physically plausible is the solution, instead we don't inform the model using the SIR directly but only a set of dictionary functions plus the constraints. Last our model is interpretable, in this work, because neural networks are used, it is far more difficult to understand what the model learned.

Chapter 4

Contributions

In this chapter our contributions are described. At the beginning, the topics are discussed in a more formal way, then Sindy Graph Implementation describes how the various techniques are implemented.

In Sindy problem formulation, it is stated how Sindy can be used to learn an infection dynamics, without discussing how to integrate graph structure information just yet. The problem Sindy tries to solve is formalized without implementation details.
Then in Constraints we list the constraints added to the learned dynamics and how to define them formally. These definitions are then used in Constraints on infinite states to present the difficulty of making sure those constraints are respected. Then some techniques that can be used to find the right constrained problem formulation are presented.
When the formulation is found, the non linear constrained problem should be solved. A well known solution method is presented in Constraints and gradient descent.
Sindy is a technique that encourages sparse symbolic models. A way in which it can do it is described in Thresholding.
After these introductions, our technique called Sindy Graph is introduced. It is here explained how Sindy can be informed with graph structure, to learn a dynamics on graph.
The last section is Sindy Graph Implementation. Here the techniques described before are enriched with implementation details. In particular it is explained how to deal in practice with Aggregated supervision, that is a central problem when learning from data coming from statistics over large populations.

4.1 Sindy problem formulation

Given a time signal $\boldsymbol{X}_{t,i}$ (where $\boldsymbol{X}_{t,:}$ is abbreviated with $\boldsymbol{X}_t$) that varies discretely with time t for the variables $i \in \{S, I, R\}$, the goal of this technique is to learn a *simulator* $s_{\boldsymbol{C}}$ that advances the signal by one time step in the future $\tilde{\boldsymbol{X}}_{t+1,:} = s_{\boldsymbol{C}}(\boldsymbol{X}_t)_:$.
Internally it learns a *difference function* $d_{\boldsymbol{C}}$ with learnable parameters $\boldsymbol{C}_{:,:}$ so that $s_{\boldsymbol{C}}(\boldsymbol{X}_t), := \boldsymbol{X}_t + d_{\boldsymbol{C}}(\boldsymbol{X}_t) * dt$.

The difference function approximates the real differences of the signal between successive time steps: the mean square error it has with respect to the real signal's differences is to be minimized. More formally the objective is

$$min_{\boldsymbol{c}} \qquad \sum_{t,i} (d_{\boldsymbol{C}}(\boldsymbol{X}_t)_i - \boldsymbol{X}'_{t,i})^2 \tag{4.1}$$

where $\boldsymbol{X}'_t = \frac{\boldsymbol{X}_{t+1} - \boldsymbol{X}_t}{dt}$ is a discrete approximation of the time derivative of the input signal.
The difference function $d_{\boldsymbol{C}}$ actually learns a set of **differential equations**, one for each input variable $i \in \{S, I, R\}$:

$$\begin{cases} \frac{dS}{dt} = d_{\boldsymbol{C}}(\boldsymbol{y})_S \\ \frac{dI}{dt} = d_{\boldsymbol{C}}(\boldsymbol{y})_I \\ \frac{dR}{dt} = d_{\boldsymbol{C}}(\boldsymbol{y})_R \end{cases} \tag{4.2}$$

Each equation is a linear combination of non linear functions, took from a predefined dictionary:

$$d_{\boldsymbol{C}}(\boldsymbol{y})_i = \sum_j \boldsymbol{C}_{ji} f_j(\boldsymbol{y}) \tag{4.3}$$

where $f_j(\boldsymbol{y})$ is the function j of the dictionary evaluated on a state of the system $\boldsymbol{y} = (\boldsymbol{y}_S, \boldsymbol{y}_I, \boldsymbol{y}_R)$.
Because the value of the functions $f_j(\boldsymbol{X}_t)$ can be pre-computed for each state of the input signal, $d_{\boldsymbol{C}}(\boldsymbol{X}_t)_i$ is a linear function of the learned coefficients $\boldsymbol{C}$. Because also the input differences $\boldsymbol{X}'_t$ can be pre computed, (4.1) represents a **linear least squares problem**.

4.2 Constraints

Adding constraints to the reconstructed signals $\tilde{\boldsymbol{X}}$ is a way of adding *physical information* that is known a priori and helps to find a good candidate model.
Because we analyse the pandemic spread we know that a the features S, I, R must be in the range $[0, 1]$, with 1 indicating the maximum number of people. Moreover the sum of people divided in the 3 compartments is supposed to remain constant.
An important consequence of bounding the solutions is that they **cannot grow exponentially** as time goes on. In fact unconstrained differential equations most likely are not bounded and for their nature their solutions grow very fast, separating quickly from the true behaviour. This means that without constraints the model most likely generalizes bad.

In order to make sure that the sum of the reconstructed signal's components $\tilde{\boldsymbol{X}}_{:,S}$, $\tilde{\boldsymbol{X}}_{:,I}$, $\tilde{\boldsymbol{X}}_{:,R}$, remains **constant** as the simulation unfolds, it can be noticed that this is equivalent to set the sum of the respective differences to 0:

$$d_{\boldsymbol{C}}(\boldsymbol{X}_t)_S + d_{\boldsymbol{C}}(\boldsymbol{X}_t)_I + d_{\boldsymbol{C}}(\boldsymbol{X}_t)_R = 0 \qquad \forall t \leq T$$

this constraint is easy to add to the initial problem (4.1) as it doesn't require the numerical integration of the difference function.

A much more strict constraint can be enforced though, because no deviation from the constant sum should be possible for any *feasible state* of the system $\boldsymbol{y} \in SetS$ where $SetS = [0,1]^3 \cap \{\boldsymbol{y} | \boldsymbol{y}_S + \boldsymbol{y}_I + \boldsymbol{y}_R = 1\}$.
The constraint becomes:

$$d_{\boldsymbol{C}}(\boldsymbol{y})_S + d_{\boldsymbol{C}}(\boldsymbol{y})_I + d_{\boldsymbol{C}}(\boldsymbol{y})_R = 0 \qquad \forall \boldsymbol{y} \in SetS \tag{4.4}$$

The **positivity constraint** is added to make sure that starting from a feasible initial condition, the simulated signal never goes below 0, for any of its components S,I,R. To avoid expressing this constraint as a numerical integration, a formulation based on the difference function is given.
The insight is that for a simulated variable $\tilde{\boldsymbol{X}}_{:,i}$, the only way to go below 0 after some time, if starting from a feasible initial condition, is to have negative derivative when crossing 0. So by imposing a derivative greater or equal to 0 when the variable is 0, the signal for that variable must slow down when close to 0 or increase and never become negative.
The constraints are 3, one for each variable S,I,R, and are expressed as:

$$\begin{array}{ll} d_{\boldsymbol{C}}(\boldsymbol{y})_S \geq 0 & \forall \boldsymbol{y} \in SetP_S \\ d_{\boldsymbol{C}}(\boldsymbol{y})_I \geq 0 & \forall \boldsymbol{y} \in SetP_I \\ d_{\boldsymbol{C}}(\boldsymbol{y})_R \geq 0 & \forall \boldsymbol{y} \in SetP_R \end{array} \tag{4.5}$$

where

$$SetP_i = \{\boldsymbol{y} | \boldsymbol{y}_i = 0, \boldsymbol{y}_j \in [0,1] \forall j \neq i\} \cap \{\boldsymbol{y} | \boldsymbol{y}_S + \boldsymbol{y}_I + \boldsymbol{y}_R = 1\}$$

Because the sum remains constant to 1 and the state's variables remain positive, if the problem is constrained it is also true that each variable must be ≤ 1.

By combining constant sum and positivity constraints the simulator takes in input a feasible state and outputs another feasible state. By iterating the simulation, the trajectory of states remains inside the feasible set $SetS$ for any number of steps. So $SetS$ is **closed** under $s_{\boldsymbol{C}}$.

4.3 Constraints on infinite states

The problem with the constraints (4.4) and (4.5) is that in order to be able to optimize in practice the objective by enforcing them, they should not be defined on an infinite number of initial conditions $\boldsymbol{y}$. Three solutions have been identified to solve this issue.

4.3.1 Random sampling

The first solution is to **discretize** the sets $SetS, SetP_S, SetP_I, SetP_R$ into a finite number of elements taken at random, to represent the real sets as well as possible. So for example $RandSetS$ is built from $SetS$ by sampling uniformly n points at random.
The constrained optimization problem then becomes:

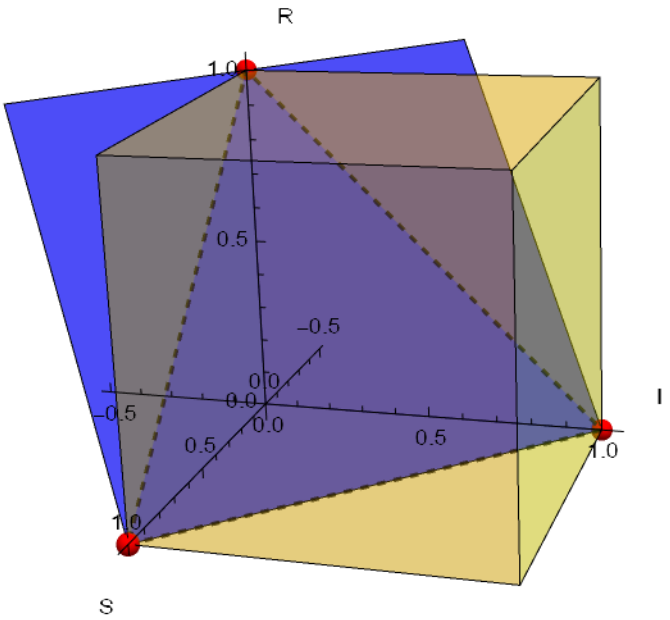


Figure 4.1: Feasible states in SetS. They are given by the intersection of a cube where S,I,R are in the range [0,1] and a plane in which the features sum to 1. The region is a triangle.

$$min_{\boldsymbol{C}} \qquad \sum_{t,i} (d_{\boldsymbol{C}}(\boldsymbol{X}_t)_i - \boldsymbol{X'}_{t,i})^2 \tag{4.6}$$

s.t.

$$d_{\boldsymbol{c}}(\boldsymbol{y})_S + d_{\boldsymbol{c}}(\boldsymbol{y})_I + d_{\boldsymbol{c}}(\boldsymbol{y})_R = 0 \qquad \forall \boldsymbol{y} \in RandSetS$$

$$d_{\boldsymbol{c}}(\boldsymbol{y})_S \geq 0 \qquad \forall \boldsymbol{y} \in RandSetP_S$$

$$d_{\boldsymbol{c}}(\boldsymbol{y})_I \geq 0 \qquad \forall \boldsymbol{y} \in RandSetP_I$$

$$d_{\boldsymbol{c}}(\boldsymbol{y})_R \geq 0 \qquad \forall \boldsymbol{y} \in RandSetP_R$$

The problem with this solution is that the states space of the system can become large if the number of features is high (as it is the case when constraining sindy graph), the random samples can become sparse into the space, leading to large volumes of states that are left unconstrained.

4.3.2 Minima seeking

To improve on Random sampling limitations, a second solution is proposed. The idea is to define a parallel problem to solve that asks for the **single feasible state $\boldsymbol{y}_k$** that violates a constraint the most (or is closest to do so). There will be one critical state for each of the 4 constraints. By asking that the difference function evaluated on this state doesn't violate the constraint considered, by definition the function evaluated on any other feasible state will not violate either.

An interesting observation is that for the positivity and constant sum types of constraints, the feasible state that violates them the most can be found by **searching for the minimum** of a function.
For example to make sure that a positivity constraint is satisfied all the values of the difference function evaluated on $SetP_i$ must be positive. It is sufficient to make sure that the minimum of the difference function is positive. A similar reasoning is possible for the constant sum constraint and even for bound constraints different from 0.

The new problem searches for a state $\boldsymbol{y}_k$ for each constraint $k \in \{S0, S1, P_s, P_i, P_r\}$ that minimizes (or maximizes) the difference function.

$Problem(a)$	$Problem(b)$
$min_{\boldsymbol{C}} \quad \sum_{t,i} (d_{\boldsymbol{C}}(\boldsymbol{X}_t)_i - \boldsymbol{X'}_{t,i})^2$	$min_{\boldsymbol{y}} \quad - d_{\boldsymbol{C}}(\boldsymbol{y}_{S1}) + \sum_{k \neq S1} d_{\boldsymbol{C}}(\boldsymbol{y}_k)$
$s.t.$	
$d_{\boldsymbol{C}}(\boldsymbol{y}_{S0})_S + d_{\boldsymbol{C}}(\boldsymbol{y}_{S0})_I + d_{\boldsymbol{C}}(\boldsymbol{y}_{S0})_R \geq 0$	$\boldsymbol{y}_{S0} \in SetS$
$d_{\boldsymbol{C}}(\boldsymbol{y}_{S1})_S + d_{\boldsymbol{C}}(\boldsymbol{y}_{S1})_I + d_{\boldsymbol{C}}(\boldsymbol{y}_{S1})_R \leq 0$	$\boldsymbol{y}_{S1} \in SetS$
$d_{\boldsymbol{C}}(\boldsymbol{y}_{Ps})_S \geq 0$	$\boldsymbol{y}_{Ps} \in SetP_S$
$d_{\boldsymbol{C}}(\boldsymbol{y}_{Pi})_I \geq 0$	$\boldsymbol{y}_{Pi} \in SetP_I$
$d_{\boldsymbol{C}}(\boldsymbol{y}_{Pr})_R \geq 0$	$\boldsymbol{y}_{Pr} \in SetP_R$

In order to solve in practice this double problem, gradient descent techniques can be used. A solution is to alternate the calculation of the gradient for the first problem and update of parameters $\boldsymbol{C}$ and calculation of gradient for the second problem and update of the minimum candidates $\boldsymbol{y}_k$. In this way the critical states that are used to constrain the $Problem(a)$ are updated as its objective is optimized.
The problem of using gradient descent to find the critical states is that it can converge to **local minima**. If this happens, to find a non-critical case, the constraining will not be sufficient.

A solution is to introduce multiple candidate critical states for each constrain $\boldsymbol{Y}_{:,k} \in Set_k$ initialized at random, and each should search for a local minima in parallel. In this way even if there are multiple local minimums they should be reached by at least one candidate state, given that **enough candidate states** are introduced. The number of candidate states needs to depend on the complexity of the difference function, so simpler functions are more easily constrained.
Moreover searching for minima dynamically as the first problem is optimized introduces a set of added challenges, for example there are more hyper parameters to set and the interaction between the 2 problems is not clear, the **convergence** of this double problem is more difficult to study.

4.3.3 Restricting dictionary functions

Finding the minimum of the difference function can be problematic if no properties about it are known. Because in Sindy the difference function is given by linear combinations of dictionary functions, if these functions have known properties this can simplify the search of minima and thus the constraining of the problem.

A **Harmonic function** is a twice continuously differentiable function $f : U \to \mathbb{R}$ where U is an open set of $\mathbb{R}^n$ that satisfies the Laplace equation

$$\nabla^2 f = 0$$

everywhere on U.
This class of functions have interesting properties. First a linear combination of harmonic functions is still an harmonic function.
Another property is that if K is a nonempty *compact subset* of U (so it is closed and bounded) then f restricted to K attains its **maximum and minimum on the boundary** of K. Now, because the boundary of K has 1 degree of freedom less than U, the minimum can be found in the boundary decreasing the search space of 1 dimension. If random sampling is used, the density of points is increased, making the constraints more precise.

