Deep statistical solvers & power systems applications

Balthazar Donon

université
PARIS-SACLAY

Deep Statistical Solvers & Power Systems Applications

Solveurs Statistiques Profonds & Applications au Réseau Électrique

Thèse de doctorat de l'université Paris-Saclay

École doctorale n° 580, Sciences et Technologies de l'Information et de la Communication (STIC)
Spécialité de doctorat : Informatique
Graduate School : Informatique et sciences du numérique
Référent : Faculté des Sciences d'Orsay

Thèse préparée dans l'unité de recherche LISN (Université Paris-Saclay, CNRS) et Inria Saclay-Île-de-France (Université Paris-Saclay, Inria), sous la direction d'Isabelle GUYON, Professeure, le co-encadrement de Marc SCHOENAUER, directeur de recherche INRIA et la co-supervision de Rémy CLÉMENT, responsable d'études R&D à RTE, entreprise.

Thèse soutenue à Paris-Saclay, le 16 mars 2022, par

Balthazar DONON

Composition du jury

Gaël VAROQUAUX Directeur de Recherche, INRIA	Président
Marc LELARGE Directeur de Recherche, INRIA	Rapporteur & Examinateur
Louis WEHENKEL Professeur, Université de Liège	Rapporteur & Examinateur
Patrick GALLINARI Professeur, Sorbonne Université	Examinateur
Madeleine GIBESCU Professeure, Université d'Utrecht	Examinatrice
Isabelle GUYON Professeure, Université Paris-Saclay	Directrice de thèse

THESE DE DOCTORAT

NNT : 2022UPASG016

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre: Solveurs Statistiques Profonds & Applications au Réseau Électrique
Mots clés: Réseau électrique, apprentissage profond, graph neural networks

Résumé: Confrontés à l'intégration croissante d'énergies renouvelables intermittentes et à de nouveaux mécanismes de marché, les réseaux électriques sont dans une phase de mutation profonde. Ainsi, face à une complexité croissante, RTE, le gestionnaire du réseau de transport d'électricité français, étudie les opportunités offertes par les méthodes issues du Deep Learning. Les changements de topologie (façon dont les lignes sont interconnectées) étant quotidiens, il est essentiel de permettre aux réseaux de neurones de prendre en compte la structure des données, ce qui est rendu possible par l'utilisation de Graph Neural Networks (GNNs). Après avoir démontré la capacité des GNNs à imiter un simulateur physique du réseau électrique, cette thèse développe une approche qui vise à "apprendre à optimiser" de façon non-supervisée. Un GNN est ainsi appris par minimisation directe des lois physiques, plutôt que par imitation. L'approche est par la suite étayée d'une analyse théorique, puis étendue à un problème d'optimisation à deux niveaux qui repose sur l'emploi de deux GNNs distincts, l'un d'entre eux jouant le rôle d'un opérateur, et l'autre émulant les lois physiques.

Title: Deep Statistical Solvers & Power Systems Applications
Keywords: Power systems, deep learning, graph neural networks

Abstract: Facing with the growing integration of intermittent renewable energies and disruptive market mechanisms, power systems are experiencing profound changes. To overcome this increasing complexity, RTE, the French Transmission System Operator, is investigating the use of methods arising from the Deep Learning literature. Topological changes (which affect the way power lines are interconnected) occur multiple times a day, and should thus be taken into account by the considered neural network architecture, which is made possible by Graph Neural Networks (GNNs). After having proven the ability of GNNs to imitate a power grid simulator, this PhD thesis develops an approach that aims at "learning to optimize" in an unsupervised fashion. A GNN is thus trained by direct minimization of physical laws, and not by imitation. This work is further elaborated by a theoretical analysis, and then extended to a bi-level optimization problem which requires the use of two distinct GNN models, one of them playing the role of an operator, while the other emulates physics.

Synthèse

Les réseaux de transport d'électricité sont confrontés à de multiples mutations, parmi lesquelles on peut citer l'insertion grandissante d'énergies renouvelables intermittentes et difficiles à prévoir, les nouvelles utilisations de l'énergie électrique (véhicules électriques), ou encore la libéralisation du marché de l'énergie qui tend à autoriser les centrales électriques à changer leur plan de production à des échéances très proches du temps réel. Ces changements rendent les flux électriques de plus en plus volatiles et difficiles à anticiper, avec pour conséquence de rendre l'exploitation des réseaux électriques plus complexe. Cette complexité grandissante pouvant à terme devenir trop importante pour les capacités cognitives humaines, il est nécessaire de développer de nouvelles méthodes pour alléger la charge de travail des opérateurs du réseau. Dans cette optique, Réseau de Transport de l'Électricité (RTE), le gestionnaire de réseau de transport français, étudie la possibilité d'employer diverses techniques issues du Deep Learning (DL).

Des travaux antérieurs au début de cette thèse ont été entrepris dans cette direction. Cependant, dans les approches précédentes, chaque modèle appris était propre à une instance de réseau. De tels modèles présentent comme limite de mal se généraliser lorsque la topologie du réseau électrique étudié est différente de celle qui a été considérée lors de l'entrainement. Or, des travaux de maintenance sur les ouvrages du réseau ainsi que des actions de la part des dispatchers pour aiguiller différemment les flux électriques se produisent quotidiennement sur le réseau électrique. Il existe donc un besoin de faire évoluer les méthodes de Machine Learning (ML) employées jusqu'alors de sorte à ce qu'elles soient capables de mieux appréhender une topologie changeante.

Dans ce but, cette thèse explore l'utilisation des Graph Neural Networks (GNNs) appliqués au réseau électrique. Les GNNs sont des réseaux de neurones dont l'architecture est particulièrement bien adaptée aux problèmes dont les entrées s'expriment sous forme de graphes. L'ensemble des travaux de cette thèse concernent le

problème *d'apprendre à optimiser*, c'est à dire créer des réseaux de neurones (des GNNs) qui résolvent des problèmes d'optimisation.

Une première contribution de cette thèse concerne l'utilisation de GNNs pour l'imitation d'un simulateur physique utilisé par RTE [1]. Les résultats démontrent la capacité des GNNs à apprendre sur des graphes qui ont des géométries différentes. Ces modèles ne sont pas seulement capables de généraliser à des réseaux dont la topologie diffère légèrement de celles vues lors de l'entrainement; des expériences démontrent par exemple la capacité à apprendre sur des graphes de 9 noeuds et à généraliser à des graphes de 1089 noeuds. Par ailleurs, des travaux menés en utilisant des données issues du réseau français réel ont mis en évidence le besoin d'utiliser une description du réseau électrique au moyen de structures de graphs plus complexes appelées Hyper Heterogeneous Multi Graphs (H2MGs). Ceci permet de ne pas altérer la structure du réseau électrique par une étape de pre-processing qui aggrègerait ensemble plusieurs objets, comme cela était fait jusqu'alors.

Une deuxième contribution de cette thèse [2] développe une approche basée sur les GNNs pour *apprendre à optimiser* de façon non supervisée. Elle consiste en l'apprentissage d'un modèle GNN par minimisation directe de la violation des lois physiques, au lieu d'apprendre de manière supervisée à partir des résultats d'un autre solveur. Cette méthode – que nous appelons Deep Statistical Solver (DSS) – est étayée, dans une troisième contribution [3], d'une analyse théorique fournissant une relation entre l'expressivité d'un modèle, le nombre d'étapes de propagation de messages, et le diamètre maximal des graphes présents dans les données. Les résultats expérimentaux montrent que l'approche non-supervisée est viable pour la résolution de systèmes linéaires issus de la discrétisation de l'équation de Poisson, ainsi que pour la simulation non-linéaire de réseaux allant de 14 à 118 noeuds.

Enfin, une quatrième contribution concerne le problème de contrôle de la tension (en boucle ouverte). Une nouvelle méthode de résolution est proposée et se base sur l'utilisation de deux modèles GNN. Un premier modèle (*contrôleur*) prend en entrée une situation de réseau et renvoie des consignes de tension des générateurs qui garantissent la sécurité du système. Un second modèle (*solveur*) prend en entrée la situation de réseau et la sortie du premier modèle, et résout les équations physiques du système. Ainsi la fonction de coût du *contrôleur* se base sur l'approximation de la physique fournie par le *solveur*. Cette approche à double réseau de neurones est d'une certaine façon semblable aux approches adversariales, bien qu'ici les ob-

jectifs des deux modèles ne soient pas nécessairement antagonistes. Des résultats préliminaires qui n'ont pas encore été publiés indiquent que l'approche est viable sur des réseaux jouets.

En conclusion, cette thèse s'est intéressée au développement de l'approche DSS qui vise à entraîner des GNNs à résoudre des problèmes d'optimisation variés de façon non supervisée. Le développement d'heuristiques basées sur des réseaux de neurones rapides offre la perspective d'accélérer la résolution de certains problèmes coûteux en temps de calculs, voire d'offrir des solutions à certains problèmes de prise de décision pour lesquels aucune méthode assez rapide n'existe actuellement.

Abstract

Power transportation networks are facing multiple changes, which include among others the growing amount of intermittent and hard to predict renewable energies, new ways of using electricity (electric vehicles), or the energy market liberalization which allows producers to change their plans on a short notice. Those changes make power flows increasingly volatile and hard to anticipate, with the result that the operation of power systems becomes even more complex. As this growing complexity may eventually become overwhelming for human cognitive capacities, it is necessary to develop new methods to lighten the workload of power grid operators. With this in mind, Réseau de Transport de l'Électricité (RTE), the French transmission system operator, is investigating the use of diverse methods stemming from the Deep Learning (DL) literature.

Prior to this work, artificial neural networks had already been used to perform various tasks on power grids. However, in previous approaches, each trained model was specific to a power grid instance. Therefore, such models suffer from the limitation that they do not generalize well when the topology of the studied electrical network differs from the one that has been considered during the training phase. Yet, maintenance work on the network structures as well as actions from the dispatchers to redirect the electrical flows are common events that may occur multiple times a day in the actual system. Thus, there is a need to improve the Machine-Learning methods used until now so that they are able to better handle systems with changing topology.

To this end, this thesis explores the use of Graph Neural Networks (GNNs) applied to power grids. GNNs are neural networks whose architecture is particularly well suited to problems whose inputs can be expressed as graphs. All the works of this thesis concern the problem of *Learning to Optimize*, which amounts to training neural networks (GNNs) that solve optimization problems.

A first contribution of this thesis concerns the use of GNNs to imitate the physical simulator used by RTE [1]. The results demonstrate the ability of GNNs to learn on graphs that have different geometries.