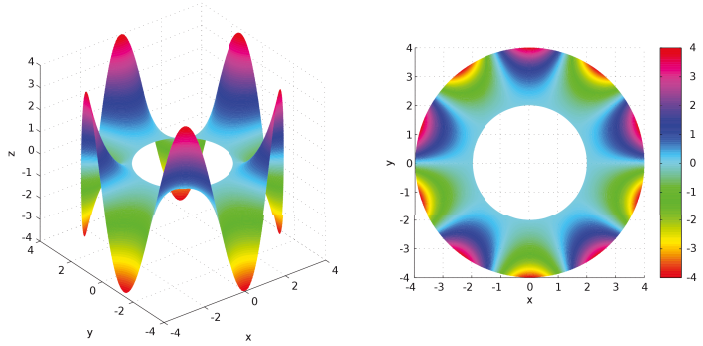


Figure 4.2: A harmonic function on $\mathbb{R}^2$. U is an annulus and the minimum and maximum are on the inner and outer circumferences.

A **Multilinear polynomial** is a multivariate polynomial $p : \mathbb{R}^n \to \mathbb{R}$ in which no variable occurs to a power of 2 or higher. It has the property that if all variables are held constant except for one, the function becomes linear on that variable.
For example, the multilinear polynomial of 3 variables is:

$$p(\boldsymbol{y}_0, \boldsymbol{y}_1, \boldsymbol{y}_2) = c_0 + c_1\boldsymbol{y}_0 + c_2\boldsymbol{y}_1 + c_3\boldsymbol{y}_2 + c_4\boldsymbol{y}_0\boldsymbol{y}_1 + c_5\boldsymbol{y}_0\boldsymbol{y}_2 + c_6\boldsymbol{y}_1\boldsymbol{y}_2 + c_7\boldsymbol{y}_0\boldsymbol{y}_1\boldsymbol{y}_2$$

Multilinear polynomials are *harmonic functions* that have additional properties. They arise from the multilinear interpolation of the vertices' values of an **hyper rectangle**. In this kind of interpolation, when moving a point on an edge connecting 2 vertices of the hyper rectangle, its value interpolates linearly between

the values of the extremes. This is an intuition on an interesting property of multilinear polynomials:
If a compact subset K of U is considered that is an hyper rectangle, the minimum of p restricted to K is **in one of the vertices** of K.

In figure 4.1 there is an cube representing variables bound constraints. If only those bounds were considered (no constant sum), it would have sufficed to search for the minimum on the cube's vertices.
Even if searching on vertices is not sufficient for more complex constraints, it is possible to calculate **exactly** where the minimum of p lies if the set K is determined by a set of **linear constraints**.

A **Multivariate polynomial of degree 2** is a polynomial of many variables in which at most 2 variables are multiplied together:

$$p(\boldsymbol{y}_0, \boldsymbol{y}_1, \boldsymbol{y}_2) = c_0 + c_1\boldsymbol{y}_0 + c_2\boldsymbol{y}_1 + c_3\boldsymbol{y}_2 + c_4\boldsymbol{y}_0^2 + c_5\boldsymbol{y}_0\boldsymbol{y}_1 + c_6\boldsymbol{y}_0\boldsymbol{y}_2 + c_7\boldsymbol{y}_1^2 + c_8\boldsymbol{y}_1\boldsymbol{y}_2 + c_9\boldsymbol{y}_2^2$$

These polynomials have the property that every partial derivative is a **linear function**. This fact makes the search of their minimum simpler, so solving *problem*(b) is possible with exact methods.

Minimum of constrained polynomials of degree 2

If polynomials of degree 2 are used as dictionary functions, to find the critical states and constrain *problem(a)* linearly, *problem(b)* can be solved by using the Quadratic programming algorithm, listed in the appendix. This is the case because the objective of *(*b) is a quadratic polynomial and its constraints are linear.
Because *problem(b)* keeps on changing as the optimization of *problem(a)* goes on, *(*b) should be continuously solved. Solving a quadratic programming problem for every iteration of (a) is very expensive computationally.

Fortunately it is possible to speed up the optimization by exploiting the fact that every time that (b) is solved there is only a slight change in the coefficients of the polynomial, if (a) is solved iteratively using gradient descent. This suggests that (b) is always solved starting from states that are **close to the optima**.
Though because there can be a different local minima in each flat that bound K, by only slightly changing the coefficients it is possible that the global optimal state of (b) changes the flat in which it lies. However the local minimums cannot change much, so by doing a parallel optimization in each flat, it is possible to find the global optima in only few iterations.

Quadratic programming with updating objective Starting from the definition A.1, the new algorithm for solving (b) starting from good guesses from previous iterations is the same as Quadratic programming, but the linear equations solver found in the min equality function at line 2, should be iterative and accept an initial guess. This initial guess $\boldsymbol{y}_J$ will correspond to the previous local minima in the selected flat J.

The new definition is then:

$$\boldsymbol{y}_{min} = arg \min_{J \in 2^D} p_e(min\_equality(p_e, \boldsymbol{C}'_J, \boldsymbol{d}'_J, \boldsymbol{y}_J) \cap K) \tag{4.7}$$

where p_e is the last update on the polynomial done solving $problem(a)$.

If (a) is constrained using the state 4.7, from an iteration to the next this state can "jump", becoming potentially very different. Not having stable constraints can make (a)'s current solution forget the contribution of previous critical states. To improve stability it is sufficient to introduce a constraint for each local minimum of (b). This means adding redundant constraints, among which there is the global minimum. The local minimums are:

$$\boldsymbol{y}_{J \in 2^D} = p_e(min\_equality(p_e, \boldsymbol{C}'_J, \boldsymbol{d}'_J, \boldsymbol{y}_J) \cap K) \tag{4.8}$$

4.4 Constraints and gradient descent

Supposing that the right constraints have been found, *Problem a* has to be solved respecting them and considering that it is has a non linear formulation. A typical way of solving this kind of problems is by using the **method of Lagrange multipliers**. It is based on gradient descent, and it works by adding the constraints as an additional term to the objective function. During the iterations the constraints term becomes more and more prevalent, so after a good region of the search space have been reached by focusing on the objective, the constraints are enforced.
A *Lagrangian function* is defined as

$$\mathcal{L}(\boldsymbol{C}, \boldsymbol{\lambda}) = L_{obj}(\boldsymbol{C}) + \boldsymbol{\lambda}^T L_{constr}(\boldsymbol{C})$$

where L_{obj} is the objective function (4.1) to be minimized with parameters $\boldsymbol{C}$, L_{constr} is a loss function derived from the previous constraints, such that it outputs one loss component for each constraint:

$$L_{constr}(\boldsymbol{C})_{eq\_0} = -d_{\boldsymbol{C}}(\boldsymbol{y}_{S0})_S - d_{\boldsymbol{C}}(\boldsymbol{y}_{S0})_I - d_{\boldsymbol{C}}(\boldsymbol{y}_{S0})_R \tag{4.9}$$

$$L_{constr}(\boldsymbol{C})_{eq\_1} = d_{\boldsymbol{C}}(\boldsymbol{y}_{S1})_S + d_{\boldsymbol{C}}(\boldsymbol{y}_{S1})_I + d_{\boldsymbol{C}}(\boldsymbol{y}_{S1})_R$$

$$L_{constr}(\boldsymbol{C})_{ineq\_i} = -d_{\boldsymbol{C}}(\boldsymbol{y}_{Pi})_i$$

$\boldsymbol{\lambda}$ is a new vector of parameters, one for each constraint, that will be optimized alongside $\boldsymbol{C}$. The inner product of $\boldsymbol{\lambda}$ and L_{constr} is the second term of the Lagrangian function. The original constrained problem

$$\begin{aligned} &min_{\boldsymbol{C}} L_{obj}(\boldsymbol{C}) \\ &s.t. \\ &L_{constr}(\boldsymbol{C})_{eq} = \boldsymbol{0} \\ &L_{constr}(\boldsymbol{C})_{ineq} \preceq 0 \end{aligned}$$

becomes

$$\begin{aligned} & min_{\boldsymbol{C}} max_{\boldsymbol{\lambda}} \mathcal{L}(\boldsymbol{C}, \boldsymbol{\lambda}) \\ & s.t. \\ & \boldsymbol{\lambda}_{\boldsymbol{ineq}} \succeq 0 \end{aligned} \tag{4.10}$$

where $\succeq$ indicates $\geq$ for each element on the vector.
This new formulation has only some positivity constraints on parameters $\boldsymbol{\lambda}$ corresponding to previous inequality constraints. This is to make sure that inequalities do not constrain the search if they are respected. Because the constraints on λ are satisfied by simply cutting the gradient, gradient descent techniques can be now used for solving (4.10) without other considerations.

To optimize this new formulation a library called *Cooper* for Lagrangian-based constrained optimization in Pytorch has been selected. It works by alternating the optimization of $\boldsymbol{C}$ by fixing $\boldsymbol{\lambda}$ and vice versa. In this way the lagrange multipliers are adjusted to force the constraints as L_{obj} gets optimized.
If the optimizer manages to converge to a saddle point, the constraints are guaranteed to be satisfied by the *Karush–Kuhn–Tucker conditions*.

4.5 Thresholding

Sindy or Sparse Identification of Nonlinear Dynamics, as the name implies searches for the coefficients $\boldsymbol{C}$ so that they are sparse. This means that the learned difference function uses only some of the functions present in the dictionary, choosing to leave some out, so with the corresponding coefficient to 0. The procedure used to zero out some coefficients is called *thresholding*.
Leaving some functions out makes the difference function simpler with the goal of making the simulator overfit less, and explain the data in the simplest way possible. This is a form of *regularization*.
To decide which coefficients put to zero the least important ones are eliminated until the validation error starts to degrade. This is done to make the model generalize better.

4.6 Sindy Graph

In order to model the *spatial* information of the epidemics evolution, a graph is used. Each node models a person's infection status at a given time instant, the arcs of the graph model the social relationships between people: if 2 people are connected they may come in contact with a certain probability. This is an instance of **Spatial-temporal Graph** with a graph signal $\boldsymbol{X} \in \mathbb{R}^{T \times |V| \times d}$ where $|V|$ is the number of nodes and d the number of features per node (3) and T the number of time steps, $\boldsymbol{X}_t$ is a shorthand for $\boldsymbol{X}_{t,:,:}$ and is the *snapshot* of the graph at time t.
A susceptible person that comes in contact with an infected individual becomes infected with a certain probability. This probability also changes with time of exposure, so given the same conditions (infection status of the person and that

of neighbours in a given time) two individuals can belong to different departments.
For this reason, in each node there are features $\{S, I, R\} \in [0, 1]$ that evolve in time, indicating the probability that a person falls into a category and $S + I + R = 1$.
Because those probabilities depend on the node's local status at the previous timestep $\boldsymbol{y}$, the learned model takes as input the previous probabilities $\{S, I, R\}$ in addiction to $\{S_n, I_n, R_n\}$, that correspond to the neighbours' status.
To obtain $\{S_n, I_n, R_n\}$ a local aggregation is performed on the graph for each node, with the idea that the more infected people are near an individual, the higher the probability of being infected.
Given the local status $\boldsymbol{y}$, the model then outputs the probabilities of falling into S, I or R at the next timestep.

The model also has **shared parameters** between nodes. This is because 2 different nodes with the same $\boldsymbol{y}$ can infect differently by chance, but are supposed to have the same probabilities of doing so. The model works by probabilities because given the same $\boldsymbol{y}$ the parameters are determined by averaging and are not specific per node.

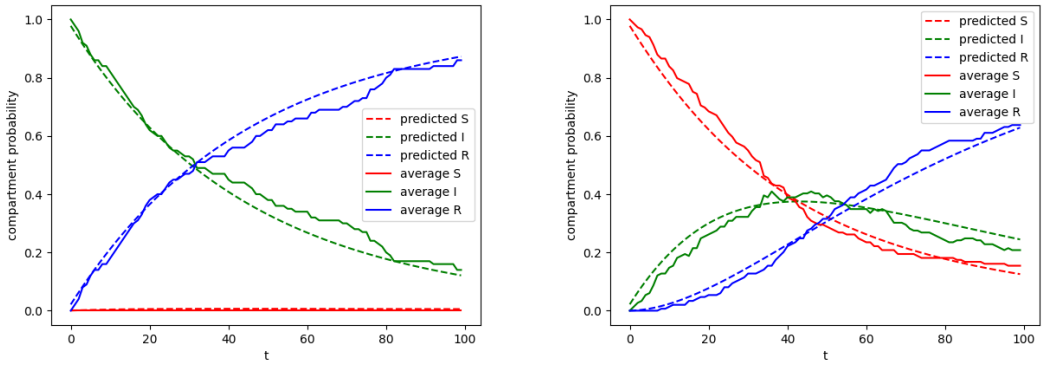


Figure 4.3: Average evolution of nodes with similar local status in 2 different cases. The learned model accounts for different local statuses and uses the average behaviour to predict the probabilities evolutions of similar nodes.

Moreover the fact that the parameters remain the same for each node, and neighbours features are aggregated, creates a **separation** between infection dynamics and graph structure: the model does not depend on the structure, only on local infection state, the structural information is used only in the aggregation phase to diffuse the infection based on how people are socially related. This means that the model can be potentially learned on a graph with certain structural features and work also on a very **different one**.

4.6.1 Definitions

From a message passing point of view, the propagation is done in only one step by simply **summing** neighbours' features, because only **first order neigh-**

bours infect a person. A sum is preferred over the average as a node with more neighbours should be more affected by their state.
The message function simply copies the input:

$$\boldsymbol{M}_{t,v,:} = aggr(\boldsymbol{X}_t, v) = \sum_{w \in N(v)} \boldsymbol{X}_{t,w,:}$$

$$\boldsymbol{H}_t = mess\_pass(\boldsymbol{X}_t) = concat(\boldsymbol{X}_t, \boldsymbol{M}_t, 1)$$

The *concat* function concatenates the 2 matrices along the features axis.
This is repeated for each time step t for each node v to obtain the matrix $\boldsymbol{H} \in \mathbb{R}^{T \times |V| \times 2d}$. Its last dimension contains the 3 node's features plus the 3 new features given by the message passing.

The task is now to learn a simulator that describes a state transition and that does a message passing first before using the leaned difference function. This new simulator is called *sg* or *simulator on graph*:
$\tilde{\boldsymbol{X}}_{t+1} = sg_{\boldsymbol{C}}(\boldsymbol{X}_t)$ if **node level supervision** is used.
$\tilde{\boldsymbol{X}}_{t+1} = sg_{\boldsymbol{C}}(\tilde{\boldsymbol{X}}_t)$ if **aggregated supervision** is used.
If node supervision is available, the train set contains information on how the features of each node evolve in time, otherwise only the mean evolution of the features is available. In the second case the state of each node at a specific time is estimated from the output of *sg* at the previous timestep. *sg* is trained to make sure that the predicted snapshot has the average of node's features corresponding to the supervision.

sg integrates the difference function, that is called after a message passing step is done:

$$sg_{\boldsymbol{C}}(\boldsymbol{X}_t) = \boldsymbol{X}_t + d_{\boldsymbol{C}}(mess\_pass(\boldsymbol{X}_t)) * dt \tag{4.11}$$

The difference function is now computed in parallel for each node:

$$d_{\boldsymbol{C}}(\boldsymbol{H}_t)_{v,i} = \sum_j \boldsymbol{C}_{ji} f_j(\boldsymbol{H}_{t,v,:}) \qquad \forall t, v \in V, i \in \{S, I, R\}$$

With **node supervision** the simulator is trained by solving the problem:

$$min_{\boldsymbol{C}} \qquad \sum_{t,v,i} (d_{\boldsymbol{C}}(\boldsymbol{H}_t)_{v,i} - \boldsymbol{X'}_{t,v,i})^2 \tag{4.12}$$

where $\boldsymbol{X'}_t = \frac{\boldsymbol{X}_{t+1} - \boldsymbol{X}_t}{dt}$ is the discrete approximation of the time derivative for each node and feature of the graph.
If **aggregated supervision** is used, the problem becomes:

$$min_{\boldsymbol{C}} \qquad \sum_{t,i} (mean(sg^t_{\boldsymbol{C}}(\boldsymbol{X}_0)_{:,i}) - mean(\boldsymbol{X}_{t,:,i}))^2 \tag{4.13}$$

where $sg^t(\boldsymbol{X}_0)$ is the t-th step of simulation starting from the initial condition $\boldsymbol{X}_0$; it is actually an integration in time of the learned difference function. The graph simulator is iterated because there is no supervision at node level, so the input needed for the simulator to calculate the next snapshot is simply the output at the previous iteration. This is only an approximation of the real

input, but this is expected to become increasingly better during optimization. Because now the error is calculated after an average, there is little telling how the single node's signal should behave, there is only aggregated information and the **inductive bias**. The dictionary's functions should limit the freedom of choice of the node's behaviour, in order to avoid overfitting.

4.6.2 Constraints

The problems (4.12) and (4.13) are constrained to make sure that the probabilities make sense. Because the simulator $sg_{\boldsymbol{C}}$ has parameters shared, each node of the graph behave the same given the same input state. For this reason the simulator is constrained disregarding nodes distinction, and considering all possible input states.
The idea is to find the most critical state $\boldsymbol{h}$ for a single node, for which the learned difference function maximally breaks a constraint and request that instead the constrain on the critical state is not broken. So *problem(a)* is solved substituting $\boldsymbol{X}$ with augmented state snapshots $\boldsymbol{H}$, and using the method of **Lagrange multipliers** to solve the constrained problem.
In order to find the most critical $\boldsymbol{h}$, it is sufficient to solve *problem(b)* in parallel with *(a)*, and adding some bound constraints for the new features S_n, I_n, R_n. Those feature must be positive and less than an upper bound. Because they come from the sum of neighbours features, and those features cannot be larger than 1, the maximum value they can get is the maximum degree among every node:

$$0 \leq \boldsymbol{h}_i \leq max(deg(v)) \qquad \forall v \in V, i \in \{S_n, I_n, R_n\}$$

4.7 Sindy Graph Implementation

For implementing an optimizer for learning the simulator sg, the library PyTorch is selected. This machine learning library contains many functions useful for doing parallel optimization, the methods are extensible and can be used for **automatic differentiation**. Moreover there are libraries based on PyTorch, such as Torch Geometric, that can be used for doing optimization on graphs.
For implementing Sindy Graph it is sufficient to define the inputs, the simulator's structure, the tensor operations that lead to an output and thanks to the automatic differentiation and with a constrained optimizer, it is possible to find the model's parameters.
The hyper parameters are found using specific techniques. The trained model is then tested on much data, this is possible because the datasets are generated by NDLib.

Every component of the implementation is discussed in the next sub sections.

4.7.1 Difference function

The functions $f_j \in \mathbb{R}^6 \to \mathbb{R}$ that are linearly combined to form the difference function, take as input a state $\boldsymbol{y}$ with a component for each local feature: S, I, R, S_n, I_n, R_n and output a single real value.
They are chosen to represent the possible interactions between features that condition the change in probability of a variable S, I, R with time (4.2).
This is the list of functions introduced in the dictionary divided by type of interaction between features:

Node only	S	I	R	$I * S$	$R * S$	$R * I$	1
Neighbours only	S_n	I_n	R_n	$I_n * S_n$	$R_n * S_n$	$R_n * I_n$	
Mixed	$S * I_n$	$I * S_n$	$R * S_n$	$S * R_n$	$I * R_n$	$R * I_n$	

These functions are **multilinear polynomials** of **degree 2**, this makes the constraining simpler.
Because features are multiplied together they influence each other. For example the rate of change of the probability that a person is infected $\frac{dI}{dt}$ may depend on the neighbours infected I_n. But this sensibility on I_n is there only if the person is susceptible, so the function $S * I_n$ gets a large value only if the person is susceptible and with many infected neighbours, so is expected to get a large learned coefficient.

The difference function contains then 57 parameters $\boldsymbol{C}$, because there is a coefficient for each dictionary function for each of the 3 differential equations to learn. The forward pass of the difference function is simply a tensor multiplication between $\boldsymbol{C}$ and the functions evaluated on the input features. Because using node supervision also the inputs to the model are supervised per node, this is equivalent to doing **teacher forcing**, and this means that the functions evaluations can be precomputed to gain performance.

4.7.2 Message Passing

The message passer is an essential component of the simulator sg. It is required for being able to train sg in the case of aggregated supervision and test sg.
A message passer that aggregates neighbours using the sum is already implemented in Torch Geometric, so a class of this library is extended. The extended class is able to calculate features S_n, I_n, R_n for every node of an input snapshot $\boldsymbol{X}_t$ and also **measure** nodes' features, meaning evaluate the dictionary functions f_j on them.

The **integration on graph** $sg^t_{\boldsymbol{C}}$ is implemented by iterating the simulator starting from an initial snapshot $\boldsymbol{X}_0$. This is a cause of a major bottleneck because by working recursively on a previous iteration's output creates a very long *critical length*, meaning that the code is less parallel. In order to improve performance several integrations are done in parallel, when it is possible to pack many together.

4.7.3 Thresholding

By training $sg_{\boldsymbol{C}}$, the coefficients $\boldsymbol{C}$ each get a value. Sindy is a sparse technique, meaning that it tries to reduce the complexity of the model by selecting only some terms to put on the difference function. In this way the model is more *interpretable* and less prone to overfitting.
To exclude some terms from $d_{\boldsymbol{C}}$, the corresponding coefficients are zeroed out by multiplying $\boldsymbol{C}$ with a mask before the matrix multiplication with the measures.

The terms are removed after an initial training of the model is performed in a phase called **thresholding**. The initial training is done for estimating each term "worth", so how much it contributes to the difference function. Based on the worth a certain **percentage** of less important coefficients are excluded. Then another train phase is performed to adjust the truncated model and then many other thresholding cycles are performed.
Because for every cycle only a few coefficients are excluded, the train procedure becomes an increasingly better *heuristics* for deciding terms' worth.

Because the difference function $d_{\boldsymbol{C}}$ comprises functions of different nature (e.g. I and I_n), simply looking at coefficients to know which terms to threshold can lead to bad results. So a term worth is calculated as:

$$w(i,j) = \boldsymbol{C}_{i,j} \sum_{t,n} |f_j(\boldsymbol{X}_{t,n,i})|$$

this is to take into account the fact that different functions can have different bounds. In fact if a function has a large bound, the model can learn to multiply it by a small coefficient to compensate. If only small coefficients were selected that would exclude such coefficients, even if they are important.

The percentage of terms to be eliminated at each cycle is controlled by an hyper parameter (thr_percent). The elimination of terms is done in relative terms, ordering them by worth and thresholding a percentage of the smallest ones. In this way the hyper parameter is more interpretable and less sensitive

than a fixed threshold.
The elimination is not done in one shot but gradually between epochs, so that the coefficients have the time to readjust after the truncations.
The number of thresholds is decided via early stopping using the validation set. This gradual elimination of coefficients corresponds to a feature selection technique called **Recursive feature elimination with cross-validation** implemented also in scikit learn.

```
n_zero_coeffs = thr_percent*n_coeffs*(thr_index+1)/n_thr
ind_zero = argsort(w(f))[:n_zero_coeffs]
```

Listing 4.1: coefficients selection

4.7.4 Constraints

In order to optimize the problem (4.10) a library called Cooper has been selected. It uses the method of Lagrange multipliers to optimize non linear constrained problems and integrates with the PyTorch workflow. Because it works by gradient descent on top of the PyTorch engine, it can handle a large number of parameters and constraints.
From all the available optimizers, the *Extra gradient SGD* has been selected. It is based on stochastic gradient descent and works in 2 steps:

$$F(\boldsymbol{w}) = [\nabla_{\boldsymbol{C}}\mathcal{L}(\boldsymbol{C},\boldsymbol{\lambda}), -\nabla_{\boldsymbol{\lambda}}\mathcal{L}(\boldsymbol{C},\boldsymbol{\lambda})]$$

$$\boldsymbol{w}_{t+\frac{1}{2}} = \boldsymbol{w}_t - \eta F(\boldsymbol{w}_t)$$

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \eta F(\boldsymbol{w}_{t+\frac{1}{2}})$$

Essentially it updates parameters $\boldsymbol{C}$ and $\boldsymbol{\lambda}$ by doing gradient descent separately on the 2 and the gradient is also extrapolated. The extrapolation is similar to the Runge–Kutta methods, and is useful for having more convergence guarantees.
The parameters are all initialized to zero, in this way the problem is initially unconstrained and this makes it possible to reach more easily a good region before constraining. This is because as $\boldsymbol{\lambda}$ parameters grow, the gradient is less prone to cross unfeasible regions.

The method chosen for the constraining is the **random sampling**. A certain number of samples are taken at random from the feasible region of states, and the constraints are enforced on these states hoping that they are sufficiently close to the critical ones.
Because the constraints are approximated, after the train, the constraint error of the model can be recalculated more precisely by focusing on more specific states, for example those reached by the model in the test set. For doing this a new set of states is considered and the constraints error is calculated on them. Because the model was not trained on this new set, this new error should generalize well:

$$constr\_err(\boldsymbol{C}) = \sum_{\boldsymbol{y}\in NewSetS} |L_{constr}(\boldsymbol{C},\boldsymbol{y})_{eq}| + \sum_{\boldsymbol{y}\in NewSetP} relu(L_{constr}(\boldsymbol{C},\boldsymbol{y})_{ineq}) \tag{4.14}$$

where L_{constr} are the constraint functions of (4.6).

Another method for constraining that could have been used is **Quadratic programming with updating objective**. It has the disadvantage that it is specifically designed for dictionary functions that are polynomials of degree 2. The advantages though is that it is exact, so no constraints have to be approximated and it is reasonably fast.

4.7.5 Training

The training set consists of a series of snapshots $\boldsymbol{X}_t$ that are subdivided into consecutive time intervals to form the training set, the validation set and the test set. The validation set is 20% of the training set and because the dataset is generated and the test set can be as long as wanted, it is the same length as the training set.

The train cycle consists of a certain number of epochs in which the whole training set is used. The number of epochs is decided by **early stopping** by using the validation error and the constraint error.
The early stopping works by considering the best epoch so far and stopping the train cycle if there is no improvement after a certain number of epochs (the tolerance). The best epoch is decided with:

```
1 def is_best_epoch(constr_err, val_err, best_constr_err,
      best_val_err):
2     return (constr_err <= 1e-4 and val_err < best_val_err) or (
      constr_err > 1e-4 and constr_err < best_constr_err)
```

Listing 4.2: function for selecting the best model so far

The constraint error is minimized until it is below a threshold, then the validation error is used.
The validation error is calculated by integrating the current learned model from the first time step until the end of the validation set, then the MSE is calculated only on the validation interval.

$$val\_err(\boldsymbol{C}) = MSE(sg_{\boldsymbol{C}}^{val\_t}(\boldsymbol{X}_0)_{train\_t:val\_t,:,:}, \boldsymbol{X}_{train\_t:val\_t,:,:})$$

4.7.6 Node supervision

When node supervision is used, all input and output snapshots $\boldsymbol{X}_t$ are available. This means that the model can be trained using teacher forcing (4.12), for this reason the train loss can be computed in parallel for each timestep and the train goes fast.
During the train, multiple thresholding cycles are performed, until there is no improvement on the validation error for a certain number of cycles.

4.7.7 Aggregated supervision

With supervision at node level, for every time step the expected output $\boldsymbol{X}_{t+1}$ is available for each node. Because of this also every input $\boldsymbol{X}_t$ is available for each node, by simply considering the previous time step.

With aggregated supervision the graph's signal is unknown, and the only available supervision is given by the **aggregation** of the signal with respect to all the nodes (4.13):

$$m(t, i) = mean(\boldsymbol{X}_{t,:,i})$$

Because no input is available at node level for a given time step there is no way of calculating the output of the model and then the loss. For this reason the graph's signal has to be approximated someway.
Two strategies have been formulated to approximate the inputs:

1. Trying to recover node level information from m by spreading it equally:

$$\tilde{\boldsymbol{X}}_{t,:,i} = m(t, i)$$

2. Estimate input from the output given by the model at current epoch e:

$$\tilde{\boldsymbol{X}}_t = sg_{\boldsymbol{C}_e}(\tilde{\boldsymbol{X}}_{t-1})$$

The problem with the first technique is that it does not model how the infection evolves with clusters of infected individuals. All individuals have all the same probability of being infected regardless of the number of neighbours and their infection probability, so the model does not learn how to handle the spatial spread of the virus.
The problem with the second technique is that it heavily relies on the goodness of the prediction at previous time steps to give a precise input. When at $e = 0$ the model gives bad predictions, leading to an increasing error from a time step to the next that makes the training procedure completely diverge.

Because the downsides of 1. and 2. are complementary, the techniques are combined initializing the model with the first technique and slowly transitioning to the second. This can be seen as a **curriculum learning** technique [37]. The idea is to define a window of length l and calculate the outputs by integrating the model for l timesteps with 2., starting from t_0 with an estimation from 1. By using many shifted windows all the outputs can be calculated.

$$\tilde{\boldsymbol{X}}_{t_0+t} = sg^t_{\boldsymbol{C}_e}(m(t_0, :)) \qquad \forall t < l$$

The training is performed in multiple phases, by growing the window's length every time, starting from $l = 1$ that correspond to 1., to $l = train\_points$ that correspond to 2. In this way the model is initialized good enough to avoid divergence, and slowly it learns how to differentiate the probabilities based on the local infection status.
To exploit the dataset as much as possible, the windows are overlapped. Using curriculum learning the objective function (4.13) becomes:

$$min_{\boldsymbol{C}} \sum_{j,t<l,i} (mean(\tilde{\boldsymbol{X}}_{jo+t,:,i}) - m(jo + t, i))^2 \tag{4.15}$$

where j is the index of a window and o is the offset between windows.

After the curriculum learning a thresholding phase is performed composed

of many cycles. The training in between cycles is done using (4.15) with $l = train\_points$.
As the windows grow, the performance of the optimizer degrade, this happens because the integrations become longer and longer, and they are not very parallel operations.
As the curriculum learning unfolds, neighbours' features should be more and more exploited to follow the aggregated supervision. A better model means also better estimated inputs to learn an even better model.