These models are not only able to generalise to networks whose topology differs slightly from those seen during training; experiments have shown for example the ability to learn on graphs of 9 nodes and to generalise to graphs of 1089 nodes. Furthermore, work carried out using data from the real French network has highlighted the need to use a description of the electrical network by means of more complex graph structures called Hyper Heterogeneous Multi Graphs (H2MGs). This makes it possible to avoid altering the structure of the electrical network by a pre-processing step that would aggregate several objects together, as was done until now.

A second contribution of this thesis [2] is to develop an approach based on GNNs to *learn to optimise* in an unsupervised fashion. It consists in training a GNNs model by direct minimisation of the violation of physical laws, instead of building a surrogate model of another solver. This method – which we call Deep Statistical Solver (DSS) – is supported, in a third contribution [3], by a theoretical analysis providing a relationship between the expressivity of a model, the amount of message-passing operations performed by the GNN, and the maximum diameter of the graphs in the considered dataset. Experimental results show that the unsupervised approach is viable for the resolution of linear systems stemming from the discretization of Poisson's equation, and for the non-linear simulation of power grid that are composed of 14 and 118 buses.

Finally, a fourth contribution concerns the voltage control problem (in open loop). A novel resolution method based upon the use of two GNN models is proposed. The first model (*controller*) takes as input a power grid snapshot, and outputs voltage setpoints for generators that guarantee that the whole system is in security. A second model (*solver*) takes the snapshot and the output of the controller as input, and solves the physical equations that govern the system. Thus, the cost function of the controller depends on the approximation of physics provided by the solver. This dual neural network approach is somewhat similar to adversarial approaches, although in our case the goals of the two models are not necessarily antagonistic. Preliminary results not yet published indicate that the approach is viable on toy networks.

As a conclusion, this PhD thesis is devoted to developing an approach, called Deep Statistical Solver, which amounts to training Graph Neural Networks to solve various optimization problems in an unsupervised fashion. The development of heuristics based on fast neural networks paves the way for accelerating some heavy computations, and providing more tractable solutions to highly complex control problems.

Acknowledgement

I am extremely grateful to Isabelle Guyon for her experience and guidance. I am deeply indebted to Marc Schoenauer for his invaluable contribution, the quality of his advices and his thoughtfulness. I would also like to express my deepest gratitude to Rémy Clément for his unparalleled support and his insightful suggestions.

I cannot begin to express my thanks to Patrick Panciatici and Lucas Saludjian for their profound belief in my work and invaluable support.

I had the great pleasure of working with Victor Berger, Laure Crochepierre, Zhengying Liu, Wenzhuo Liu, Julie Sliwak, and the TAU team as a whole, who have all significantly contributed to this PhD, whether it be through experiments, theoretical conversations, or insightful discussions. I wish to thank Guillaume Houry and Maxime Sanchez, whom I had the chance to mentor during their respective internships.

I would also like to extend my gratitude to my dear colleagues and friends from RTE, Vincent Barbesant, Pauline Gambier-Morel, Sylvain Leclerc, Étienne Lesot and Camille Pache for their kindness and friendliness. I gratefully acknowledge Marie-Sophie Debry for her listening capacity and her very precious help. I extend my sincere thanks to Guillaume Trimbach for his help. I would like to acknowledge the assistance of Benjamin Donnot and Antoine Marot during the beginning of my PhD.

I would like to thank my dear friends Benoît Duroi, Xavier Falque, Alexis Léautier, and Wojciech Sitarz for their unconditional camaraderie and goodwill during those busy years, Anne-Sophie Ambroisine for her kindness and support, and Isaac for having made these last years of research more peaceful and less lonely.

I deeply thank my parents Brigitte and Marc Donon and my sister Clémence Donon for their unconditional love, patience and understanding.

Finally I thank my beloved Sibylle for her unerring support during those trying years, for her empathy, and for offering me a love I hope to be worthy of.

Contents

List of Figures x

List of Tables xii

Glossary xiii

Introduction 1

I Background 19

1 Power Systems 21
- 1.1 Transporting power using electricity . . . 21
- 1.2 AC Power Systems . . . 24
- 1.3 Power grid modelling . . . 28

2 Deep Learning 35
- 2.1 Machine Learning . . . 35
- 2.2 Artificial Neural Networks . . . 37
- 2.3 Training Neural Networks . . . 41
- 2.4 Convolutional Neural Networks . . . 46

3 Graph Neural Networks 51
- 3.1 Graph data . . . 51
- 3.2 Graph Neural Networks . . . 58

II The Deep Statistical Solver Approach 65

4 Deep Statistical Solver Architecture 67
- 4.1 Hyper Heterogeneous Multi Graphs . . . 67
- 4.2 H2MGNN Architecture . . . 71
- 4.3 Implementation considerations . . . 75
- 4.A Historical DSS architecture . . . 79

5 Statistical Solver Problems **81**
5.1 Single-level unconstrained optimization . . . 82
5.2 Bilevel optimization . . . 87

6 Proving that global continuous problems can be solved through local operations **95**
6.1 Universal approximation theorem . . . 96
6.2 Proof of the Universal Approximation theorem . . . 100

III Applications **113**

7 Toy Examples **115**
7.1 System of springs . . . 116
7.2 Discretized Poisson equation . . . 124

8 AC Power Flow **131**
8.1 Synthetic data experiments . . . 131
8.2 Real data experiments . . . 136

IV Conclusion & Future Research **143**

9 Discussion and future work **145**
9.1 Voltage control . . . 145
9.2 Incorporating time . . . 152
9.3 Incorporating uncertainties . . . 154
9.4 Future research . . . 156
9.5 Long term vision . . . 158

10 Conclusion **159**

Bibliography **160**

List of Figures

1 Global energy consumed per year . . . 2
2 Global CO_2 emissions per year . . . 3
3 Temperature anomaly . . . 3
4 Renewable energy generation per year . . . 4
5 Californian duck curve . . . 5
6 Overall organization of power grids . . . 6
7 Hierarchy of scientific fields . . . 10

1.1 Dipole and quadrupole . . . 22
1.2 Electric and single-line diagrams . . . 24
1.3 Transformer . . . 25
1.4 Instantaneous power in AC . . . 27
1.5 IEEE case14 . . . 29
1.6 Transmission line electric diagram . . . 32
1.7 Transformer electric diagram . . . 33

2.1 Underfitting and overfitting . . . 38
2.2 Biological neuron . . . 39
2.3 Artificial neuron . . . 39
2.4 Activation functions . . . 40
2.5 Artificial neuron, Single-Layer Perceptron and Multi-Layer Perceptron . . . 41
2.6 Gradient back-propagation . . . 45
2.7 Discretization of a 2d image . . . 47
2.8 Convolution operation . . . 49

3.1 Standard graph . . . 52
3.2 Standard graph representation of a power grid . . . 53
3.3 Dense and sparse representations . . . 54
3.4 Permutation of a graph . . . 55
3.5 Permutation-invariant mapping . . . 56
3.6 Permutation-equivariant mapping . . . 56
3.7 Graph probability distribution . . . 57
3.8 Image and graph structures . . . 59
3.9 GNN encoding step . . . 61

3.10 GNN message passing . . . 62
3.11 GNN decoding step . . . 63
3.12 Information flow in a GNN . . . 64

4.1 H2MG representation of a power grid . . . 70
4.2 H2MG input graph . . . 76
4.3 H2MGNN vertex update . . . 76
4.4 H2MGNN hyper-edge latent update . . . 76
4.5 H2MGNN hyper-edge output update . . . 76
4.6 Data pre and post processing . . . 77
4.7 Previously used architecture . . . 79

5.1 Single level AC-PF problem . . . 82
5.2 Single level SSP . . . 86
5.3 Bilevel voltage control problem . . . 88
5.4 Bilevel SSP . . . 90

6.1 Kernel function . . . 108

7.1 Spring systems of various sizes . . . 117
7.2 Spring system instance . . . 118
7.3 Out-of-distribution correlations for springs systems . . . 121
7.4 Latent variables evolution . . . 122
7.5 Latent variables trajectories . . . 123
7.6 Discretization of spatial domains . . . 125
7.7 Intermediate losses and predictions . . . 128
7.8 Out-of-distribution generalization for different graph sizes . . . 128
7.9 Out-of-distribution generalization for different features . . . 129

8.1 IEEE case14 and case118 . . . 132
8.2 Real data experiment . . . 141

9.1 Voltage setpoint control . . . 146
9.2 Voltage control correlation with baseline . . . 149
9.3 Voltage control sensitivity analysis . . . 151
9.4 Graph time-series factorization . . . 153
9.5 Factorizable power grids . . . 154

List of Tables

1.1 Features and metrics in power grids 34

7.1 Spring experiment results 120
7.2 Discretized Poisson equation experiment results 127

8.1 AC-PF experiment on artificial data results 135
8.2 AC-PF experiment on real data results 140

9.1 Voltage control results 149

Introduction

Power Systems (PS) are a key component of modern societies. They enable energy transportation from places where it is produced (nuclear or fossile power plants, hydro-electric generators, wind turbines, solar panels, etc.) to places of consumption (houses, factories, public lighting, etc.). Their use is vital to the well being of a country, its citizens and its economy. It relies on thousands of kilometers of transmission lines, and on the ongoing work of thousands of people. Power Systems have been running for more than a century, and have enabled the development of countless improvements in our daily lives. Relying on electricity has become so common that one may take it for granted. Nonetheless, this domain is currently facing systemic changes which are driving the Power Systems community to take interest in innovations brought by the blooming field of Deep Learning.

In order to provide some context to this work, the present introductory chapter explains the main causes of the abovementioned changes, and their impact over power production and consumption patterns. It then details how power grid operation adapts to such systemic changes, and how the recent emergence of Deep Learning and Graph Neural Networks could contribute to reducing emerging security issues.

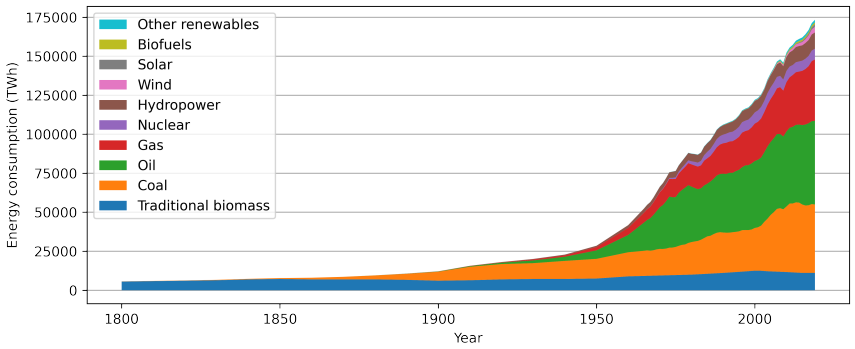
Energy shift

In this section, we provide an overview of the reasons for the current global warming and then detail how policies aimed at mitigating it increase the uncertainty of power injection (*i.e.* power production and consumption patterns).