4.7.8 Hyper parameters

There are multiple hyper parameters to be selected: the primal and dual learning rates, the tolerance of epochs, the tolerance of thresholds, then increment on the size of the windows for the aggregated supervision, the percentage of coefficients to remove at each thresholding cycle.
The primal and dual learning rates are the only hyper parameters searched for during the testing, using grid search. The others are fixed, in order to improve performance.

Chapter 5

Experimental Results

The goal of this chapter is to discuss how our method, called **Sindy graph with aggregated supervision and constraints**, performs in the task of epidemiological forecasting and compare it to an instance of existing techniques called Spatial-Temporal Graph Neural Networks.
We notice that our technique works well for forecasting and outperforms STGNNs.

We include many more experiments to evaluate how Sindy graph's performance are increased by enforcing constraints, how it would perform if node supervision was available, and how sensible is the technique to the length of the training set.
To show that Sindy is a good starting point on which adding our contributions, we compare it to STGNNs improved with same constraints and aggregated supervision found in our technique.

5.1 Tests introduction

For testing, all the experiments are done in the same way for each method considered, in order to be able to compare them.
In this section the methodology, datasets, statistics and plots used for testing are described.

All the methods tested require information about the structure of the **graph** that represents the relations between people in the population. All the experiments are run on the same graph, that has 1000 nodes and is generated randomly using the Erdős–Rényi model. For this type of graph each edge has a fixed probability of being present or absent, so no considerations on how people might cluster together are included, this is less realistic but more simple to study.
Each person has on average 5 neighbours, this means that the virus has to pass from many people before infecting everyone. This makes the spread dynamics more interesting and similar to what happens when people are isolated.

The **datasets** are generated with NDlib, that is capable of simulating the evolution of a viral disease on a known graph. NDlib makes it possible to define

some infection parameters that in our case have values $\beta = 0.01$ and $\gamma = 0.02$ and a seed, that is used to initialize the random number generators.

To test the different methods, **20 runs** are done for each, using 20 datasets generated by NDlib choosing each time a different seed. All the results are then averaged across the runs, to have more meaningful statistics.
In case some tests **fail** because some of their statistics have a nan value, those are excluded from the average calculation, but their count is reported.

For each run a **grid search** is performed to find the best set of hyperparameters. The best model is selected by looking at the validation error. Usually a small set of hyperparameters is used, to accelerate the tests, in fact from a run to another the best set of hyperparameters does not change significantly.
Because focus has been given to get the best performing models, the train time is quite long. It is possible to greatly reduce it by doing little compromises with performance, for example by reducing the various tolerances or avoiding spending much time for slow computations such as the sequential integration in Sindy with aggregated supervision.
The hyper parameters considered in the grid search are reported for each method. For each method the size of the train, validation and test set in terms of number of timesteps is given. Then the learning rates are reported as well as the tolerance used for epochs, this describes how much the early stopping waits for improvement before finishing the training.

A model's performance is evaluated by calculating different **statistics**:

- **mean time:** is the average across runs of the time taken to train and evaluate the model while doing grid search
- **mean train error:** it is the value of the loss corresponding to the best model selected. It is specific for Sindy graph and the STGNNs so they cannot be compared. It is averaged across runs
- **mean test errors:** there are 4 types of **test errors** presented. The test MSE is an error calculated between the true dynamics and the learned one that gives more weight to instances with large errors. The MAPE error is an absolute percentage error. This means that it is more interpretable and relative to the true value, meaning that behaviours close to 0 have more weight.
  The mean test forecast % indicates for how many timesteps into the test set the MAPE error is below the given percentage. This is useful to understand what are the forecast capabilities of the model.
  All these errors are calculated on the test set after an aggregation, that is an average of every node's dynamics. This aggregation can be interpreted as the expected number of people on a compartment at a given time.
- **mean constraint errors:** for the tests, these are calculated differently than the constraint losses used for training. The errors are calculated by considering only the states reached in the test set by using 4.14. This gives a measure of how much the model's predictions diverge as time grows because of the non satisfaction of the constraints.
- **specific statistics:** some methods have more statistics to

Different types of **plots** are presented along with the results:

- **Node's dynamics plot** represent how the learned probability of being in a compartment changes with time as the infection spreads for a single sample node. Along with the probability it shows the real changes of compartment given by the simulation.

- **Aggregated plot** shows the expected number of people in a compartment as a function of time, with 0 meaning no one and 1 meaning everyone. This functions are found by averaging the probabilities of each node for every timestep. It is split into train, validation and test time windows.

- **Constraints evolution plot** shows how the approximated constraint loss evolves as the number of epochs increases. The various phases are delimited along with the target constraint loss.

- **Validation error plot** shows how the mean squared error in the validation set evolves with the number of epochs.

5.2 Sindy Graph

Sindy is the technique used as the basis for our method. We added the possibility of using it on graphs, added constraints and the possibility of training it using aggregated supervision.

The way we extended Sindy to graphs is explained in more detail in Sindy Graph. In short Sindy identifies dynamics from data by selecting some functions from a dictionary, and combining them linearly to discover the differential equations that govern the physical phenomena under study. With graph snapshots as inputs, the relationships between nodes, that form the graph's structure, should be taken into consideration. This information is encoded as spatial features extracted from the graph and is added to Sindy by combining those features into new dictionary functions.

For selecting the right functions, Sindy uses a procedure called Thresholding. This is not a simple task as for performing the right selection a good model should be learned first, but for learning a good model, only a good subset of functions should be considered. The exclusion of terms from differential equations is performed iteratively and controlled by the validation error.
A new hyper parameter is added for Sindy called **tolerance thresholds**, that indicates how many iterations of the threshold to perform before stopping and selecting the best model found. Another hyper parameter **coeffs increment** tells the percentage of the total coefficients that are 56 is removed at each threshold iteration.
Also a new statistic is added, called **mean nonzero coefficients** that gives an idea of how many terms are present in the differential equations.

Aggregated supervision is the only one available in practice from real world

datasets. In fact knowing the dynamics of infection of every single person in a population is nearly impossible. Only estimates and probabilities regarding the number of infected individuals at a given time can be known. For this reason Sindy Graph is trained using only aggregated supervision and approximate teacher forcing. The approximations are improved as epochs increase.
Because the datasets come from simulations, also node supervision is available, and is useful to see how Sindy would perform if perfect information was available. So also this kind of tests is included.

For the experiments using **constraints**, the Random sampling technique that we introduced is used, to find the constraint problem formulation adapted to graph 4.6.2. The implementation is discussed in 4.7.4. This technique is chosen for its simplicity and stability.
Constraints prove to be an important tool for improving the performance of the models, as will be discussed.

The models are tested using shorter and longer training sets to understand how performant is Sindy Graph with a lack of large quantities of data.

Because Sindy models are interpretable, the learned differential equations can be written in symbolic form.
Each of the 20 tests lead to different **differential equations** because the training set is very small so little variations in the simulation can lead to changes in the coefficients and in the selected terms. Only an average of the terms present in the differential equations is presented. To do this the coefficients are averaged across runs, and the mean absolute deviation (MAD) of each coefficient is calculated, then only the top 5 most important terms are kept. A term importance is given by how few times it has been thresholded across the tests.

5.2.1 Sindy graph with node supervision 100

In this model, Sindy is adapted to work on graphs, without constraints or aggregated supervision. It functions as a baseline for the other methods based on Sindy to then understand how our other contributions affect the performance.

Dataset 1	
constraints	no
supervision	node
train,validation,test size	[100,125,225]
learning rates	[0.002,0.022]
coeffs increment	0.05
tolerance thresholds	3
tolerance epochs	2000
mean time	1025s
mean constr error	$856 * 10^{-4}$
mean train error	25.33
test MSE	$27.68 * 10^{-4}$
test MAPE	40.96%
mean test forecast 10%	5.0
mean test forecast 20%	9.63
mean nonzero coefficients	50
failed tests	0

The **mean constr error** is an average error over all types of constraint, meaning that with a single value it is possible to get a sense of how bad the constraint violations are.
It can be noticed how the mean constraints error is close to 0.1. This means that on average a node's dynamics on the test set, violates the constraints by nearly 0.1. For example, instead of having constant sum, the sum varies by 0.1 on average and a variable becomes < 0 or > 1, violating the bounds constraints by an average value of 0.1.
This is a noticeable violation and indicates that without constraints enforced, the learned dynamics are **not physically plausible**.
By looking at the test MSE and MAPE, the errors are reasonable, and observing the mean test forecasts, it can be noticed how the models found are even able to make forecasts on the future dynamics for a few time steps within a reasonable MAPE (10% and 20%).
The fact that the constraints are violated though, indicates that as the number of time steps increases so do the test errors, probably in an exponential way. In fact because of the amplifying nature of differential equations it is easy to learn solutions that explode exponentially. This invalidates all possible forecasts after a certain time limit.

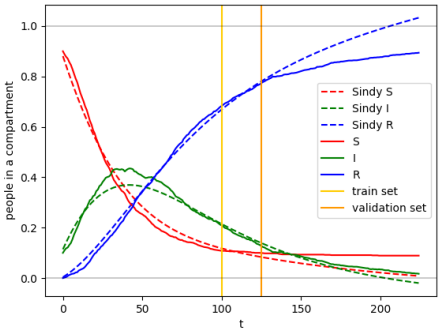
$\frac{dS}{dt}$	$-0.44(\pm0.06)S + 0.41(\pm0.05)I + 0.39(\pm0.05)R - 0.38(\pm0.06) - 0.21(\pm0.15)S_n$
$\frac{dI}{dt}$	$+0.63(\pm0.21)S - 0.64(\pm0.21)I + 0.02(\pm0.03)R + 0.07(\pm0.04) + 0.21(\pm0.13)S_n$
$\frac{dR}{dt}$	$-0.38(\pm0.03)S + 0.41(\pm0.09)I - 0.38(\pm0.03)R + 0.35(\pm0.06) + 0.08(\pm0.07)S_n$

Because Sindy is an interpretable method, the learned differential equations can be reasoned upon. By looking at $\frac{dI}{dt}$, the change of probability of a node's infection, should depend on the probability that the neighbours are infected. In particular the neighbours contribution should depend also on how probably the node is susceptible.
An important term is then $S * I_n$ for the derivative of I, that makes the infection probability of a node rise if it has high probability of being susceptible and it has many infected neighbours.
Now, the most important terms here don't include $S * I_n$, this happens probably because without constraints the model has less physical information so it overfits the training set more easily.
Another noticeable thing is that the most important terms in the derivatives are not neighbours features. This means that the learned models rely on an average behaviour that is only slightly corrected by neighbours contributions.

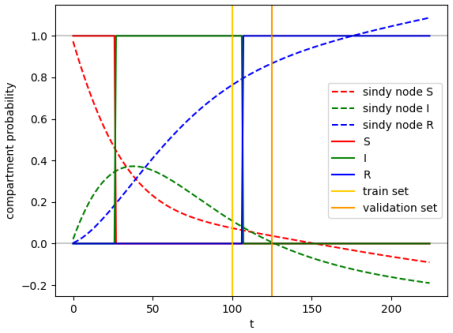
Thresholding in Sindy Graph, as it will be seen also in the other tests, tends to keep many coefficients and spread the learned dynamics among them. This conservative approach seems to reflect the complexity of selection. A more throughout study of thresholding will certainly be helpful for extending the method in future.

By looking at the aggregated 5.1a and node 5.1b plots it is clear how the model diverges quickly as time goes on. Also it can be seen that the node's constraints are much violated.

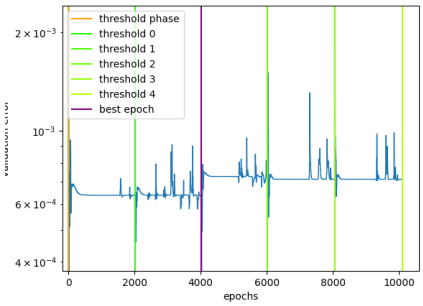
By looking at the figure 5.1c, it can be seen how the validation error changes as the number of epochs increases, the epochs in which a thresholding is done are shown. The error remains relatively stable after the first threshold, then it starts to increase, indicating that with less terms in the differential equations, the model generalizes worse.



(a) Aggregated plot



(b) Node's dynamics



(c) validation error

Figure 5.1

5.2.2 Sindy graph with node supervision and constraints 100

This method adds constraints to the previous Sindy graph with node supervision. Because Random sampling is used to find the constrained problem formulation, the constraints definition is approximated. Moreover the optimizer used to solve the constrained problem is different, meaning that the behaviour of the errors during the training is expected to be different. Now the learning rates are 2: one used to solve the primal problem and one for setting the lambdas.

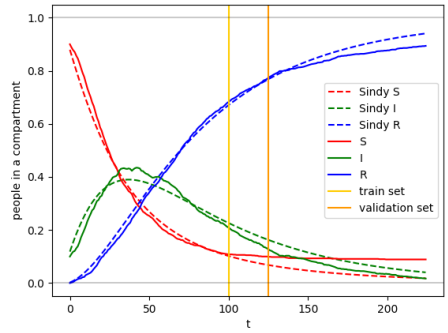
Dataset 1	
constraints	yes
supervision	node
train,validation,test size	[100,125,225]
learning rates (primal,dual)	[0.002,0.022],[0.2,2.2]
coeffs increment	0.05
tolerance thresholds	3
tolerance epochs	2000
mean time	2469s
mean constr error	$3.99 * 10^{-4}$
mean train error	25.298
mean test MSE	$12.83 * 10^{-4}$
mean test MAPE	37.02%
mean test forecast 10%	0.0
mean test forecast 20%	0.0
mean nonzero coefficients	41
failed tests	0

It can be observed that adding **constraints**, the constraint loss target is easily met 5.2c and the test constraint error is very low, meaning that the solutions to the differential equations should remain bounded and with constant sum even after many timesteps. This happens even though the constraints definition is approximated. This is probably due to the problem 4.6 being relatively simple, without many local minimums and with a smooth enough behaviour to be correctly sampled even with few points.
The MSE and MAPE are good too, the mean test forecast is 0 though, indicating that the learned solution parts too much from the true behaviour to make reliable forecasts.

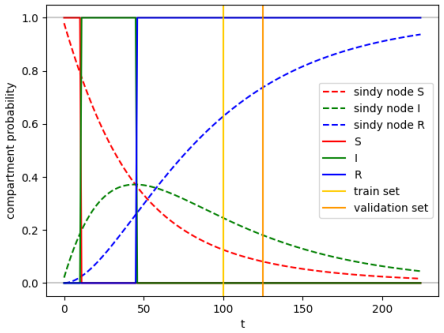
$\frac{dS}{dt}$	$-0.75(\pm0.06)S + 0.25(\pm0.02)I + 0.25(\pm0.02)R - 0.25(\pm0.02) - 0.01(\pm0.09)S_n$
$\frac{dI}{dt}$	$+1.00(\pm0.06)S - 1.00(\pm0.05)I - 0.00(\pm0.02)R + 0.00(\pm0.02) + 0.65(\pm0.18)S * I_n$
$\frac{dR}{dt}$	$-0.25(\pm0.02)S + 0.75(\pm0.05)I - 0.25(\pm0.02)R + 0.25(\pm0.02) + 0.03(\pm0.10)I * S_n$

Also in this case the most important features in the differential equations are S, I, R that are then corrected with neighbours terms. The other important term is $S * I_n$ for the derivative of I, it also has low variation meaning that across the tests, models agree on its presence. This is a good sign indicating that with constraints the model is more physically correct, and its interpretation can lead to insight into the phenomenon.