Global warming

Our daily lives are filled with a wide range of devices that perform meaningful tasks while requiring very little effort from their users.

Those machines are able to convert energy from a primary form into a useful form. For instance, cars transform chemical energy stored in fuel into kinetic energy, *i.e.* into a movement. Energy is available at the Earth's surface in various forms: nuclear (uranium), gravitational (water in mountains), kinetic (wind), etc. Centuries of technological developments have enabled the conversion of energy from one form to the other. Electric power serves as an intermediary form of energy that can easily be transported across large distances through transmission lines. Figure 1 shows how the global production of energy has evolved over the past two centuries.



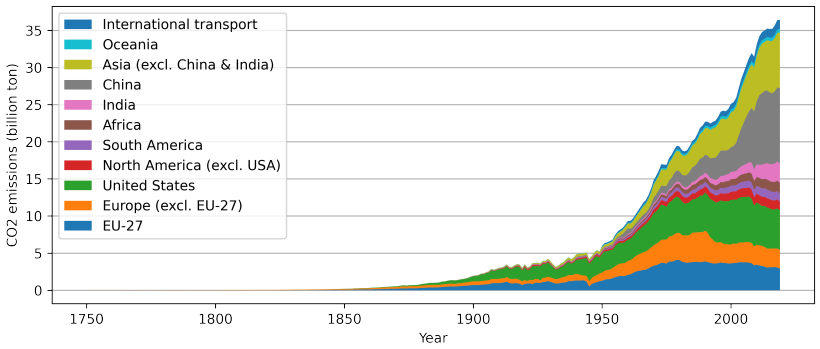
(source ourworldindata.org – last accessed July 2021)

Figure 1: Global energy consumed per year, by type of energy

In the late 2010s, around 85% of the energy produced came from the combustion of fossil fuels that emit greenhouse gas such as (but not limited to) CO_2. Those emissions have been consistently growing since the start of the industrial era, as illustrated by Figure 2.

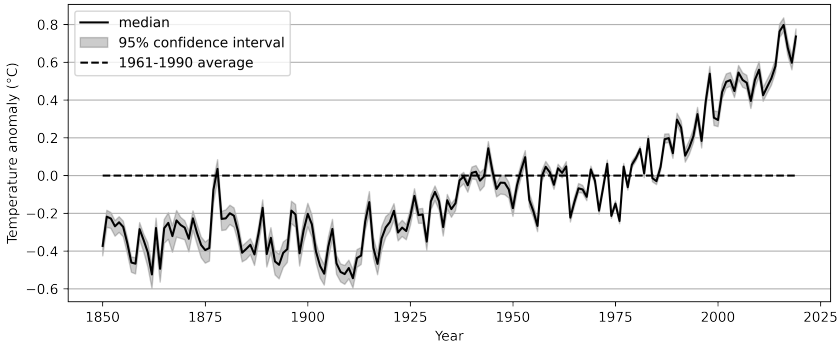
It is nowadays commonly admitted that the negative impact of modern societies on the environment has become non-negligible since the 1950s [4]. The current era is thus referred to as the *anthropocene*. Our understanding of the relationship between temperatures and our greenhouse gas emissions can be traced back to the XIXth century: in 1896, Svante Arrhenius predicted that changes in the concentration of CO_2 in the atmosphere would significantly impact the surface temperature [5]. Empirical evidence of such an anthropogenic global warming have been gathered by Guy Callendar in 1938 [6], and Gilbert Plass formulated the *Carbon Dioxide Theory of Climate Change* [7] in 1956. Figure 3 presents the evolution of the temperature anomaly on Earth compared to the average of the period 1961-1990, thus sketching a worrying trajectory for the upcoming decades.

The brutal change of climate that we will most likely go through



(source ourworldindata.org – last accessed July 2021)

Figure 2: Global CO_2 emissions per year



(source ourworldindata.org – last accessed July 2021)

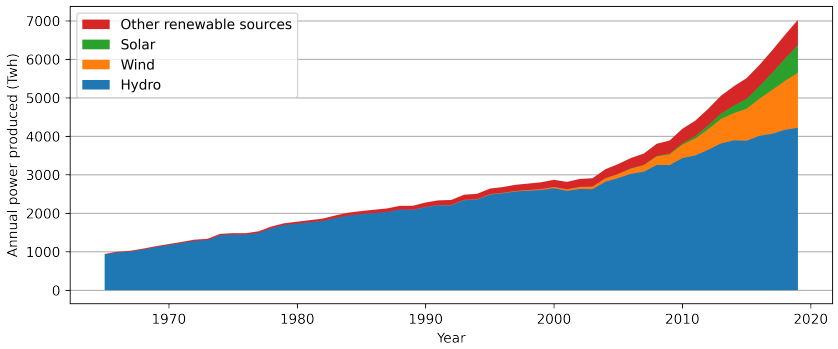
Figure 3: Temperature anomaly compared to the average of the 1961-1990 period

should cause the 6^{th} mass extinction in the History of Life on Earth. In order to prevent the irreversible destruction of an ecosystem which we need for our own survival, it has become urgent to drastically reduce, *inter alia*, the emission of greenhouse gas.

Moving to a new energy mix

Policy makers have been pushing towards the development of alternative energy conversion devices that exploit renewable and low-carbon forms of energy. A source of energy is said to be renewable if exploiting it does not prevent future generations of doing so. Direct radiation of the Sun – which should persist for the next 5×10^9 years – can be exploited using thermal or photovoltaic devices. The wind, which is also

indirectly caused by the Sun's radiation, is another source of energy that can be harnessed by wind turbines. Devices that harness those sources of energy have drastically improved over the past two decades, which has enabled their large scale deployment, as illustrated in Figure 4. A growing amount of research is dedicated to investigating the feasibility of a 100% renewable energy system in the medium term [8, 9], and advocate for a massive use of the latter two technologies.



(source ourworldindata.org – last accessed July 2021)

Figure 4: Renewable energy generation per year, by type

Increasingly uncertain power injection patterns

Unfortunately, solar and wind power come with some drawbacks with regards to their integration in power grids:

- Their production is highly dependent on the weather, which is notoriously known to be hard to predict accurately. This increases the uncertainty of actual production patterns, and may cause an unexpected saturation of some areas of the Power Grid.

- Our energy storage capacity being quite low, the production should always equate to the consumption. As solar and wind power are intrinsically intermittent, it is mandatory to have controllable generation in reserve, so as to compensate for fast variations of renewable generation. For instance, massive investments toward solar energy in California has caused the apparition of the so-called *duck curve*. Since solar panels generation peaks around noon, the need for other sources of energy is also reduced, as illustrated in Figure 5. This causes steep ramps of apparent demand in the morning and in the evening.

- Power generation is predominantly ensured by large rotating machines. The inertia of their rotation provides a fast and accurate way of ensuring the stability of the whole system. Devices such as solar panels do not involve the rotation of any of their parts and cannot take part in this critical stability mechanism. Thus, such devices actually erode the security of the whole system[1].

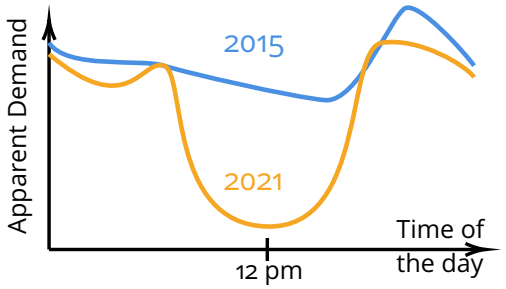


Figure 5: Schematic representation of the Californian duck curve, on typical days of 2015 and 2021.

In addition to challenges posed by renewable energy integration into the grid, two other phenomena cause additional uncertainties, namely the rise of the electric vehicle, and novel European regulations:

- While electric vehicles are still a marginal phenomenon in 2021, RTE projects that there should be around $12,000,000$ electric cars on the French roads by 2035 [11]. Car batteries could be used as an additional flexibility for the Power Grid, but their charging may also create new and unexpected electric consumption patterns.

- Recent European regulations enforce a strict separation of the different parts of the power grid, so as to give rise to the internal energy market. The energy market pushes suppliers to buy electricity at the lowest economic price, regardless of the physical reality of the power grid. Thus, energy can be produced very far from the place of consumption which can cause important power flows across the whole European grid, which can result in congestion issues. In addition, new regulations allow energy producers to change their plans on a very short notice, which causes an additional source of uncertainty for power grid operation.

[1]The *grid forming* domain [10] aims at improving the stability of systems that have a large proportion of renewables.

Changes in power grid operation

The growing uncertainty over power injection patterns is endangering power grids, which pushes their operators to investigate possible solutions. In this section, we first succinctly introduce power grids and how they are operated. Then, we detail some of the currently investigated solutions to improve the system's security with respect to projected trends.

Transmission systems in a nutshell

The electric power grid can be broken down into three main functions, as illustrated in Figure 6: production (power generation, in red), transport (power lines, in blue and orange), and consumption (end users, in green). The transport part is usually split into the "transmission system" (long distances, in blue) and the "distribution system" (local scale, in orange). This PhD thesis is funded by Réseau de Transport de l'Électricité (RTE), the French Transmission System Operator (TSO), which operates the largest European transmission grid ($106,000\ km$ of high voltage and extra high voltage transmission lines). The present document solely considers the French transmission network, although most of the developed ideas and concepts can be easily transposed to other networks and even to other domains.

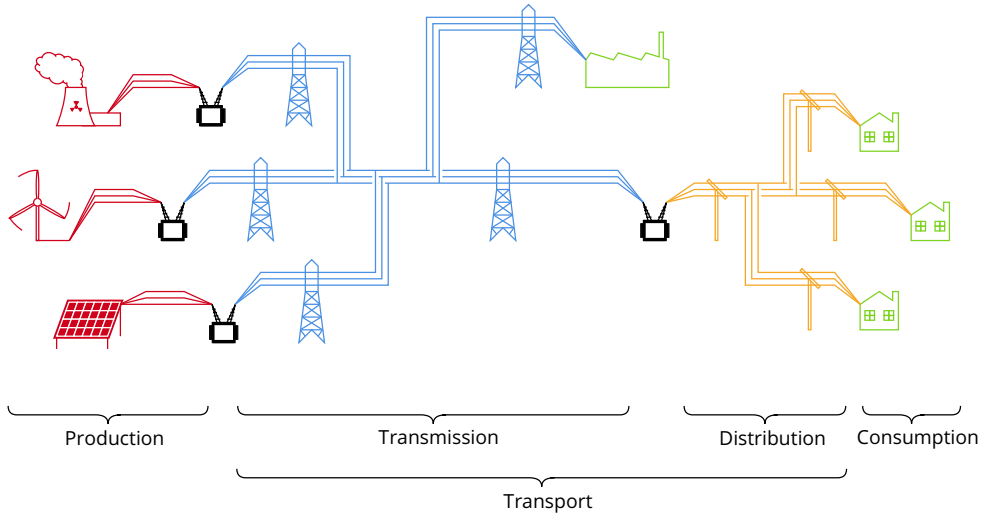


Figure 6: Overall organization of power grids. Power transportation is split into the transmission system (Extra High Voltage and High Voltage), and the distribution system (Medium Voltage and Low Voltage).

RTE is in charge of managing the French transmission system in real time, and ensures that the production equates to the consumption. It anticipates impacts of potential outages, whether these are planned or accidental, and takes appropriate actions. Highly trained engineers – called *dispatchers* – ensure the system's security by monitoring power flows through transmission lines and voltage magnitudes everywhere across the grid.

- A power overflow through a transmission line can cause it to stretch and endanger nearby trees, roads, infrastructures or passers-by. Automatic and decentralized mechanisms can cut open overflowing lines, which can push the overflow to other lines. These newly overflown lines are subsequently disconnected, which then leads other transmission lines to meet the same fate. This type of cascading failure can then quickly lead to a blackout of the whole system if no action is taken. To counter this, dispatchers can change the interconnection patterns of transmission lines, so as to redirect power flows.

- Voltage magnitude should remain in an acceptable range at all times and everywhere. Electric devices are designed to work with a certain voltage amplitude, and can withstand reasonable variations around this value. Thus, straying too far from this nominal value can cause damage to devices, and Transmission System Operators. The main levers of action in this regard include the control of voltage set points of some generators and the activation of shunts.

Those two tasks are actually entangled, but presenting them as being distinct is a good first-order approximation. In both cases, dispatchers have to rely on their thorough understanding of the system. Current optimization-based methods are struggling with the complexity of both problems, and some satisfying heuristics exist or are in the process of being experimented.

Investigated solutions and current limitations

RTE and the Power Systems community as a whole anticipate that the systemic changes evoked in the previous section will bring additional challenges to real-time power systems operation, and investigate various avenues to improve the system's stability and safety.

Several projects aim at expanding the capacity of integration of renewable energies into the actual power grid without building new expensive transmission lines. For instance, the NAZA project (*Nouveaux*

Automates de Zone Adaptatifs, which can be translated as New Zonal Adaptive Automata) prevents the system from saturating in the case of renewable generation peaks by clipping power generation[2]. A companion project named Ringo aims at employing controllable batteries located in three locations far from each other so as to create virtual power lines: energy is withdrawn by batteries in places where congestion might occur, while an equivalent amount of energy is injected in a safer location[3]. These automata make power grids cyber-physical systems where the sole knowledge of physical equations is no longer enough to accurately model reality. Thus, RTE is also pushing in the direction of improving its simulation tools to better take into account the fact that numerous automata take instantaneous decisions on the grid[4].

Another line of work aims at better incorporating uncertainties in power grid operation. The GARPUR consortium [12], in which RTE took part, advocates for a novel reliability management approach that takes into account – among others – uncertainties over production and generation, socioeconomic costs of power supply interruptions and demand side flexibilities. To that end, a probabilistic approach to reliability management has been developed. However, such approaches need to perform numerous expensive simulations, making the whole approach intractable. As a consequence, they advocate for the use of fast proxies based on Machine Learning to quickly estimate the state of power grids, which would allow expensive Monte Carlo simulations.

Current computational methods are unable to advise dispatchers in controlling voltages and power flows, even though this task is projected to become largely more complex in the upcoming decades due to increasingly uncertain injection patterns. Moreover, some methodologies considered to improve power grid security with respect to renewables integration and increasing uncertainties are currently intractable because of the slowness of current methods. Thus, the Power Systems community is investigating methods stemming from other domains, in the hope to find tools that would better suit its current needs.

The Deep Learning opportunity

Resounding successes achieved by the Deep Learning (DL) domain have drawn the attention of the Power Systems community for two

[2]https://www.rte-france.com/actualites/naza-rte-developpe-nouvelle-solution-numerique-pour-renforcer-la-flexibilite-du-reseau

[3]https://www.rte-france.com/projets/stockage-electricite-ringo

[4]https://dynawo.github.io/

main reasons:

- DL methods can deal with extremely complex tasks that require a very high level of abstraction;
- they are fast and parallelizable, because all the computational burden is deferred to a training phase.

In this section we first introduce the domains in which DL is included. Secondly, we review some of its early applications to Power Systems. We then introduce the blooming domain of Graph Neural Networks (GNNs), a subdomain of DL that considers graph data, which will prove to be key in applying DL to Power Systems. Finally, we give some examples of how DL can be enhanced by physical knowledge.

From Artificial Intelligence to Deep Learning

As shown in Figure 7, DL is included in a hierarchy of scientific domains, which can be defined as follows:

- Artificial Intelligence: It can be described as "any system that perceives its environment and takes actions that maximize its chance of achieving its goals" [13]. It was founded as an academic discipline in the 1950s and includes a wide variety of approaches: imitating brain cells, problem solving, formal logic, knowledge databases, etc.
- Machine Learning (ML) / Statistical Learning: Among all the approaches investigated by researchers, Machine Learning [14] has become prominent since the beginning of the XXIst century. It is aimed at developing algorithms that improve by learning from data. It includes a wide variety of algorithms: decision trees [15], k-nearest neighbors [16], linear regression [17], naive bayes [18], support vector machines [19], etc. Some of them have in common that they are trained on a dataset, in the hope to achieve good performance on another dataset stemming from the same distribution. Other approaches that are not discussed in the present document include for instance Reinforcement Learning, which aims at learning by interaction with an environment.
- Representation Learning: This subset of Machine Learning methods is concerned with automatically discovering a data representation that is relevant for the problem at hand. The underlying motivation is that in many real-life problems, data can be very hard to process. For instance in image classification, being able

to classify whether there is a cat or not in an image cannot be performed by a simple linear regression over pixel values. It is mandatory to build a certain level of abstraction and move beyond the raw data. Methods include neural networks, principal component analysis, restricted Boltzmann machines, etc.

- Deep Learning: This part of Representation Learning considers the use of artifical neural networks. It is described by Goodfellow, Bengio & Courville in their *Deep Learning* book [20] as

  > *"... to allow computers to learn from experience and understand the world in terms of a hierarchy of concepts, with each concept defined through its relation to simpler concepts."*

  Depending on the problem at hand, this hierarchy of representations can involve many different abstraction levels, thus making the process of solving it *deep*. This domain dates back to the 1940s, and has known three waves of innovation so far [20]. The current wave started around 2006, and has been enabled by the massive processing power of modern computers, the availability of extremely large datasets, and also by a proactive and creative community of researchers.

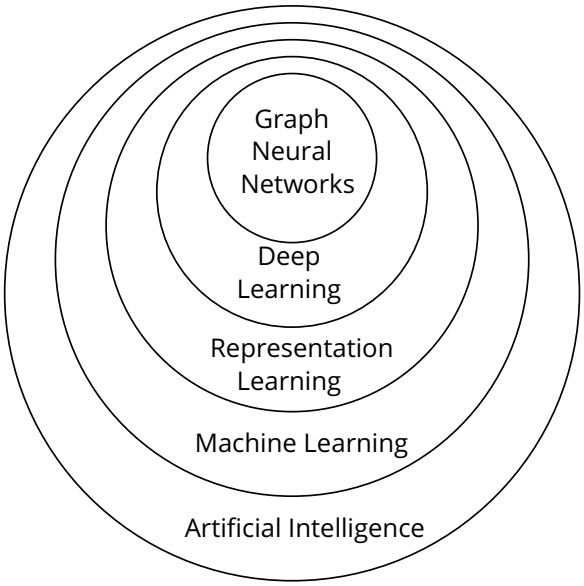


Figure 7: Venn diagram showing the hierarchy of scientific fields in which Deep Learning and Graph Neural Networks fall [20].

Deep Learning has achieved multiple major breakthroughs in a wide variety of domains. In computer vision, it has surpassed human performances in image recognition [21, 22, 23], is able to generate new realistic images [24] and can identify objects in a scene [25]. In natural language processing, it has become the state-of-the-art technique in sentiment analysis [26], information retrieval [27], spoken language understanding [28], machine translation [29], writing style recognition [30], and others. Most commercial voice recognition systems [31] are based on DL. In all of these applications, DL showed its ability to tackle complex problems that require a very high level of abstraction. We refer readers to Chapter 2 for a succinct introduction to DL techniques.

Applications of Deep Learning to Power Systems

As early as in the 90s, seminal work [32] started applying ideas from ML and DL to issues related to Power Systems operation. A review paper by Duchesne et al. [33] provides a thorough overview of various applications of ML, and in particular of DL methods to power grid static reliability management. In what follows, we review several applications of DL to the AC Power Flow (AC-PF) and AC Optimal Power Flow (AC-OPF) problems. Other possible applications are being investigated by RTE, such as the use of Reinforcement Learning [34], although this falls out of the scope of this PhD thesis.

The AC Power Flow (AC-PF) problem can be framed as follows: knowing power production and consumption, and the way power lines are interconnected to each other, the goal is to compute the power flow through lines. In [35], Schaefer et al. investigate the use of various ML methods to solve the AC-PF, and show that deep neural networks bring the best performances, a line of work that has been consistently growing over the past few years [36, 37, 38, 39]. Neural networks can also be used to warm start a traditional optimization method [40], or to detect if a grid is in security or not [41, 42].