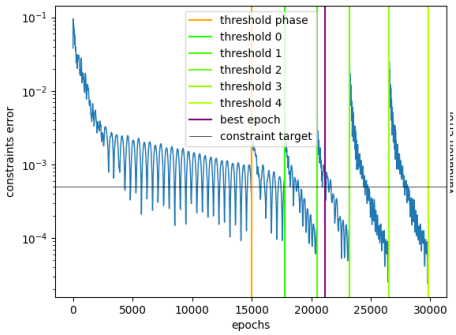
The aggregated plot 5.2a shows good adherence of the learned dynamics to the true one.
The node plot 5.2b shows that because the constraints error is very low, the dynamics remain well bounded for every node. In fact the 3 functions flex near the bounds 0 and 1 indicating that they slow down as they approach the bounds.



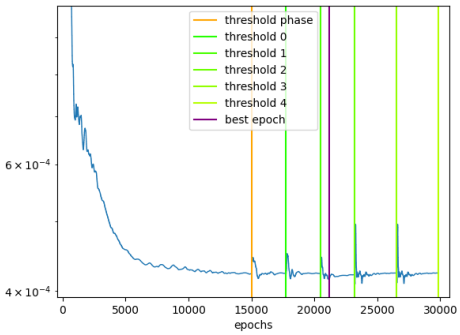
(a) Aggregated plot



(b) Node's dynamics



(c) Constraints evolution



(d) Validation error

Figure 5.2

The constraints evolution plot 5.2c displays many interesting details regarding the way the training behaves. A noticeable thing is the oscillatory nature of constrained optimization, in fact because the Lagrange multipliers are optimized with the model's parameters (4.10) the objective tends to alternate the focus between the constraints and the parameters of the model. This behaviour is useful to improve the search in the hypothesis space, by exiting local minima more easily.
Looking at the constraints and validation plots 5.2d is clear how after any thresholding phase the model needs some epochs to return to the error it had before and improve it.
Moreover it can be noticed how the constraints error and validation error struggle to decrease at the same time. In fact the best validation error that respects the constraints goal is close to the target threshold. The constrained optimizer is working hard to find a good compromise as with constraints the search freedom is restricted, but at the same time the constraints lead the search towards good regions.
As epochs increase, the validation error tends to increase as well because of overfitting. Instead the constraints error keeps on decreasing. In fact the constrained optimizer should reach error 0 if enough epochs are done.

5.2.3 Sindy graph with aggregated supervision 100

This method implements aggregated supervision meaning that it receives much less information from the simulation, but is more realistic and better reflects the results that can be obtained using real world datasets.
For dealing with aggregated supervision a technique has been proposed 4.7.7, that works by generating better and better approximations of node supervision as epochs increase. Using this technique the model should learn by itself how the virus spreads spatially, by only considering the graph's structure and how many people is in a given compartment at a given time.
Because no teacher forcing is available, the model is trained by running it recurrently for 100 steps before calculating the loss. This causes instability in the training process, for this reason it is split into more phases in which the recurrent steps grow, slowly.

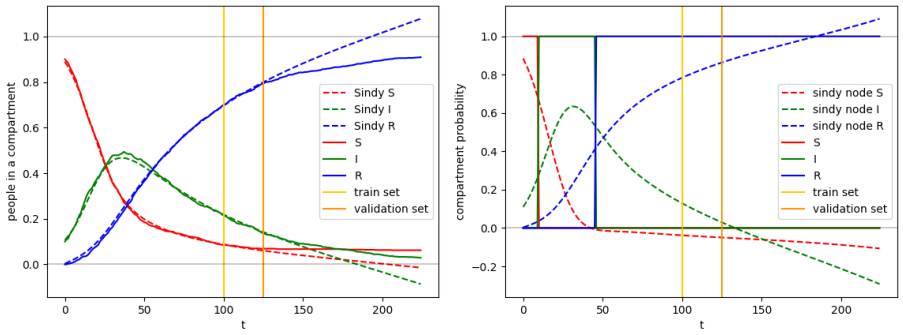
Dataset 1	
constraints	no
supervision	aggregated
train,validation,test size	[100,125,225]
learning rates	[0.005,0.055]
coeffs increment	0.05
tolerance thresholds	3
tolerance epochs	3000
bootstrap increment	25
mean time	10352s
mean constr error	$1064 * 10^{-4}$
mean train error	0.017
mean test MSE	$15.19 * 10^{-4}$
mean test MAPE	30.27%
mean test forecast 10%	1.63
mean test forecast 20%	9.05
mean non zero coefficients	50
failed tests	8

First of all, in this case 40% of the tests failed because the solutions to the found difference function increase exponentially so the test errors become nan. This happens because the constraints are not enforced. For the tests that did not fail, the test error is initially not bad but then diverges.
For the valid tests the mean constraint error is similar to that of Sindy graph with node supervision, and the test errors are not bad, indicating that the model is able to forecast for some time steps into the test set, before exploding.

$\frac{dS}{dt}$	$+0.90(\pm0.69)I + 0.30(\pm0.33)R - 0.05(\pm0.13) + 0.70(\pm0.55)I_n - 0.10(\pm0.21)R_n$
$\frac{dI}{dt}$	$+0.59(\pm0.51)S - 1.74(\pm1.02)I - 0.48(\pm0.43)R + 0.41(\pm0.40) - 0.50(\pm0.62)S_n$
$\frac{dR}{dt}$	$+0.67(\pm0.44)I + 0.06(\pm0.22)R + 0.17(\pm0.22) + 0.40(\pm0.37)I_n - 0.07(\pm0.15)R_n$

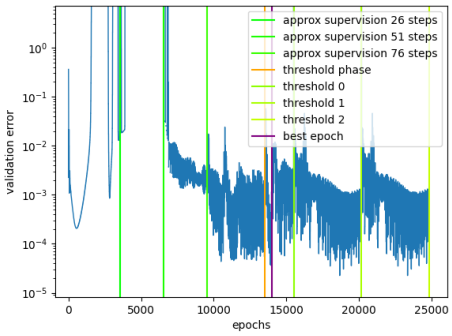
Also in this unconstrained case the most important coefficients don't reflect intuition, and the mean deviation is high, indicating that the models from different test runs don't agree on the terms to use.

Looking at 5.3b it is clear how constraints are not respected and because



(a) Aggregated plot

(b) Node's dynamics



(c) validation error

Figure 5.3

of this the dynamics in the test set tend to be straight lines going past the bounds instead of curving.
The training phase without constraints is much more chaotic and unstable 5.3c as the method is used recurrently with larger and larger windows. To compensate for this, the learning rate is kept much lower, potentially increasing training times.
Even if unstable, the validation error decreases as approximated teacher forcing is used less and less.

5.2.4 Sindy graph with aggregated supervision 150

This method is the same as 5.2.3, but it uses a larger training set, while the validation and test sets are shifted forward in time. This experiment is carried out to evaluate how the performance of Sindy graph change as available data

increase. This is done to get a sense of how soon reliable forecasts can be had, after the pandemic strikes.

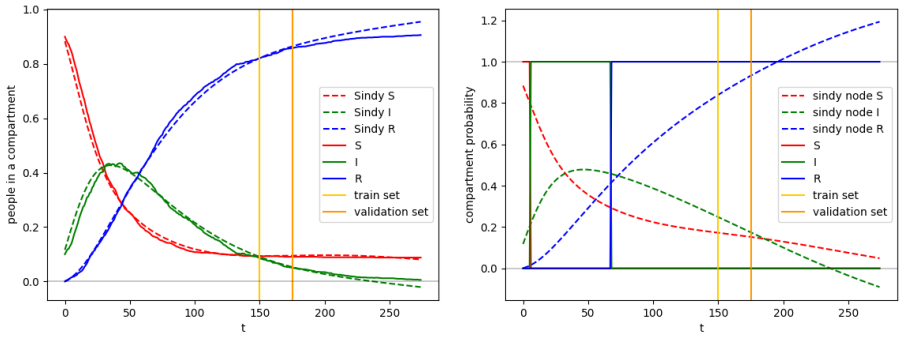
Dataset 1	
constraints	no
supervision	aggregated
train,validation,test size	[150,175,275]
learning rates	[0.005,0.055]
coeffs increment	0.05
tolerance thresholds	3
tolerance epochs	3000
bootstrap increment	25
mean time	20106s
mean constr error	1266 * 10^{-4}
mean train error	0.019
mean test MSE	5.61 * 10^{-4}
mean test MAPE	31.93%
mean test forecast 10%	14.78
mean test forecast 20%	35.31
mean non zero coefficients	52
failed tests	0

In the unconstrained case of sindy aggr, growing the training set does improve performance. In fact there are now 0 tests failed and the test errors are better. Moreover forecast errors indicate that the model is now able to better extrapolate future dynamics.

$\frac{dS}{dt}$	$-0.81(\pm0.33)S + 0.93(\pm0.72)I + 0.18(\pm0.36)R + 0.04(\pm0.18) + 0.48(\pm0.34)S_n$
$\frac{dI}{dt}$	$+0.84(\pm0.46)S - 2.02(\pm0.75)I - 0.63(\pm0.33)R + 0.56(\pm0.33) - 1.01(\pm0.37)S_n$
$\frac{dR}{dt}$	$-0.51(\pm0.37)S + 0.86(\pm0.48)I - 0.15(\pm0.24)R + 0.30(\pm0.24) + 0.14(\pm0.41)S_n$

Differential equations' terms still don't reflect intuition and still have great variability.

The constraints error is still high, in fact by observing 5.4b, it is clear how the dynamics is still not aware of the bounds and simply passes through them, albeit at a later time than 5.2.3.



(a) Aggregated plot (b) Node's dynamics

Figure 5.4

5.2.5 Sindy graph with aggregated supervision and constraints 100

This is the method with all our contributions. It is realistic as only aggregated supervision is used, with only 100 time steps in the training set. Even if aggregated supervision is usually unstable as seen in 5.3c, the usage of constraints here has a positive effect, not only for lowering generalization error, but also for stabilizing the train process.

Dataset 1	
constraints	yes
supervision	aggregated
train,validation,test size	[100,125,225]
learning rates (primal,dual)	[0.1],[10]
coeffs increment	0.05
tolerance thresholds	3
tolerance epochs	3000
bootstrap increment	25
mean time	9089s
mean constr error	$2.68 * 10^{-4}$
mean train error	0.027
mean test MSE	$6.27 * 10^{-4}$
mean test MAPE	24.86%
mean test forecast 10%	12.68
mean test forecast 20%	29.78
mean non zero coefficients	46
failed tests	0

By using aggregated supervision the errors are all lower than Sindy graph with node supervision and constraints 100 and the forecast capabilities are higher. This probably happens because by aggregating, the model is less sensible to **noise**. In fact Sindy Graph with node supervision is supervised by first calcu-

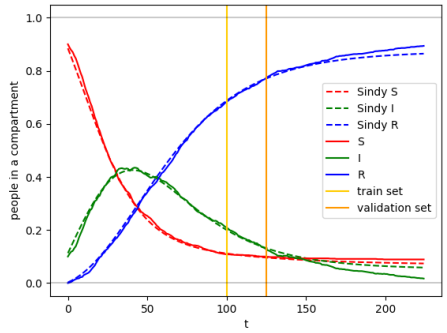
lating a discrete derivative of the dynamics at node level, that is by definition very noisy as it follows a probability distribution. The derivative operation amplifies the noise even more, making Sindy struggle. With aggregated supervision the noise is reduced as it is the mean of all nodes' dynamics.
The generalization errors are also better than Sindy graph with aggregated supervision 100, thanks to constraints, whose error is now much lower. The model forecasts better because with constraints it is much less prone to later explosions of the dynamics.

$\frac{dS}{dt}$	$-0.62(\pm0.11)S + 0.21(\pm0.03)I + 0.22(\pm0.04)R - 0.20(\pm0.04) + 0.25(\pm0.07)S_n$
$\frac{dI}{dt}$	$+0.76(\pm0.13)S - 0.77(\pm0.07)I - 0.23(\pm0.07)S_n + 1.39(\pm0.34)S * I_n + 0.38(\pm0.10)I * S_n$
$\frac{dR}{dt}$	$-0.14(\pm0.04)S + 0.56(\pm0.06)I + -0.23(\pm0.03)R + 0.18(\pm0.02) - 0.06(\pm0.03)R_n * I_n$

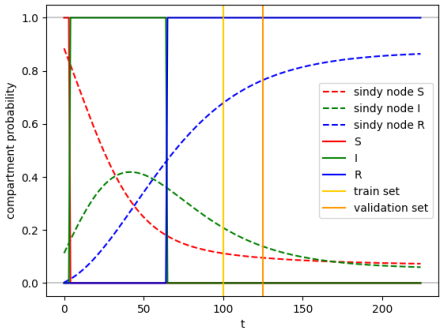
Here the term $S * I_n$ has the largest coefficient indicating that despite not knowing how the simulation behaves at node level, the model correctly captures how neighbours contribute to a node's infection probability.
Also the term $I * S_n$ is large, indicating that if a node is likely infected and many neighbours are susceptible, the infection probability grows. This is probably because when the spread is at the beginning the probability of infection is low, so many susceptible neighbours does not mean more probability of a future spread, the situation changes when the infection is spread more.
This kind of insight is only possible because the constraints make the models physically plausible.

Figure 5.5b shows how the dynamics curve so they will never go past the bounds. Moreover the sum remains constant throughout all time steps. In figure 5.5a it can be appreciated how this method is able to exploit the physics information given, to follow the true behaviour many steps into the validation and test sets.

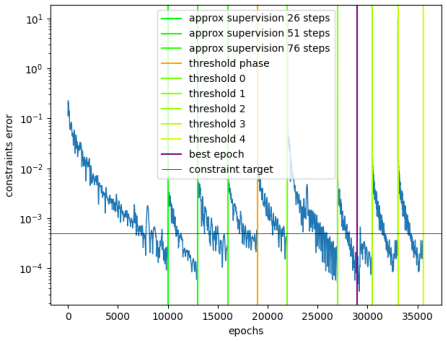
Here the validation error 5.5d tends to rise a bit as the supervision window grows, to then drop as the thresholding eliminates unnecessary coefficients that worsen the generalization capabilities of the model.
Also the validation error drops in the first 1000 epochs to then rise, this is due to the fact that the Lagrange multipliers rise slow, so the problem is unconstrained at the start. This is useful to reach a good region of the hypothesis space fast, without being pushed away by the constraints.