On the other hand, the AC Optimal Power Flow (AC-OPF) problem is a non-linear and non-convex optimization problem in which a cost function should be minimized while respecting physical and operational constraints. Deep Learning has already been extensively applied to this issue so as to directly predict optimal control variables [43, 44, 45]. Neural networks have been successfully used to warm start traditional optimization techniques [46, 47, 48], perform fast screenings of situations [49, 50, 51, 52], and reduce the computational burden of the AC-OPF by predicting the set of active constraints [53, 54, 55, 56]

Most methods surveyed above assume a fixed topology of the grid,

i.e. the graph structure does not change. However, the actual grid topology of power grids changes several times per day under the action of dispatchers. This raises the following questions:

> *How can we learn from data that have a changing underlying structure?*

During his PhD thesis, Benjamin Donnot began investigating such a critical issue by developing the Latent Encoding of Atypical Perturbations (LEAP) network [57], also referred to as *guided-dropout*. This DL architecture is able to conditionally activate or deactivate sets of neurons depending on the situation. He experimentally showed that this architecture had good generalization properties, even to grid topologies that were never encountered during training. However, this approach only allows a limited amount of perturbations to be considered, and does not completely take into account the fundamental invariant of graph data: *permutations*. A simple node reordering of the input data shatters the predictive power of the trained neural network. However, a class of neural networks called Graph Neural Networks (GNNs) allow to make a conceptual and experimental leap towards addressing such issues, as introduced thereafter and further detailed in Chapter 3.

Graph Neural Networks

Power grids have a graph structure which cannot be processed properly by traditional neural networks. Thankfully, the blooming domain of GNNs provides us with a class of neural network architectures that are purposefully designed to handle graphs. They can intrinsically withstand any node reordering of their input graph, by directly encoding the input graph structure into the neural network architecture. As explained by Battaglia et al. [58], they use traditional neural networks as elementary trainable blocks entangled in a much larger architecture that inherently respects the graph structure of its input.

Although this domain has only recently started to become a major area of research (2017-2018), early work by Sperdutti et al. [59] on applying neural networks to acyclic graphs can be traced back to 1997. Then in 2005, Gori et al. [60] first introduced the notion of GNN, although it does not quite resemble the approach that currently bears this name. This line of research was then continued in the late 2000s by Scarselli et al. [61] and Gallichio et al. [62]. Motivated by the accomplishments of Convolutional Neural Networks (CNNs) on images, research focused on trying to extend the notion of convolution to graphs, which lead to the emergence of two distinct approaches.

The first one is called Spectral GNN [63] and relies on spectral graph theory. It decomposes input graphs according to a spectral basis which is then fed to a neural network architecture. While this approach has been applied with success on various problems [64, 65, 66, 67], it suffers from a poor generalization capability: as soon as the graph structure is slightly altered, the spectral decomposition can change drastically and the trained neural network model stops being relevant.

The second one is called Spatial GNN [68] and relies on local message passing operations. No spectral decomposition is required, and a trained neural network easily generalizes to various graph structures. Although the approach fell into oblivion for almost a decade, it has recently emerged again [69, 70, 71] and has then become the prominent approach in many domains.

GNNs have been applied to various non-Euclidean data. In computer vision, they are used to generate semantic graphs that explain relations between objects in a scene [72, 73, 74], or to generate a realistic scene knowing a semantic graph [75]. They are also applied to human joint detection [76, 77], human-object classification [78, 79] and visual reasoning [80]. In chemistry, the 3D structure of a molecule being a graph, GNNs can predict their fingerprints [81, 82] and properties [71], proteins interfaces [83] and be used to synthesize organic compounds [84, 85, 86].

Ideas and methods involved in GNNs are introduced and succinctly explained in Chapter 3. However, since this work solely focuses on Spatial GNNs, we refer interested readers to the review paper written by Wu et al. [87] and to the book *Graph Representation Learning* by William L. Hamilton [88] for a more exhaustive presentation of the domain.

Introducing GNNs as a tool to accelerate power grid related computations is one first contribution of this thesis. Another one is described in the next section, and relates to the methodology employed to train such GNNs.

Merging Deep Learning and physics

In the past few years, the amount of publications dedicated to the application of DL to physics-oriented problems has consistently grown. In most cases, the goal is to accelerate potentially expensive simulations using fast neural networks. Two main approaches can be distinguished: The supervised “proxy” approach, which consists in imitating the output of a classical physical simulator, and the semi-supervised or unsupervised approach, which aims at incorporating physical laws directly into the neural network architecture or into the training loss.

The "proxy" approach The former approach, which we refer to as the "proxy" approach consists in imitating potentially expensive physics simulators, generally with the aim to find a more suitable balance between computational speed and accuracy. For instance, Eulerian fluid simulations were successfully accelerated by replacing a computational block by a neural network approximation [89]. Other applications include the acceleration of various scientific computations using convolutional neural networks [90, 91], and the modelling of high-energy particle physics [92]. As evoked above, early applications of DL to Power Systems [38, 32] also fall into this line of work.

The "proxy" approach can be enhanced by Graph Neural Networks: some physics problems are defined on well-structured systems which are explicitly modelled as graphs. Including the structure directly into the neural network architecture can improve both the performances and generalization capabilities of trained models [93, 94].

Incorporating Physical Knowledge Unlike many applications encountered in Machine Learning, problems stemming from physics are usually well described by a series of equations. Physics-Informed Neural Networks (PINNs), introduced by Raissi et al. in 2019 [95] propose to train a model using a combination of classical data and of the knowledge of physical equations. Such an approach provides a semi supervised setting and allows for the resolution of both direct and inverse problems. Applications include for instance fluid mechanics [96]. In the present PhD thesis, we propose to go even further by considering a fully unsupervised approach where only the violation of physical equations is penalized during training.

Main contributions

Applying methods from the DL literature to Power Systems presents major challenges to inspire confidence to the Power Systems community and meet necessary criteria of testability and reliability, particularly because the theory underlying DL methods is still in its infancy. This thesis proposes new problem formulations and DL solutions, as a step towards the adoption of DL methods in this application context.

This thesis includes the following main technical contributions:

- We provided the first application of GNNs to power grids [1], experimentally proving the ability of such neural network architectures to withstand variations in the amount of nodes, lines, and

interconnection patterns, and experimentally proving the viability of the approach.

- We were the first to train GNNs to solve instances of an optimization problem by direct minimization of physical laws, instead of relying on the imitation of classical optimization methods [2].

- We introduced a Universal Approximation Theorem which states that GNNs architectures are suitable to solve optimization problems [3].

- We developed a Hyper Heterogeneous Multi Graph (H2MG) formalism that models power grids more naturally, and a matching Hyper Heterogeneous Multi Graph Neural Network (H2MGNN) architecture to process such data structures.

We call our approach the Deep Statistical Solver (DSS). It consists in training permutation-equivariant GNNs to solve instances of optimization problems in an unsupervised fashion, *i.e.* without imitating the output of a traditional optimization method. As a consequence, our methodology is an optimization technique on its own. Ongoing work include the development of a bilevel H2MGNN approach to solve bilevel optimization problems.

Concurrent work

Since the beginning of this work, several papers have been published that exploit similar ideas to the ones presented in this document. After our paper *Graph neural solver for power systems* [1], the use of GNNs to power grids has quickly become commonplace [97, 55, 98, 99], and the idea to use of GNNs to perform scientific computation has been successfully applied to fluid dynamics simulations [100]. After our paper *Neural networks for power flow: Graph neural solver* [2], GNNs and physical knowledge were similarly combined on power grid problems [101, 102].

PhD thesis outline

This document aims at being self-contained and providing the right level of details to enable any interested reader with a basic background in mathematics and physics to grasp the main ideas. This contribution

being a bridge between two scientific communities, each domain is introduced, and references to the relevant literature is provided. Readers from the Power Systems domain are encouraged to skip Chapter 1, while readers familiar with the Deep Learning literature should skip Chapter 2.

Part I - Background & motivations The first part introduces the domains within which this work falls.

- Chapter 1 - Power Systems: introduces power systems, starting from basic physical phenomenon, sweeping through the electrotechnical modelling of the main components of power grids, and further detailing some key aspects of how they are operated.
- Chapter 2 - Deep Learning: introduces key statistical learning concepts, deep learning basics, and some major ideas at the core of the success of CNNs.
- Chapter 3 - Graph Neural Networks: introduces graph data, and how GNNs manage to accurately process them.

Part II - Deep Statistical Solvers This second part details the core contribution of the present PhD thesis, which includes both a class of problems and a proposed resolution method.

- Chapter 4 - Deep Statistical Solver Architecture: discusses some key features of real world data, and proposes a suitable H2MGNN architecture.
- Chapter 5 - Statistical Solver Problems: discusses the conversion of optimization problems into statistical learning problems, and how training can be performed without imitating the output of a traditional optimization method.
- Chapter 6 - Universal Approximation Theorem: discusses some key properties of the proposed architecture in terms of expressivity, and introduces our extension of the Universal Approximation Theorem to the considered class of GNN architectures.

Part III - Applications This third part is devoted to the application of our proposed Deep Statistical Solver approach to a series of optimization problems.

- Chapter 7 - Toy Examples: applies our proposed Deep Statistical Solver approach to a series of toy problems.
- Chapter 8 - AC Power Flow: applies our method to the non-linear problem of estimating the flows across a power grid knowing power injections and the actual power grid topology.

Part IV - Conclusion & Future Work This final part concludes and explores some questions that were opened by this work

- Chapter 9 - Discussion & Future Work: discusses ongoing work on voltage control, and potential extensions of the DSS to problem that include time and uncertainty.
- Chapter 10 - Conclusion: summarizes main contributions and observations of this PhD thesis.

Part I

Background

Chapter 1

Power Systems

This PhD thesis aims at developing DL techniques to solve optimization problems defined over power grids. In order to understand the basic principles of the latter, and how their components behave and interact, this chapter introduces Power Systems (PS) fundamentals. Firstly, it defines power grids as a network of interacting dipoles and quadrupoles. Then it details how Alternating Current (AC) systems rely on oscillations of electrons to transport electrical power across large distances. Finally, it dives into the physical modelling of the main devices that are considered in this PhD thesis.

We refer interested readers to the book *Power System Stability and Control* by P. Kundur [103] for more details about power grids.

1.1 Transporting power using electricity

Modern power grids are predominantly in Alternating Current: electrical power is transported by oscillations of electrons. These oscillations are driven by generators, and slowed down by consumers. In this section, we give some insights about the fundamental mechanisms at work in power systems. After having introduced basic principles of electricity, we explain the behavior of dipoles and quadrupoles, and how they can be interconnected into a network called a power grid.

1.1.1 Electricity

The term electricity denotes the phenomenon induced by the motion of electrons in a conductive material. In such materials (mostly metals), charged particles – called electrons – are loosely attached to atoms and are free to move. Those electrons can be set in motion by devices that exploit various forms of energy (chemical, thermal, nuclear,

kinetic, etc.). Conversely, other devices can convert the motion of electrons into another form of energy (lighting, heating, etc.). The motion of electrons is not useful by itself, and only serves as an intermediary. Electrical power can be transported through conductive cables, which allows to generate energy far from where it is consumed.

1.1.2 Dipoles

A dipole is a electric device that has one port, which is made of two terminals: $+$ and $-$. Electrons can either enter the dipole through the $+$ terminal and get out through the $-$ terminal, or the opposite. As shown in the left part of Figure 1.1, we denote by u in Volt (V) the voltage drop – which measures the work applied by the dipole over an elementary charged particle – between the $+$ and $-$ terminals, and by i in Ampere (A) the current of electrons flowing into the $+$ terminal (which necessarily equates to the current flowing from the $-$ terminal). In AC systems, electrons oscillate around a fixed position, thus they alternately enter and exit the dipole through each port. The instantaneous power p in Watt (W) injected by the dipole through its port is given by:

$$p = ui \tag{1.1}$$

Generators are dipoles that inject power into the system ($p > 0$) and loads (consumers) are dipoles that withdraw power ($p < 0$).

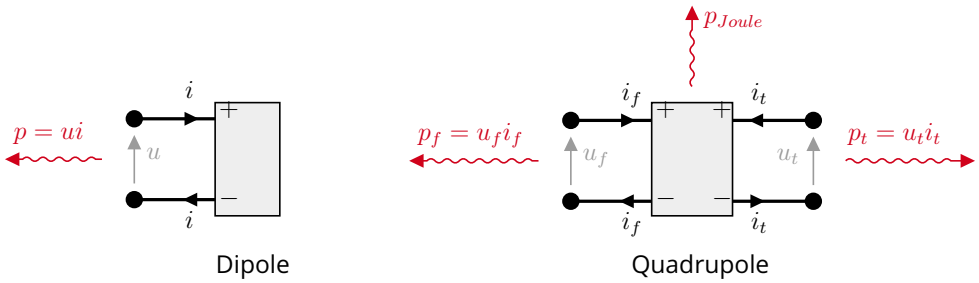


Figure 1.1: Schematic representation of a dipole (left) and of a quadrupole (right).

1.1.3 Quadrupoles

In power grids, electric power is transported across long distances using cables of conductive materials known as power lines. In order to allow electrons to flow in a closed circuit, transmission lines are made

of two cables, each allowing electrons to flow in opposite directions. This pair of cables belongs to the category of *quadrupoles*, which are electric devices that have two ports. As shown in the right part of Figure 1.1, we index by f quantities defined at the "from" port, and by t quantities defined at the "to" port. Power is injected into one port and retrieved from the other. However, friction between the flow of electrons and atoms of the transmission line cause some power to be lost due to Joule's effect:

$$p_f + p_t + p^{Joule} = 0 \tag{1.2}$$

The actual equation for Joule's effect in AC power grids is deferred to the last section in equation (1.23). However, it can be approximated by $p^{Joule} \approx ri^2$ where r is the resistance of the cable in Ohm (Ω), and $i \approx i_f \approx i_t$. Joule's effect can be seen as a mandatory tax for the transport of energy. The resistance of a line being proportional to its length, the more production and consumption are spread apart, the more power will be lost.

1.1.4 Power grids

Dipoles and quadrupoles can be interconnected together into a network called a power grid. The purpose of this power grid is to transport electrical power from dipoles that produce it to dipoles that consume it. Ports of multiple devices can be connected together to form a "bus": their respective $+$ and $-$ terminals are connected together, as illustrated in Figure 1.2. Kirchhoff's laws govern the behavior of interconnected devices:

- Kirchhoff's current law: the algebraic sum of currents flowing into collocated ports is zero. In Figure 1.2 we obtain $i_g + i_f = 0$ and $i_t + i_l + i_{l'} = 0$.
- Kirchhoff's voltage law: all ports connected to the same bus share the same voltage. In Figure 1.2 we obtain $u_1 = u_g = u_f$ and $u_2 = u_t = u_l = u_{l'}$.

Power grids can be represented in two distinct ways. The "electric diagram" displays the detailed electric circuit, showing both terminals $+$ and $-$. It also clarifies the fact that electrons flow in a closed loop of conductive material. On the other hand, the "single-line" diagram proposes a simplified visualization where both terminals are merged. It allows for a clearer representation of power flows.

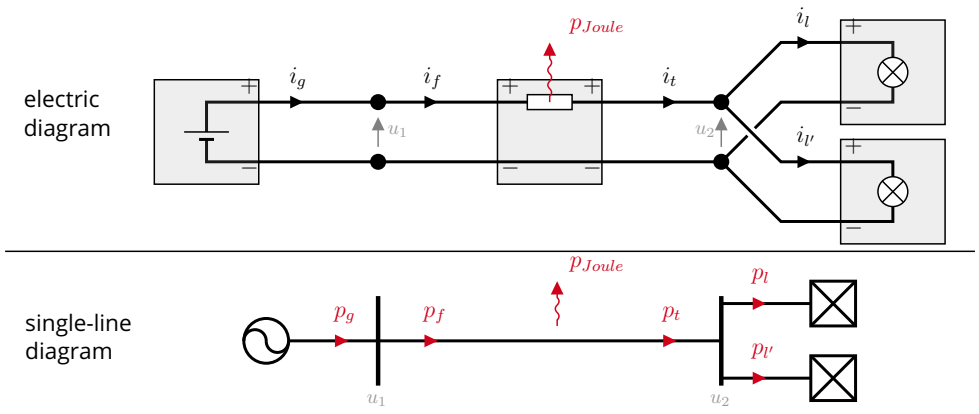


Figure 1.2: Two representations of the same power grid. It is made of one generator (indexed by g), two loads (l and l'), and one transmission line (whose ends are indexed by f and t). Devices are connected together via buses. The single-line diagram does not represent the $+$ and $-$ nodes of each bus, and represents buses as perpendicular lines.

Power grids are networks of interconnected dipoles and quadrupoles that allow electrons to flow in a closed loop of conductive material. However, electrons do not continuously flow around those loops: they actually oscillate around a fixed position. Electrical power is thus transported by oscillations of those electrons, just like sound is transported by oscillations of air particles.

1.2 AC Power Systems

As stated before, modern power systems rely on the use of Alternating Current (AC) which allows to transport electrical power without too much loss. This section introduces some basic ideas behind the design of AC systems, and how they have been critical to the electrification of societies. Then it provides explanations about the notion of "apparent power" that arises from the oscillation of both voltages and currents.

1.2.1 Different voltage levels

Joule's effect can be mitigated by increasing the voltage (and subsequently decreasing the current). However, in the beginning of commercial power grids (1870s-1880s), there was no technological means to change the voltage level. A significant part of the energy produced was lost due to Joule's effect, which prevented electricity from being

transported too far from generation [104, 105]. Power Systems were very decentralized, with many small generation facilities located near places of consumption. In addition, not all electric devices required the same voltage orders of magnitude: there were several power grids in parallel, each having their own voltage level. In the late XIXth century however, the ability to scale up or down the voltage level was achieved thanks to the combined use of AC and of passive devices called transformers.

In AC power systems, the force applied by dipoles over electrons is constantly oscillating. All electrons oscillate around a fixed position and transmit their motion by pushing and pulling their neighbors. Transmission lines act as an oscillation coupling mechanism that transports energy through oscillations.

Voltage and current oscillations of two disconnected AC circuits can be coupled using transformers, as illustrated in Figure 1.3. Such devices consist in two windings wrapped around a high magnetic permeability material: oscillations in one winding create an electromagnetic flux through the core which then causes oscillations in the other winding. While electrical power injected into an end is mostly retrieved at the other end, the ratio of voltage magnitudes depends on the ratio of turns in both winding, as shown in Figure 1.3.

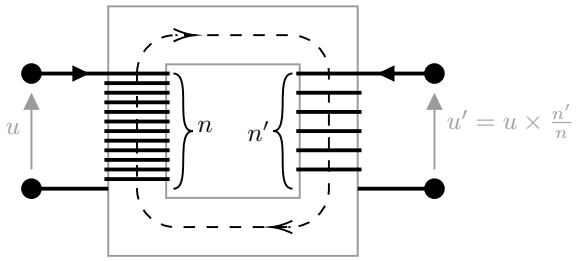


Figure 1.3: Schematic representation of a transformer. The ratio between voltage u and voltage u' is imposed by the ratio of turns between both winding.

Power grids are thus made of a series of disconnected circuits whose oscillations are coupled by transformers. Transformers can scale up voltage amplitudes near places of generation, and scale them down near places of consumption, so as to reduce Joule's effect in between[1]. Moreover, devices that require different voltage levels can now

[1]See Figure 1.2, power injected by the generator is $p_g = i_f u_1$, and the power losses due to Joule's effect is $p^{Joule} \approx r i_f^2$ if the transmission line is approximated as a resistance r; by combining both equations we obtain that $p^{Joule} \approx r p_g^2 / u_1^2$. Thus, increasing the voltage u_1 can decrease power losses.

be connected to the same grid. Overall, AC power allows for a drastic simplification of Power grids for both consumers and producers, and quickly became the predominant model for electric power transport.

However, high voltage levels cause strong electric fields that may trigger electric arcs and fires. They require larger and more expensive infrastructures to spread power lines apart, and to lift them high enough. Modern power systems are thus designed as a trade-off between Joule's effect losses and higher costs of high voltage infrastructures. In France, they are composed of the following voltage levels, which can be split into two distinct categories (recall Figure 6):

- The transmission system made of High Voltage ($63kV$ and $90kV$) and Extra High Voltage ($225kV$ and $400kV$) power lines, and operated by a Transmission System Operator (TSO);
- The distribution system, made of Medium Voltage ($20kV$) and Low Voltage ($230V$ and $400V$), and operated by a Distribution System Operator (DSO).

In this PhD thesis, we only consider the transmission system.

1.2.2 Oscillations and apparent power

The present PhD thesis is notably interested in using DL techniques to find equilibrium states of power grids, which are dictated by Kirchhoff's laws. However, their formulation in AC is different from what has been previously introduced, and is central to experiments conducted in Chapter 8. In order for readers not familiar with the PS literature to understand Kirchhoff's laws AC formulation, the main underlying concepts and intermediate steps are detailed in the following.