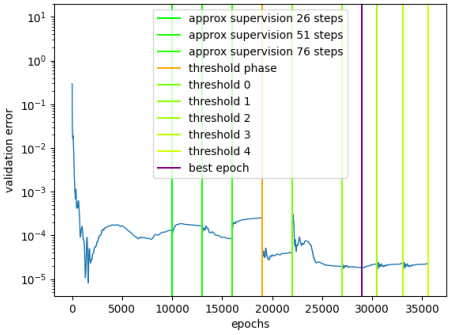
(a) Aggregated plot



(b) Node's dynamics



(c) Constraints evolution



(d) validation error

Figure 5.5

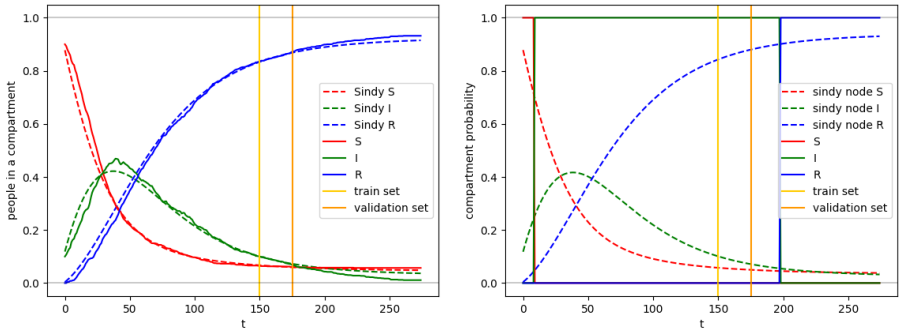
5.2.6 Sindy graph with aggregated supervision and constraints 150

Dataset 1	
constraints	yes
supervision	aggregated
train,validation,test size	[150,175,275]
learning rates (primal,dual)	[0.005],[0.5]
coeffs increment	0.05
tolerance thresholds	3
tolerance epochs	3000
bootstrap increment	25
mean time	20487s
mean constr error	$18.13 * 10^{-4}$
mean train error	0.028
mean test MSE	$2.33 * 10^{-4}$
mean test MAPE	46.13%
mean test forecast 10%	0.6
mean test forecast 20%	9.4
mean non zero coefficients	57
failed tests	0

Incrementing the size of the training set to 150 does not lead to an improvement of the performance. The MSE is lower, but the other statistics are worse than sindy aggr 100. This is probably because the most interesting part of the dynamics is in the first 100 timesteps and growing the training set does not make much difference. Moreover the constraints help keeping the dynamics close to the true values.

$\frac{dS}{dt}$	$-0.73(\pm0.03)S + 0.26(\pm0.01)I + 0.27(\pm0.01)R - 0.42(\pm0.07)I * S - 0.14(\pm0.03)R * S$
$\frac{dI}{dt}$	$+0.77(\pm0.05)S - 0.66(\pm0.03)I - 0.04(\pm0.01)R + 0.32(\pm0.07)I * S - 0.05(\pm0.02)R * S$
$\frac{dR}{dt}$	$-0.04(\pm0.04)S + 0.40(\pm0.03)I - 0.23(\pm0.00)R + 0.10(\pm0.01)I * S + 0.19(\pm0.01)R * S$

In this case the terms corresponding to neighbours features are not included in the 5 most important terms, but they have large coefficients nonetheless.



(a) Aggregated plot (b) Node's dynamics

5.3 Spatial-Temporal Graph Neural Networks

Sindy graph is a symbolic technique that has been adapted to learn dynamics on graphs. This technique is compared to a more traditional approach, a type of Spatial-Temporal Graph Neural Network, to understand how they compare.

Spatial-Temporal Graph Neural Networks 1.5.4 usually work by extracting first spatial feature using a GNN, then the evolution in time of those features is captured by a RNN. In our case the RNN is kept simple, because no long term memory is needed. In fact the learned model should use only the previous state to forecast the current one.

Given graph signal $\boldsymbol{X} \in \mathbb{R}^{T \times |V| \times d}$ where $|V|$ is the number of nodes and d the number of features per node (3) and T the number of time steps, the task is to learn a **simulator** $sg_{\boldsymbol{C}}$ that advances the signal by one time step in the future.
$\tilde{\boldsymbol{X}}_{t+1} = sg_{\boldsymbol{C}}(\boldsymbol{X}_t)$ if **node level supervision** is used.
$\tilde{\boldsymbol{X}}_{t+1} = sg_{\boldsymbol{C}}(\tilde{\boldsymbol{X}}_t)$ if **aggregated supervision** is used.

Where $\tilde{\boldsymbol{X}}_t$ is the estimation given by sg of the real snapshot $\boldsymbol{X}_t$.
The graph simulator takes in a graph snapshot and extracts *spatial* features that are then fed into a multilayer perceptron.

$$sg_{\boldsymbol{C}}(\boldsymbol{X}_t) = mlp_{\boldsymbol{C}_m}(gcn_{\boldsymbol{C}_g}(\boldsymbol{X}_t)) \tag{5.1}$$

The parameters $\boldsymbol{C}$ are shared for every node in the graph, so all nodes behave identically given the same state.
The gcn is a **GraphConv**. This type of gnn only aggregates information of first order neighbours by summation, so the spatial features extracted are local, then linearly combines the aggregated features.
The mlp is introduced to process the extracted spatial features, and combine them non linearly (a leaky Relu is used as non linear activation function). Also in Sindy the spatial features are first aggregated by summation then processed non linearly by the dictionary functions.

The objective of the problem to solve in the case of **node supervision** is:

$$min_{\boldsymbol{C}} \qquad \sum_{t,v,i} (sg_{\boldsymbol{C}}(\boldsymbol{X}_t)_{v,i} - \boldsymbol{X}_{t+1,v,i})^2 \tag{5.2}$$

If **aggregated supervision** is used, the loss is calculated by considering only the mean value of the features:

$$min_{\boldsymbol{C}} \qquad \sum_{i,t} (mean(sg^t_{\boldsymbol{C}}(\boldsymbol{X}_0)_{:,i}) - mean(\boldsymbol{X}_{t,:,i}))^2 \tag{5.3}$$

The constraining works by defining a parallel *problem(b)* that searches for the states that maximally break the constraints. In this case, because the *mlp* introduces complex non linearities solving *problem(b)* becomes more complicated.
A solver must deal with a non linear objective of linear constraints. A solution is to use **random sampling**, another more precise technique would be to solve *(b)* using the method of *lagrange multipliers*. The solution is not guaranteed to be optimal though, because there is un unknown number of local mimimums and the method of lagrange multipliers is not guaranteed to find the global optimum in this case.
Given **approximate constraints** then *problem(a)* is solved by using objectives (5.2) or (5.3) and the method of *lagrange multipliers* as with Sindy graph.

5.3.1 Implementation

The **training** is divided in 2 phases: a first phase in which the *mlp* layers are ignored, and only the *gcn* is trained.
In the second phase some dense layers are introduced (their number and the number of hyper parameters are given by hyper parameters) and initialized to the identity matrix, in order not to make them change the output of the *gcn*. Then the *gcn*'s parameters are freezed and only the dense layers are trained.
Each phase is trained using either node or aggregated supervision, as seen for sindy graph.

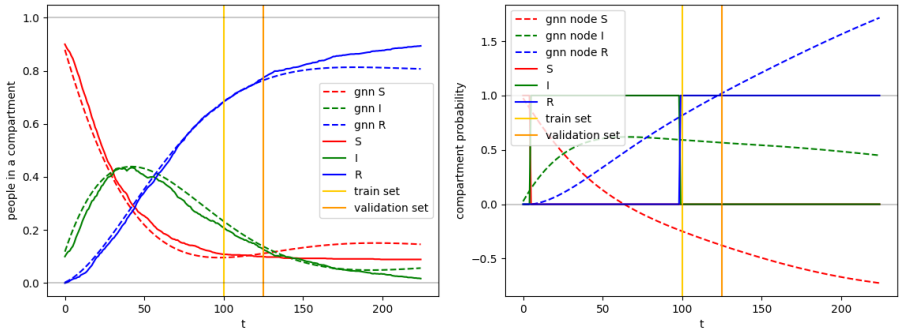
The constraints are enforced by using the method of Lagrange multipliers for optimizing. To write the constraints (4.9) the difference function is required. The difference function is found by feeding the *gcn* directly with the random states, skipping the message passing step and then differentiating the input and the output of (5.1).

5.3.2 STGNN with aggregated supervision

This technique used with aggregated supervision to understand how it compares to Sindy graph with aggregated supervision 100.

Dataset 1	
constraints	no
supervision	aggregated
train,validation,test size	[100,125,225]
learning rates	[0.00005]
mlp neurons	[3,15]
dense layers	[1,2,3]
tolerance epochs	2000
bootstrap increment	25
mean time	18583s
mean constr error	$10181 * 10^{-4}$
mean train error	0.0002
mean test MSE	$317.23 * 10^{-4}$
mean test MAPE	139.54%
mean test forecast 10%	0.0
mean test forecast 20%	0.36
failed tests	0

Looking at the table it is immediately clear how the generalization error is very bad. This is due to the huge constraints error.



(a) Aggregated plot (b) Node's dynamics

Figure 5.7

5.3.3 STGNN with aggregated supervision and constraints

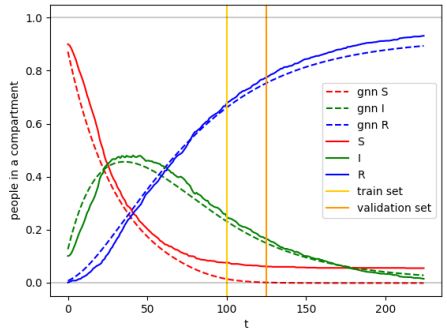
This technique is directly compared to Sindy graph with aggregated supervision and constraints 100.

Dataset 1	
constraints	yes
supervision	aggregated
train,validation,test size	[100,125,225]
learning rates (primal,dual)	[0.01],[1]
mlp neurons	[3,15]
dense layers	[1,2,3]
tolerance epochs	3000
bootstrap increment	25
mean time	14535s
mean constr error	$400.2 * 10^{-4}$
mean train error	0.00027
mean test MSE	$76.9 * 10^{-4}$
mean test MAPE	71.92%
mean test forecast 10%	0.0
mean test forecast 20%	0.0
failed tests	0

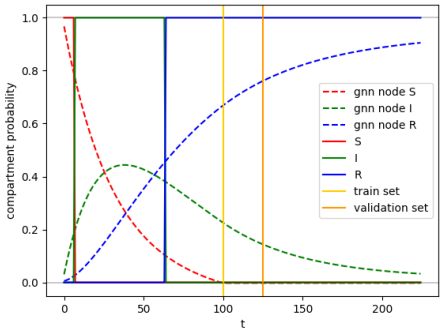
STGNNs with aggregated supervision and constraints tend to perform bad when considering the constraints error. This is due to the fact that constraining neural networks is much harder because of the complexity of the learned function. In fact the constraints are approximated much worse here than with Sindy when using Rand Sampling. Also the constrained optimizer struggles more as it deals with more complexity.
Because of the reduced physics information encoded into STGNNs and the constraints violation, the generalization error is much worse than Sindy.

Because the activation function for the MLP is a LeakyRelu, if the node dynamics approaches 0, it is clamped, avoiding negative values. Other activation functions have been tested but LeakyRelu is the one that performs best.

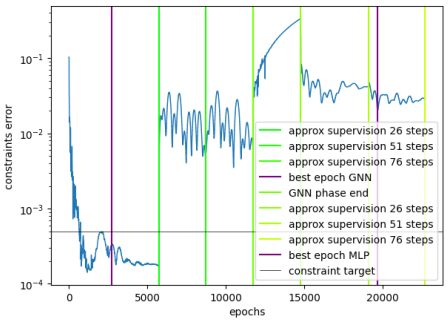
From the constraints 5.8c and validation 5.8d plots it can be seen that gnns struggle to find a good compromise between constraints satisfaction and generalization error.
In fact if no MLP layer is used, the GCN is similar to Sindy, with only linear terms in the dictionary. Because of this restriction, it is good at satisfying the constraints when the supervision is simple. When the supervision becomes more realistic because the approximated supervision window grows though, the validation error improves but the constraints satisfaction worsens. This is probably due to the limited expressivity of GCNs.
Adding the MLP introduces non linearities and improves the expressivity, but they but this does not seem to help much. Instead the constraints satisfaction is even worse as it is harder to force it.



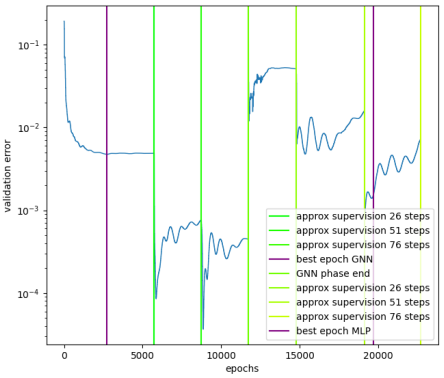
(a) Aggregated plot



(b) Node's dynamics



(c) Constraints evolution



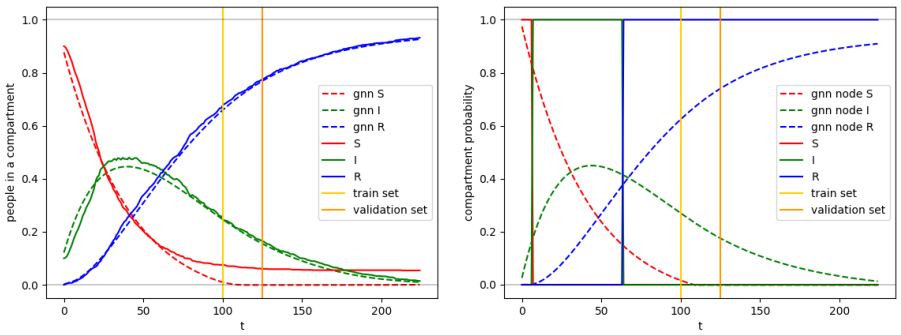
(d) validation error

Figure 5.8

5.3.4 STGNN with node supervision

Dataset 1	GNN node no constr 100
constraints	no
supervision	node
train,validation,test size	[100,125,225]
learning rates	[0.002]
mlp neurons	[3,15]
dense layers	[1,2,3]
tolerance epochs	3000
mean time	747s
mean constr error	$429 * 10^{-4}$
mean train error	0.010
mean test MSE	$25.96 * 10^{-4}$
mean test MAPE	46.54%
mean test forecast 10%	0.36
mean test forecast 20%	1.05
failed tests	0

Using node supervision increases the performance of STGNNs, differently from Sindy Graph. This is probably because STGNNs are not supervisioned with the time derivative of nodes' signals but directly with the signal, avoiding the noise amplification feature of the derivative operation.



(a) Aggregated plot

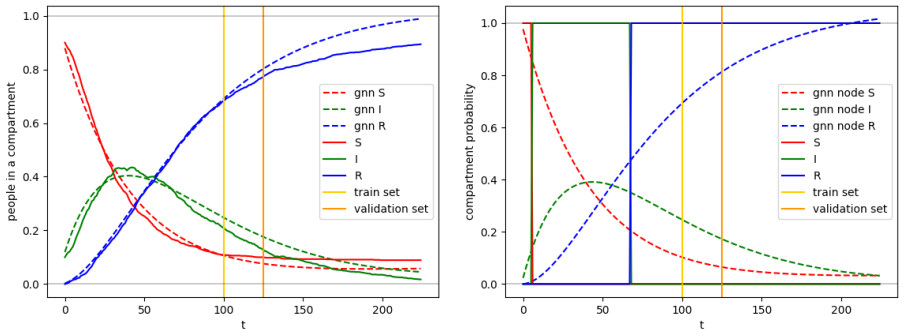
(b) Node's dynamics

Figure 5.9

5.3.5 STGNN with node supervision and constraints

Dataset 1	GNN node 100
constraints	yes
supervision	node
train,validation,test size	[100,125,225]
learning rates (primal,dual)	[0.02],[0.02]
mlp neurons	[3,15]
dense layers	[1,2,3]
tolerance epochs	3000
mean time	301s
mean constr error	$673 * 10^{-4}$
mean train error	0.010
mean test MSE	$34.74 * 10^{-4}$
mean test MAPE	54.82%
mean test forecast 10%	0.0
mean test forecast 20%	0.0
failed tests	0

Adding constraints in this case does not improve performance, confirming the bad interaction between STGNNs and constraints.