In AC power grids, voltages and currents oscillate everywhere at the same frequency $\varpi/2\pi$ (in Hz). Every bus has its own phase ϑ: some may be delayed while others may be ahead of phase. The instantaneous voltage of a bus is written as:

$$u_{inst}(t) = \Re(\sqrt{2}\, u e^{j\varpi t}) \qquad \text{with } u := \frac{u_{max}}{\sqrt{2}} e^{j\vartheta} \tag{1.3}$$

where $j = \sqrt{-1}$, and $\Re(x)$ denotes the real part of complex number x. The current flowing from a bus into a port is delayed by a phase shift ϕ compared to the voltage. Thus, the instantaneous current intensity flowing into a port is written as:

$$i_{inst}(t) = \Re(\sqrt{2}\, i e^{j\varpi t}) \qquad \text{with } i := \frac{i_{max}}{\sqrt{2}} e^{j(\vartheta-\phi)} \tag{1.4}$$

The instantaneous power injected from bus into a device can be written as:

$$p_{inst}(t) = u_{inst}(t) \times i_{inst}(t) \tag{1.5}$$

$$= \frac{u_{max} i_{max}}{2} \cos(\phi)(1 - \cos(2\varpi t + 2\vartheta)) + \frac{u_{max} i_{max}}{2} \sin(\phi) \sin(2\varpi t + 2\vartheta) \tag{1.6}$$

It is constantly oscillating at a frequency of ϖ/π and with a phase angle 2ϑ, as illustrated in Figure 1.4. It can be decomposed into the two terms of equation (1.6).

- The first term has constant sign (that depends on $\cos(\phi)$). Its mean value $p := u_{max} i_{max} \cos(\phi)/2$ is called *active power*. This is the *useful* part of the power, *i.e.* the part that is actually transmitted to the device.
- The second term has a zero mean. Its amplitude $q := u_{max} i_{max} \sin(\phi)/2$ is called the *reactive power*. It bounces on the dipole and is not injected or withdrawn by it. Although it is common to envision this as the *useless* part of the power, it is nevertheless an unavoidable part of the coupling of oscillating systems.

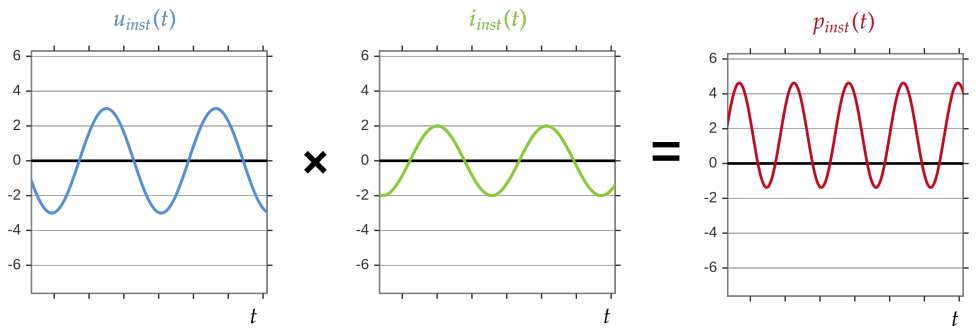


Figure 1.4: Instantaneous power of a device in an AC system. As the voltage and current intensity are slightly delayed (by a phase shift ϕ), their product is not necessarily centered around zero.

To represent both active and reactive parts of the power at the same time, it is common to introduce the notion of apparent power $s \in \mathbb{C}$ [103]:

$$s := p + jq \tag{1.7}$$

$$= ui^* \tag{1.8}$$

where i^* is the complex conjugate of i. The apparent power does not depend on the bus phase angle ϑ. To emphasize the distinction between s, p and q, the PS community measures the apparent power s in Volt-Ampere (VA), the active power p in Watt (W), and the reactive power q in Volt-Ampere reactive (VAr).

Introducing the notion of apparent power s allows to represent in a single variable both components of the oscillating power. Moreover, Kirchhoff's laws in AC systems can be rephrased in terms of the sum of apparent powers injected each bus, as follows:

$$\sum_k s_{k \to l} = 0 \tag{1.9}$$

where l denotes a bus, and the sum is over all devices k that are connected to bus l.

1.2.3 Three-phase power grids

So far we have only considered single-phase power grids. Actually, real-life power grids are three-phase systems: power is transported by 3 distinct cables, each bearing current and voltage oscillating at the same frequency, and delayed of one third of a cycle between each other. This technology is more economical than single-phase systems that are presented in this document, and allows to transport the same amount of power using a smaller amounts of conductive material. However, it is common to convert the actual three-phase system into a single-phase equivalent system, and use exclusively the latter. As a consequence, all physical models considered in this document make no mention of three-phase systems, and consider the equivalent single-phase system.

1.3 Power grid modelling

Power grids are networks of interacting dipoles and quadrupoles, designed to transport electrical power from producers to consumers. A typical instance of such a grid is displayed in Figure 1.5: the "IEEE case14" power grid, which is an approximation of the American electric power system as of February 1962. It is made of 14 buses, 5 generators, 11 loads, 17 transmission lines, and 3 transformers.

Objects that compose power grids can be split into several categories, each having a distinct behavior in terms of injected apparent power. In this section, we propose to review all classes of objects by detailing their respective set of features, as well as physical and decisional

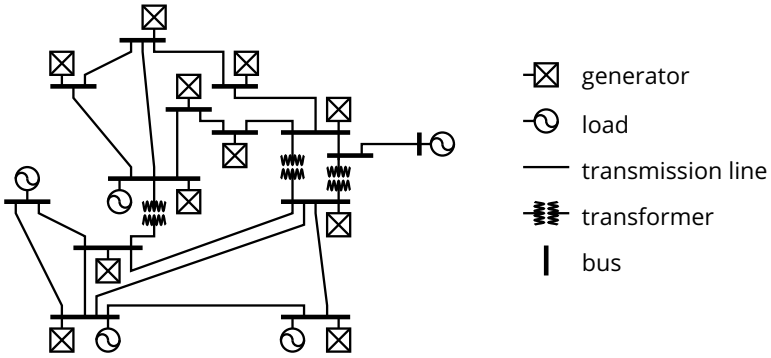


Figure 1.5: IEEE case14 – Instance of a 14 buses power grid.

modelling. We consider both active and reactive parts of steady-state power grids. Finally, we summarize all relevant quantities in Table 1.1, and explain how some of them are linked through optimization problems. This last point is at the heart of the present PhD thesis.

1.3.1 Buses

Buses lie at the interface between interconnected devices. They are usually modelled by the following set of features $(\overline{v}, \underline{v}, \mathring{v}, \mathbb{1}^{pv}, \mathbb{1}^{slack}, \hat{v}, \hat{\vartheta})$. All variables are successively introduced and put in context in the following.

The actual complex voltage of a bus is written as:

$$u := ve^{j\vartheta} \tag{1.10}$$

Relationships between v and ϑ and the above features are detailed below and depend on the role of the bus in the grid.

As mentioned in equation (1.9), Kirchhoff's laws state that the algebraic sum of apparent powers flowing into a bus should sum to zero. One may estimate the discrepancy with regards to physical laws by computing the squared norm of the apparent power mismatch at each bus. The discrepancy can be decomposed into an active and a reactive term:

$$|\Delta s|^2 = |\Delta p|^2 + |\Delta q|^2 \tag{1.11}$$

Equations for Δp and Δq depend on both the bus and the devices that are connected to it, as detailed in the following.

Both the complex voltage and apparent power mismatches actually depend on mechanisms that ensure the security and stability of the system: frequency and voltage regulation. We propose to succinctly introduce the purpose and means of both processes, and then explain how they are modelled.

Frequency regulation

This process aims at ensuring the stability of the oscillation frequency in the whole system. By injecting power into the grid, generators tend to accelerate the oscillation of electrons. Meanwhile, consumers tend to slow these oscillations by withdrawing power. Thus, if production is not equal to consumption plus losses, the frequency may vary. In such a case, dispatchers may resort to disconnecting devices from the grid or even to load shedding (*i.e.* disconnecting consumers).

Thus, frequency regulation aims at keeping oscillations close to a fixed frequency ($50Hz$ in Europe), by modulating the amount of active power injected by generators. This process is decentralized and involves multiple sub-mechanisms which operate at different time scales. Many models exist, but all are an over-simplification of the actual process.

We propose to choose the simplest model available. It defers to a single bus – the "slack" bus, identified by the boolean feature $\mathbb{1}^{slack}$ – the task of providing enough power to ensure a global equilibrium of the system. Numerically, it is equivalent to alleviating the active power mismatch objective on this bus. For all buses, the active mismatch is given by:

$$\Delta p = (1 - \mathbb{1}^{slack}) \sum_{k} p_{k \to l} \tag{1.12}$$

In addition, it is common in the PS literature to set this bus to have a zero phase angle (all phase angles are defined up to translation). For all buses, the phase angle is given by:

$$\vartheta = (1 - \mathbb{1}^{slack}) \times \hat{\vartheta} \tag{1.13}$$

Thus, ϑ takes the value of the feature $\hat{\vartheta}$ only if the bus is not slack, and zero otherwise.

Voltage regulation

The second mechanism involved aims at ensuring that the voltage magnitude at all buses remains within an acceptable range of values: $[\underline{v}, \overline{v}]$. High voltage magnitudes may endanger devices connected to it, and low voltage magnitudes can affect the quality of electricity provided to consumers, or even lead to a blackout caused by a voltage collapse.

Generators located at so-called "PV" buses – identified by the boolean feature $\mathbb{1}^{pv}$ – can modulate their reactive production so as to

ensure that the local voltage magnitude is exactly at a certain set point $\mathring{v}$. As a consequence, the reactive mismatch at PV buses is necessarily zero, while the voltage magnitude is forced to be at $\mathring{v}$. For all buses, the reactive mismatch and voltage magnitude is given by the following equations:

$$\Delta q = (1 - \mathbb{1}^{pv}) \sum_k q_{k \to l} \tag{1.14}$$

$$v = \mathbb{1}^{pv} \times \mathring{v} + (1 - \mathbb{1}^{pv}) \times \hat{v} \tag{1.15}$$

We use different variables for $\mathring{v}$ and $\hat{v}$ to reflect the fact that the former is controlled by the dispatcher, while the second is a consequence of physical equations.