(a) Aggregated plot (b) Node's dynamics

Figure 5.10

5.4 Comparative table

method name	**Sindy node 100**	**Sindy node constr 100**	**Sindy aggr 100**	**Sindy aggr 150**
mean constrain error	$856 * 10^{-4}$	$3.99 * 10^{-4}$	$1064 * 10^{-4}$	$1266 * 10^{-4}$
test MSE	$27.68 * 10^{-4}$	$12.83 * 10^{-4}$	$15.19 * 10^{-4}$	$5.61 * 10^{-4}$
test MAPE	40.96%	37.02%	30.27%	31.93%
mean test forecast 10%	5.0	0.0	1.63	14.78
mean test forecast 20%	9.63	0.0	9.05	35.31

method name	**Sindy aggr constr 100**	**Sindy aggr constr 150**	**STGNN aggr**	**STGNN aggr constr**
mean constrain error	**$2.68 * 10^{-4}$**	$18.13 * 10^{-4}$	**$10181 * 10^{-4}$**	$400.2 * 10^{-4}$
test MSE	**$6.27 * 10^{-4}$**	$2.33 * 10^{-4}$	**$317.23 * 10^{-4}$**	$76.9 * 10^{-4}$
test MAPE	**24.86%**	46.13%	**139.54%**	71.92%
mean test forecast 10%	**12.68**	0.6	**0.0**	0.0
mean test forecast 20%	**29.78**	9.4	**0.36**	0.0

method name	**STGNN node**	**STGNN node constr**
mean constrain error	$429 * 10^{-4}$	$673 * 10^{-4}$
test MSE	$25.96 * 10^{-4}$	$34.74 * 10^{-4}$
test MAPE	46.54%	54.82%
mean test forecast 10%	0.36	0.0
mean test forecast 20%	1.05	0.0

The tests performed are done to compare the technique with all our contributions and a standard technique adapted to work with aggregated supervision.
From the experiments it can be seen that Sindy graph with aggregated supervision and constraints 100 performs much better than STGNN with aggregated supervision. There are multiple reasons why Sindy Graph proves to have superior performance.

First, all the methods that use Sindy as basis perform better, considering the generalization error and test forecasts lengths, than STGNNs with the same techniques used on top. This means that Sindy is better in general at discovering dynamics, and this can be attributed to the fact that Sindy is more physically informed. In fact the dictionary functions can be chosen to reflect domain knowledge.
In our case, polynomials of degree 2 have been chosen to reflect the characteristics of terms present in the standard SIR model, providing a solid clue on how a model should be like.
Dictionary functions are also chosen to be relatively simple, this helps much with the satisfaction of constraints.

Adding constraints to Sindy Graph helps in various ways. The generalization errors become lower, except when using longer training sets. Solutions to the learned differential equations don't tend to explode after a certain period of time, remaining bounded. Models are more physically correct and this, to-

gether with the interpretability of Sindy, makes it possible to reason on and study the analysed phenomena.

Instead STGNNs satisfy constraits much less, by having higher mean constraint error everywhere except for STGNN with node supervision. Moreover adding constraints to an STGNN technique does not always mean lower constraint error.
This can be attributed to the higher complexity of neural networks, that create more complex hypothesis functions than Sindy. This makes minima searching techniques such as Random Sampling work bad, leading to a constrained problem formulation that is too approximated to really enforce the wanted constraints.

Another thing can be noticed from the experiments. With Sindy Graph, aggregated supervision works better than node supervision and this is probably due to the sensitivity of Sindy to noise.

To conclude, Sindy Graph with aggregated supervision and constraints proves to be a powerful tool for epidemiological forecasting, outperforming all other tested techniques with the same training set length, thanks to the physics information that can be included in Sindy and the constraints.

Chapter 6

Conclusions

Epidemiological modelling is a difficult task that requires human expertise. To make the task easier, classical machine learning techniques may be considered, but they require a lot of data, are not interpretable and it is not easy to integrate domain knowledge.
In order to get a better model, useful for forecasting the infection dynamics, machine learning should be enhanced with human knowledge in the form of physics informed machine learning.

The first form of physical information exploited in our techniques is the graph structure of the social networks where the infection spreads. Graphs are very powerful tools for modelling discrete entities that have some sort of relationship and can be used as a computational carrier for passing information between nodes following a structure.
The structure of the graph can be decoupled from the dynamics of interaction between nodes that can be learned separately. The same interaction dynamics will work then on different graphs without the need for retraining.

Another form of physical information added is constraints. By requiring that the learned simulator outputs a dynamics with certain characteristics we are adding domain knowledge, meaning that the hypothesis space is smaller and contains only well behaved functions. This has important consequences:
Thanks to the bound constraints the explosion of solutions of the simulator is avoided and predictions can be performed for large time frames.
With constraints the training is more stable (fig 5.3c) and can be performed at much higher learning rates, improving learning time.
Without constraints, Sindy has more difficulty finding good terms because of the reduced regularization.
The tests indicate that with constraints the generalization errors for Sindy Graph are lower.
A technique has been devised for dealing with Sindy constraints specifically when the dictionary functions are polynomials of degree 2, called Quadratic programming with updating objective. It demonstrates that Sindy's interpretability also helps with the enforcing of constraints.

A classic technique has been tested, for learning a temporal model of the dy-

namics on graph, that uses a GCN to aggregate neighbours features, plus a MLP that adds non linearities to the learned simulator. This technique is an instance of STGNNs.
This technique whether with or without aggregated supervision has proven ineffective for forecasting. This is due to the poor expressive power of the GCN and the fact that the MLP makes the model very difficult to constrain. In fact it is difficult to have an expressive model that is also easy to constrain.

The other technique used to tackle the problem is Sindy that is used to find sparse nonlinear symbolic differential equations that describe a dynamics from data. This technique has been modified by us to work on graphs enforcing some constraints.
Using this method it is possible to define a dictionary of functions that will become terms in the differential equations. Because these functions are represented symbolically, it is easy to integrate more domain knowledge into the model.
Moreover the dictionary functions can be chosen to make constraining easier, as it is not a simple task in general.
Because of these reasons and the regularization given by the selection of terms done by Sindy, we found that this technique is more powerful than STGNNs, and is able to provide good forecasting capabilities.
Sindy's models are interpretable and it is interesting to see that the most important terms from their symbolic equations reflect the expected behaviour. Interpretability enables understanding and reasoning about the learned model, and this helps with scientific discovery.

There are many ways in which our work could be extended.
More powerful optimizers could be implemented for dealing with constraints, that could be used for training Sindy Graph with more complex dictionary functions, improving the expressivity of the method.
More features could be added in the graph nodes, for example the age of each person, to improve the realisticness of the models.
Real world data could be used to train the models, to see how our method performs concretely.
The developed techniques could be also used to study other physical phenomena characterized by a graph dynamics, for example material deformations in mechanical engineering.

Studying epidemiology helps preventing and controlling serious outbreaks of viruses, that threaten public health. We hope that with our work we will contribute to understand and prevent future viral diseases.

Appendix A

Algorithms

A.1 Quadratic programming

Let p be a multivariate polynomial of degree 2 restricted to K that is a compact subset of $\mathbb{R}^n$ determined by linear constraints. The task is to prove some properties of p that will lead to an exact algorithm to calculate its minimum restricted to K.

Property 1: for every possible unconstrained p there exist 0, 1 or infinite stationary points.
To find the stationary points of p it is sufficient to check for which $\boldsymbol{y}$

$$\nabla p(\boldsymbol{y}) = 0$$

Because the partial derivative of p with respect to any variable is a linear function

$$\frac{\partial p}{\partial \boldsymbol{y}_i} = a\boldsymbol{y}_i + b$$

Finding the set of $\boldsymbol{y}$ corresponding to stationary points becomes solving a linear system of equations

$$\boldsymbol{A}\boldsymbol{y} = \boldsymbol{b}$$

Now, a linear system can have either 0, 1 or infinite solutions.

Property 2: if p has infinite stationary points they all have the same value.
This means that the system has infinite solutions S and $p(\boldsymbol{s}) = k \qquad \forall \boldsymbol{s} \in S$.
S is a linear affine subspace of $\mathbb{R}^n$.
Lets choose an $\boldsymbol{s'} \in S$, because it corresponds to a stationary point $\nabla p(\boldsymbol{s'}) = 0$

$$p(\boldsymbol{s'}) = p(\boldsymbol{s'} + d\boldsymbol{v})$$

where $\boldsymbol{v} \in \mathbb{R}^n$ is a vector parallel to S. Because S is **continuous**, every continuous movement on it does not change the value of p. This implies that $p(\boldsymbol{s}) = k \qquad \forall \boldsymbol{s} \in S$.

Property 3: if $p(S)$ is the image of a polynomial of degree 2 evaluated on an linear affine subspace S of $\mathbb{R}^n$ (geometrically a *flat*) of dimension d, it is

possible to construct $p' : \mathbb{R}^d \to \mathbb{R}$ so that p' is still a polynomial of degree 2 and $p(S) = p'(\mathbb{R}^d)$.
A flat of $\mathbb{R}^n$ with d degrees of freedom can be represented by a system of **linear** parametric equations

$$\boldsymbol{y} = \boldsymbol{A}\boldsymbol{t} + \boldsymbol{b}$$

where $\boldsymbol{A}$ is a $n \times d$ matrix, $\boldsymbol{t} \in \mathbb{R}^d$ is the parameters vector, $\boldsymbol{b} \in \mathbb{R}^n$ and $\boldsymbol{y'} \in \mathbb{R}^n$. This system describes a function $f : \mathbb{R}^d \to \mathbb{R}^n$ that maps the vector of parameters $\boldsymbol{t}$ into unique points on the flat in $\mathbb{R}^n$, and $f(\mathbb{R}^d) = S$.
Now, the function p can be evaluated on the flat by doing a function composition with f:

$$p'(\boldsymbol{t}) = p(f(\boldsymbol{t}))$$

So $p' : \mathbb{R}^d \to \mathbb{R}$ and $p'(\mathbb{R}^d) = p(S)$.
Moreover p' is still a polynomial of degree 2 because p contains monomials that involve multiplications between at most 2 variables (e.g. $c_k \boldsymbol{y}_i * \boldsymbol{y}_j$) and because of the composition with f, those variables are linear combinations of parameters $\boldsymbol{t}$:

$$c_k \boldsymbol{y}_i * \boldsymbol{y}_j = c_k(\boldsymbol{A}_{i,:}\boldsymbol{t} + \boldsymbol{b}_i) * (\boldsymbol{A}_{j,:}\boldsymbol{t} + \boldsymbol{b}_j)$$

because any multiplication involves only linear functions, the maximum degree of any monomial in p' is 2.

With these properties it is possible to understand how to find the $\boldsymbol{y}$ corresponding to the minimum of a polynomial p of degree 2 when **restricted to a flat** S of parametric function f:

```
def min_flat(p,f):
  if f = y: return y
  p_(t) = p(f(t))
  g(t) = grad(p_)
  sol = solve(g(t) = 0)
  if sol = ∅ or not is_min(sol[0]): return None
  return f(sol)
```

Listing A.1: min flat function

If f has $d = 0$ it represents a flat that is a point, that point is automatically the $\boldsymbol{y}$ corresponding to the minimum.
Otherwise the polynomial p' restricted to the flat is given symbolically by composition. Because of *property 3* it has degree 2, and its image is equal to $p(S)$, so contains the same minimum.
Then its gradient is evaluated symbolically by taking the partial derivatives with respect to the parameters $\boldsymbol{t}$. The set of $\boldsymbol{t}$ that correspond to stationary points is found by solving a linear system. This is possible because p' has degree 2 so the partial derivatives are linear.
For *property 1* the system has either 0, 1 or infinite solutions. If the solutions are 0 there is no stationary point, so there cannot be a minimum and the polynomial does not have a lower bound. For *property 2* if there are infinite solutions they must all evaluate to the same value so the first is picked. The stationary point can be a minimum and in that case it is a global minimum as no minimum with different value can exist (*property 2*). If it is not a minimum then there is none and the polynomial is again not lower bounded.

Because solving the system gives a $\boldsymbol{t}$ corresponding to a stationary point, the non parametrized solution is found by computing $\boldsymbol{y} = f(\boldsymbol{t})$.
So *min_flat* returns the input(s) $\boldsymbol{y}$ corresponding to the global minimum of p restricted to S if it exists, otherwise None.

It is now possible to understand how to find the $\boldsymbol{y}$ corresponding to the minimum of p when $\boldsymbol{y} \in K$ for a K determined by a **system of linear equality constraints**:

$$K = \{\boldsymbol{y} \in \mathbb{R}^n | \boldsymbol{A}\boldsymbol{y} = \boldsymbol{b}\}$$

```
def min_equality(p,A,b):
  sol = solve(Ay = b)
  if sol = ∅: return None
  f(t) = find_flat(sol)
  return min_flat(p,f)
```

Listing A.2: min equality function

If the system has no solution, $K = \emptyset$, so there cannot be any minimum.
Otherwise the system describes a *flat*. The same flat can be described by a linear parametric function f. There are actually more functions that describe the same flat and they can be found by keeping some variables free, these are the parameters $\boldsymbol{t}$, and map them linearly to $\mathbb{R}^n$. This linear map is determined by solving the system.
The $\boldsymbol{y}$ corresponding to the minimum is then found by computing *min_flat(p,f)*.

If K is determined by also including **inequalities**, finding the $\boldsymbol{y}$ corresponding to the minimum is a bit more involved.

Property 4: if K is a compact subset of $\mathbb{R}^n$, determined by linear equality or inequality constraints it is a *Convex polytope*.

$$K = \{\boldsymbol{y} \in \mathbb{R}^n | \boldsymbol{A}\boldsymbol{y} = \boldsymbol{b}, \\ \boldsymbol{C}\boldsymbol{y} \geq \boldsymbol{d}\}$$

Property 5: If $K \neq \emptyset$ is a convex polytope and K' is K but relaxed of the inequalities, if $\exists \boldsymbol{y} \in K$ such that $p(\boldsymbol{y}) = min(p(K'))$, then $p(\boldsymbol{y}) = min(p(K))$.
If there is no such $\boldsymbol{y}$, the $min(p(K))$ lays on the boundary of K.
If there is such $\boldsymbol{y}$, then because

$$K \subseteq K' = \{\boldsymbol{y} \in \mathbb{R}^n | \boldsymbol{A}\boldsymbol{y} = \boldsymbol{b}\}$$

$$p(K) \subseteq p(K')$$

$$min(p(K)) \geq min(p(K'))$$

then if $p(\boldsymbol{y}) = min(p(K'))$ then $p(\boldsymbol{y}) \leq min(p(K))$ but because $\boldsymbol{y} \in K$, $p(\boldsymbol{y}) = min(p(K))$.
If there is no such $\boldsymbol{y}$ then there is no local minimum of $p(\boldsymbol{k'}), \boldsymbol{k'} \in K'$ with $\boldsymbol{y} \in K$, this is true for *properties 1,2* and *3*. In fact if there are multiple stationary points on $p(\boldsymbol{y'}), \boldsymbol{y'} \in \mathbb{R}^n$ they must all have same value and this is true also for $p(\boldsymbol{k'}), \boldsymbol{k'} \in K'$ for *property 3*, as K' is a flat. So all local minima of $p(\boldsymbol{k'})$ is

also global minima. But by hypothesis $\nexists \boldsymbol{y} \in K$ such that $p(\boldsymbol{y}) = min(p(K'))$. Because there is no local minima of $p(\boldsymbol{k'})$ on K and $K \neq \emptyset$, $min(p(K))$ must lay on the boundary of K.

Property 5 suggest a way to search for the minimum were there are inequalities: either the minimum is inside K' or it is in a lower dimensional flat defined by transforming some inequalities into equalities making them active.
In general, we can define K_J to be the set of points of $\mathbb{R}^n$ where the set of inequalities corresponding to indexes in $J \in 2^D$ is active, D is the set of all inqualities indexes and $m = |J|$:

$$K_J = \{\boldsymbol{y} \in \mathbb{R}^n | \boldsymbol{A}\boldsymbol{y} = \boldsymbol{b}, \\ \boldsymbol{C}_{J,:}\boldsymbol{y} = \boldsymbol{d}_J\}$$

where $\boldsymbol{C}_{J,:}$ is a $m \times n$ matrix.

Now, to find the $\boldsymbol{y}$ corresponding to the minimum of p restricted to K it is sufficient to search for a minimum in all possible active sets. If there are many minimums it is sufficient to choose the smallest:

$$\boldsymbol{y}_{min} = arg \min_{J \in 2^D} p(min\_equality(p, \boldsymbol{C}'_J, \boldsymbol{d}'_J) \cap K) \tag{A.1}$$

where

$$\boldsymbol{C}'_J = concat(\boldsymbol{A}, \boldsymbol{C}_{J,:})$$
$$\boldsymbol{d}'_J = concat(\boldsymbol{b}, \boldsymbol{d}_J)$$

Bibliography

[1] Clarence W. Rowley et al. "Spectral Analysis of Nonlinear Flows". In: *Journal of Fluid Mechanics* 641 (Dec. 25, 2009), pp. 115–127. ISSN: 0022-1120, 1469-7645. DOI: 10.1017/S0022112009992059. URL: https://www.cambridge.org/core/product/identifier/S0022112009992059/type/journal_article (visited on 12/21/2022).

[2] M.G. Garner and S.A. Hamilton. "Principles of Epidemiological Modelling: -EN- -FR- Les Principes de La Modélisation Épidémiologique -ES- Principios de Modelización Epidemiológica". In: *Revue Scientifique et Technique de l'OIE* 30.2 (Aug. 1, 2011), pp. 407–416. ISSN: 0253-1933. DOI: 10.20506/rst.30.2.2045. URL: https://doc.oie.int/dyn/portal/index.xhtml?page=alo&aloId=31251 (visited on 01/02/2023).

[3] Steven L. Brunton, Joshua L. Proctor, and J. Nathan Kutz. "Discovering Governing Equations from Data by Sparse Identification of Nonlinear Dynamical Systems". In: *Proceedings of the National Academy of Sciences* 113.15 (Apr. 12, 2016), pp. 3932–3937. DOI: 10.1073/pnas.1517384113. URL: https://www.pnas.org/doi/10.1073/pnas.1517384113 (visited on 03/29/2022).

[4] Michael M. Bronstein et al. "Geometric Deep Learning: Going beyond Euclidean Data". In: *IEEE Signal Processing Magazine* 34.4 (July 2017), pp. 18–42. ISSN: 1053-5888, 1558-0792. DOI: 10.1109/MSP.2017.2693418. arXiv: 1611.08097 [cs]. URL: http://arxiv.org/abs/1611.08097 (visited on 12/29/2022).

[5] Justin Gilmer et al. *Neural Message Passing for Quantum Chemistry*. June 12, 2017. arXiv: arXiv:1704.01212. URL: http://arxiv.org/abs/1704.01212 (visited on 12/19/2022). preprint.

[6] Justin Gilmer et al. *Neural Message Passing for Quantum Chemistry*. June 12, 2017. arXiv: arXiv:1704.01212. URL: http://arxiv.org/abs/1704.01212 (visited on 12/30/2022). preprint.

[7] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. *Physics Informed Deep Learning (Part I): Data-driven Solutions of Nonlinear Partial Differential Equations*. Nov. 28, 2017. arXiv: arXiv:1711.10561. URL: http://arxiv.org/abs/1711.10561 (visited on 12/23/2022). preprint.

[8] Thomas Vandal et al. *DeepSD: Generating High Resolution Climate Change Projections through Single Image Super-Resolution*. Mar. 8, 2017. arXiv: arXiv:1703.03126. URL: http://arxiv.org/abs/1703.03126 (visited on 12/19/2022). preprint.

[9] Bethany Lusch, J. Nathan Kutz, and Steven L. Brunton. "Deep Learning for Universal Linear Embeddings of Nonlinear Dynamics". In: *Nature Communications* 9.1 (Dec. 2018), p. 4950. ISSN: 2041-1723. DOI: 10.1038/s41467-018-07210-0. arXiv: 1712.09707. URL: http://arxiv.org/abs/1712.09707 (visited on 03/21/2022).

[10] Giulio Rossetti et al. "NDlib: A Python Library to Model and Analyze Diffusion Processes Over Complex Networks". In: *International Journal of Data Science and Analytics* 5.1 (Feb. 2018), pp. 61–79. ISSN: 2364-415X, 2364-4168. DOI: 10.1007/s41060-017-0086-6. arXiv: 1801.05854 [cs]. URL: http://arxiv.org/abs/1801.05854 (visited on 01/13/2023).

[11] Alvaro Sanchez-Gonzalez et al. "Graph Networks as Learnable Physics Engines for Inference and Control". June 4, 2018. URL: http://arxiv.org/abs/1806.01242 (visited on 03/23/2022).

[12] Kathleen Champion et al. "Data-Driven Discovery of Coordinates and Governing Equations". In: *Proceedings of the National Academy of Sciences* 116.45 (Nov. 5, 2019), pp. 22445–22451. DOI: 10.1073/pnas.1906995116. URL: https://www.pnas.org/doi/10.1073/pnas.1906995116 (visited on 03/30/2022).

[13] Aryan Mokhtari, Asuman Ozdaglar, and Sarath Pattathil. *A Unified Analysis of Extra-gradient and Optimistic Gradient Methods for Saddle Point Problems: Proximal Point Approach*. Sept. 5, 2019. arXiv: arXiv:1901.08511. URL: http://arxiv.org/abs/1901.08511 (visited on 12/23/2022). preprint.

[14] Abdullah M. Almeshal et al. "Forecasting the Spread of COVID-19 in Kuwait Using Compartmental and Logistic Regression Models". In: *Applied Sciences* 10.10 (10 Jan. 2020), p. 3402. ISSN: 2076-3417. DOI: 10.3390/app10103402. URL: https://www.mdpi.com/2076-3417/10/10/3402 (visited on 01/08/2023).

[15] Ricardo Manuel Arias Velásquez and Jennifer Vanessa Mejía Lara. "Forecast and Evaluation of COVID-19 Spreading in USA with Reduced-Space Gaussian Process Regression". In: *Chaos, Solitons & Fractals* 136 (July 1, 2020), p. 109924. ISSN: 0960-0779. DOI: 10.1016/j.chaos.2020.109924. URL: https://www.sciencedirect.com/science/article/pii/S0960077920303234 (visited on 01/08/2023).

[16] Fabien Baradel et al. *CoPhy: Counterfactual Learning of Physical Dynamics*. Apr. 7, 2020. arXiv: arXiv:1909.12000. URL: http://arxiv.org/abs/1909.12000 (visited on 12/19/2022). preprint.

[17] Rabia M Chaudhry et al. "Coronavirus Disease 2019 (COVID-19): Forecast of an Emerging Urgency in Pakistan". In: *Cureus* (May 28, 2020). ISSN: 2168-8184. DOI: 10.7759/cureus.8346. URL: https://www.cureus.com/articles/32115-coronavirus-disease-2019-covid-19-forecast-of-an-emerging-urgency-in-pakistan (visited on 01/08/2023).

[18] Vinay Kumar Reddy Chimmula and Lei Zhang. "Time Series Forecasting of COVID-19 Transmission in Canada Using LSTM Networks". In: *Chaos, Solitons & Fractals* 135 (June 1, 2020), p. 109864. ISSN: 0960-0779. DOI: 10.1016/j.chaos.2020.109864. URL: https://www.sciencedirect.com/science/article/pii/S0960077920302642 (visited on 01/08/2023).

[19] Miles Cranmer et al. *Lagrangian Neural Networks*. July 30, 2020. arXiv: arXiv:2003.04630. URL: http://arxiv.org/abs/2003.04630 (visited on 12/20/2022). preprint.

[20] Amol Kapoor et al. *Examining COVID-19 Forecasting Using Spatio-Temporal Graph Neural Networks*. July 6, 2020. arXiv: arXiv:2007.03113. URL: http://arxiv.org/abs/2007.03113 (visited on 01/04/2023). preprint.

[21] Jungeun Kim et al. *DPM: A Novel Training Method for Physics-Informed Neural Networks in Extrapolation*. Dec. 4, 2020. arXiv: arXiv:2012.02681. URL: http://arxiv.org/abs/2012.02681 (visited on 12/22/2022). preprint.

[22] Leila Moftakhar, Mozhgan Seif, and Marziyeh Sadat Safe. "Exponentially Increasing Trend of Infected Patients with COVID-19 in Iran: A Comparison of Neural Network and ARIMA Forecasting Models". In: *Iranian Journal of Public Health* 49 (Supple 1 Apr. 28, 2020), pp. 92–100. ISSN: 2251-6093. DOI: 10.18502/ijph.v49iS1.3675. URL: https://ijph.tums.ac.ir/index.php/ijph/article/view/20549 (visited on 01/08/2023).

[23] Rohit Salgotra, Mostafa Gandomi, and Amir H Gandomi. "Time Series Analysis and Forecast of the COVID-19 Pandemic in India Using Genetic Programming". In: *Chaos, Solitons & Fractals* 138 (Sept. 1, 2020), p. 109945. ISSN: 0960-0779. DOI: 10.1016/j.chaos.2020.109945. URL: https://www.sciencedirect.com/science/article/pii/S0960077920303441 (visited on 01/08/2023).

[24] Alvaro Sanchez-Gonzalez et al. *Learning to Simulate Complex Physics with Graph Networks*. Sept. 14, 2020. arXiv: arXiv:2002.09405. URL: http://arxiv.org/abs/2002.09405 (visited on 12/19/2022). preprint.

[25] Michael M. Bronstein et al. *Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges*. May 2, 2021. arXiv: arXiv:2104.13478. URL: http://arxiv.org/abs/2104.13478 (visited on 12/20/2022). preprint.

[26] Michael M. Bronstein et al. *Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges*. May 2, 2021. arXiv: arXiv:2104.13478. URL: http://arxiv.org/abs/2104.13478 (visited on 01/01/2023). preprint.

[27] Steven L. Brunton et al. "Modern Koopman Theory for Dynamical Systems". Oct. 29, 2021. URL: http://arxiv.org/abs/2102.12086 (visited on 03/30/2022).

[28] Cornelius Fritz, Emilio Dorigatti, and David Rügamer. *Combining Graph Neural Networks and Spatio-temporal Disease Models to Predict COVID-19 Cases in Germany*. Jan. 3, 2021. arXiv: arXiv:2101.00661. URL: http://arxiv.org/abs/2101.00661 (visited on 01/04/2023). preprint.

[29] Junyi Gao et al. "STAN: Spatio-Temporal Attention Network for Pandemic Prediction Using Real-World Evidence". In: *Journal of the American Medical Informatics Association* 28.4 (Mar. 18, 2021), pp. 733–743. ISSN: 1527-974X. DOI: 10.1093/jamia/ocaa322. URL: https://academic.oup.com/jamia/article/28/4/733/6118380 (visited on 01/09/2023).

[30] Lu Lu et al. *Physics-Informed Neural Networks with Hard Constraints for Inverse Design*. Feb. 8, 2021. arXiv: arXiv:2102.04626. URL: http://arxiv.org/abs/2102.04626 (visited on 12/22/2022). preprint.

[31] Suyel Namasudra, S. Dhamodharavadhani, and R. Rathipriya. "Nonlinear Neural Network Based Forecasting Model for Predicting COVID-19 Cases". In: *Neural Processing Letters* (Apr. 1, 2021). ISSN: 1573-773X. DOI: 10.1007/s11063-021-10495-w. URL: https://doi.org/10.1007/s11063-021-10495-w (visited on 01/04/2023).

[32] George Panagopoulos, Giannis Nikolentzos, and Michalis Vazirgiannis. "Transfer Graph Neural Networks for Pandemic Forecasting". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 35.6 (6 May 18, 2021), pp. 4838–4845. ISSN: 2374-3468. DOI: 10.1609/aaai.v35i6.16616. URL: https://ojs.aaai.org/index.php/AAAI/article/view/16616 (visited on 01/04/2023).

[33] Iman Rahimi, Fang Chen, and Amir H. Gandomi. "A Review on COVID-19 Forecasting Models". In: *Neural Computing and Applications* (Feb. 4, 2021). ISSN: 1433-3058. DOI: 10.1007/s00521-020-05626-8. URL: https://doi.org/10.1007/s00521-020-05626-8 (visited on 01/04/2023).

[34] Benjamin Sanchez-Lengeling et al. "A Gentle Introduction to Graph Neural Networks". In: *Distill* 6.9 (Sept. 2, 2021), e33. ISSN: 2476-0757. DOI: 10.23915/distill.00033. URL: https://distill.pub/2021/gnn-intro (visited on 12/30/2022).

[35] Zonghan Wu et al. "A Comprehensive Survey on Graph Neural Networks". In: *IEEE Transactions on Neural Networks and Learning Systems* 32.1 (Jan. 2021), pp. 4–24. ISSN: 2162-237X, 2162-2388. DOI: 10.1109/TNNLS.2020.2978386. arXiv: 1901.00596 [cs, stat]. URL: http://arxiv.org/abs/1901.00596 (visited on 12/26/2022).

[36] Chuizheng Meng et al. *When Physics Meets Machine Learning: A Survey of Physics-Informed Machine Learning*. Mar. 31, 2022. arXiv: arXiv:2203.16797. URL: http://arxiv.org/abs/2203.16797 (visited on 12/15/2022). preprint.

[37] Petru Soviany et al. *Curriculum Learning: A Survey*. Apr. 11, 2022. arXiv: arXiv:2101.10382. URL: http://arxiv.org/abs/2101.10382 (visited on 10/26/2022). preprint.

[38] Andrew Cotter, Heinrich Jiang, and Karthik Sridharan. "Two-Player Games for Efficient Non-Convex Constrained Optimization". In: ().

[39] *What Is Epidemiology? — Teacher Roadmap — Career Paths to Public Health — CDC*. URL: https://www.cdc.gov/careerpaths/k12teacherroadmap/epidemiology.html (visited on 04/13/2023).