Dispatchers thus monitor voltage amplitude at all buses, even those which are not PV, and ensure that all magnitudes are within acceptable values, which is quantified by:

$$\Delta v = \max(0, |u| - \overline{v}) + \max(0, \underline{v} - |u|) \tag{1.16}$$

1.3.2 Loads

Loads (or consumers) are devices that withdraw power from the grid. They are defined by their active power $\mathring{p}$, and reactive power $\mathring{q}$. It is assumed that they cannot be managed in any way, and withdraw the following apparent power:

$$s^{load} = \mathring{p} + j\mathring{q} \tag{1.17}$$

1.3.3 Generators

As previously mentioned, generators contribute to both the frequency and voltage regulation mechanisms. They are defined by their active power $\mathring{p}$, and reactive power $\mathring{q}$. Furthermore, generators may take part in frequency and/or voltage regulations, which modifies respectively their active and reactive productions compared to the target values. Still, the potential additional apparent power at PV and slack buses has already been modelled by equations (1.12) and (1.14), and we assume that they have no active and reactive limits. Thus, they withdraw the following constant apparent power:

$$s^{gen} = \mathring{p} + j\mathring{q} \tag{1.18}$$

1.3.4 Shunts

A shunt is a device that has a fixed impedance, such as a capacitor or an inductor. It is used as a mean to control the voltage magnitude: dispatchers can disconnect or reconnect shunts so as to modulate the reactive power, which indirectly impacts voltage. It is defined by their conductance g and susceptance b. The apparent power injected by the shunt is given by:

$$s^{shunt} = (g - jb)|u|^2 \tag{1.19}$$

1.3.5 Transmission lines

Transmission lines are in charge of transporting electric power over long distances. They act as a coupling mechanism between oscillations of their "from" and "to" ends (respectively indexed by f and t). They are modelled by the electric diagram of Figure 1.6 and are defined by their resistance r, their reactance x and their total line charging susceptance b^c.

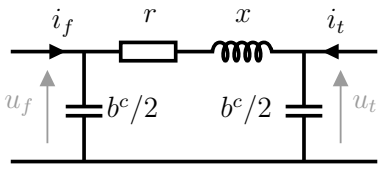


Figure 1.6: Electric diagram of a transmission line.

One may define the admittance matrix of a transmission line as:

$$Y = \begin{bmatrix} y_s + j\frac{b^c}{2} & -y_s \\ -y_s & y_s + j\frac{b^c}{2} \end{bmatrix} \qquad \text{with } y_s = \frac{1}{r + jx} \tag{1.20}$$

Transmission lines impose the following relationship between the complex voltages and currents of their ends:

$$\begin{bmatrix} i_f \\ i_t \end{bmatrix} = Y \begin{bmatrix} u_f \\ u_t \end{bmatrix} \tag{1.21}$$

Recalling the definition of apparent power of equation (1.8), we obtain the flow s_f injected into the bus "from" and the flow s_t injected into the "to" bus.

$$s_f = u_f i_f^* \qquad\qquad s_t = u_t i_t^* \tag{1.22}$$

Both apparent powers depend on complex voltages of both buses, thus inducing a coupling between buses.

Losses caused by Joule's effect is the amount of active power that is injected on one side but not retrieved on the other:

$$p^{Joule} = |\Re(s_f + s_t)| \tag{1.23}$$

1.3.6 Transformers

From an electrical point of view, transformers are akin to transmission lines: they are a coupling mechanism between oscillations of two buses. The main distinction is that they can scale up or down the voltage magnitude between their "from" and "to" ends. In addition, some transformers may induce an additional phase shift between both ends (phase shifting transformers). They are modelled by the electrical diagram shown in Figure 1.7, and are defined by their resistance r, their reactance x, their total line charging susceptance b^c, their ratio τ and their phase-shift angle ϑ^{shift}.

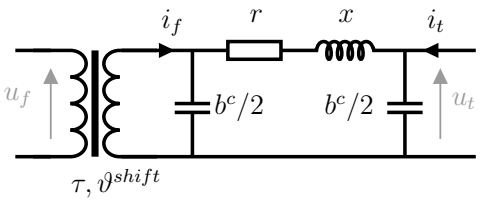


Figure 1.7: Electric diagram of a transformer

The admittance matrix of a transformer is defined as:

$$Y = \begin{bmatrix} (y_s + j\frac{b^c}{2})\frac{1}{\tau^2} & -y_s\frac{1}{\tau e^{-j\vartheta^{shift}}} \\ -y_s\frac{1}{\tau e^{j\vartheta^{shift}}} & y_s + j\frac{b^c}{2} \end{bmatrix} \qquad \text{with } y_s = \frac{1}{r + jx} \tag{1.24}$$

Transformers impose the following relationship between the complex voltages and currents of their ends:

$$\begin{bmatrix} i_f \\ i_t \end{bmatrix} = Y \begin{bmatrix} u_f \\ u_t \end{bmatrix} \tag{1.25}$$

We obtain the flow s_f injected into the bus "from" and the flow s_t injected into the "to" bus.

$$s_f = u_f i_f^* \qquad\qquad s_t = u_t i_t^* \tag{1.26}$$

Once again, the injected powers depend on the complex voltage of both ends, which induces a coupling between buses.

Transformers are also prone to Joule's effect:

$$p^{Joule} = |\Re(s_f + s_t)| \tag{1.27}$$

1.3.7 Summary of features and metrics

So far we have assumed no relationship between features. Actually, they are not independent: they are linked together through optimization problems, which involve the minimization of some metrics.

- For instance, buses features $\hat{\vartheta}$ and $\hat{v}$ depend on all other features through the minimization of the violation of Kirchhoff's laws $|\Delta s|^2$ across all buses. Power flows through transmission lines and transformers that appear in this metrics induce a coupling between buses. Finding actually realistic values for ϑ and v requires to solve a nonlinear optimization program, which is usually achieved via a Newton-Raphson method [106].
- Meanwhile, bus voltage set points $\mathring{v}$ are controlled by dispatchers. Ideally, they aim at minimizing the sum of electrical losses p^{Joule} over all lines and transformers, while ensuring that all voltages are within acceptable values (*i.e.* minimize Δv). The dispatcher's decision thus has an impact over the actual state of the system.

Table 1.1 summarizes the features and metrics defined over power grids. Experiments on the AC Power Flow (AC-PF) problem evoked above are conducted in Chapter 8, and preliminary experiments on the voltage control problem are detailed in Chapter 9.

Object class	Features	Metrics
Bus	$\overline{v}, \underline{v}, \mathbb{1}^{pv}, \mathbb{1}^{slack}, \mathring{v}, \hat{v}, \hat{\vartheta}$	$\|\Delta s\|^2, \Delta v$
Load	$\mathring{p}, \mathring{q}$	-
Generator	$\mathring{p}, \mathring{q}$	-
Shunt	g, b	-
Transmission line	r, x, b^c	p^{Joule}
Transformer	$r, x, b^c, \tau, \vartheta^{shift}$	p^{Joule}

Table 1.1: Summary of features and metrics defined at each class of objects in power grids

Chapter 2

Deep Learning

This chapter briefly introduces foundations of Deep Learning (DL) to readers who are not familiar with it. Firstly, we explain how the broader domain of Machine Learning (ML) aims at exploiting data to learn to perform potentially complex tasks. Secondly, we define the Multi-Layer Perceptron (MLP) – which is the building block of Deep Learning (DL) – as a succession of layers of artificial neurons. We then detail how one may train deep neural networks in practice. Finally we focus on a specific class of neural networks aimed at processing images, and explain how their structure actually respects some of the data invariants. This will prove to be important in the next chapter which applies DL to graph data.

This chapter contains all relevant concepts to understand the remainder of the document. Still, it was not written as an exhaustive overview of DL, and we refer interested readers to the book *Deep Learning* by Ian Goodfellow and Yoshua Bengio and Aaron Courville [20].

2.1 Machine Learning

Machine Learning (ML) is a domain that aims at designing models that learn from data to perform certain tasks. In this section we introduce the concept of empirical risk minimization, explain how ML differs from pure optimization, and underline the importance of expressivity of a model.

2.1.1 Empirical risk minimization

Let us consider two metric spaces $\mathcal{X}$ and $\mathcal{Y}$. We associate a joint probability distribution to the product set $\mathcal{X} \times \mathcal{Y}$:

$$(x, y) \sim p(x, y) \tag{2.1}$$

Moreover, we assume that when x and y are sampled from $p(x, y)$, there exists a functional dependency between them, written as:

$$y = \boldsymbol{f}^*(x; \epsilon) \tag{2.2}$$

where ϵ is a random variable independent from x that models a noise that may come from measurements, hidden variables or any other nuisance factor. In this chapter we only consider regression problems, meaning that $\mathcal{Y}$ is a continuous space.

We aim at modelling this functional dependency in order to be able to predict the value of y from the sole knowledge of x. Since it is strictly impossible to search in the set of all functions, it is common in ML to search among a set of functions $\boldsymbol{f_\theta} : \mathcal{X} \to \mathcal{Y}$ parameterized by $\theta \in \Theta$[1]. We refer to this set of function as the *hypothesis space*. Ideally, we would like to find the best function in the hypothesis space, *i.e.* the function that minimizes the so-called *risk*:

$$\mathbb{E}_{x,y \sim p(x,y)} \left[L(\theta; x, y) \right] \tag{2.3}$$

where $L(\theta; x, y)$ is a loss function that estimates the quality of the model f_θ with regards to the sample (x, y). For instance, it is common to take $L(\theta; x, y) = \| \boldsymbol{f_\theta}(x) - y \|_2^2$. The optimal function may heavily depend on the choice of L, although we do not consider this issue in the present work.

If we had access to the actual distribution $p(x, y)$, then minimizing the risk would be a "simple" optimization problem. ML differs from pure optimization in the fact that we only have access to a *train set* $D_{train} = \{(x_m, y_m)\}_{m \in M_{train}}$ sampled from $p(x, y)$. As a consequence, we can only estimate the *empirical risk* over this dataset, which is defined as:

$$\frac{1}{|M|} \sum_{m \in M_{train}} L(\theta; x_m, y_m) \tag{2.4}$$

ML thus amounts to minimizing the empirical risk, in the hope that the actual risk will also decrease significantly. The Statistical Learning

[1] Many non parametric methods exist, but they fall out of the scope of the present document.

Theory domain provides tools to understand under which assumptions empirical risk minimization actually guarantees a low risk. We refer interested readers to the book *Statistical Learning Theory* by Vladimir Vapnik [107].

2.1.2 Generalization

The ability of a model f_θ to perform well on data sampled from $p(x, y)$ that do not appear in D_{train} is called *generalization*. In order to quantify the generalization capacity of a model, we can estimate its risk over the so-called *test set* $D_{test} = \{(x_m, y_m)\}_{m \in M_{test}}$ also sampled from $p(x, y)$. The risk over the test set is referred to as *generalization error*.

Some models are too "simple" compared to the function $\boldsymbol{f}^*$ they aim at imitating. For instance, quadratic functions are poorly approximated by linear mappings: even the best linear regression will have a high empirical risk. This phenomenon is called *underfitting* and is illustrated by the left part of Figure 2.1. It appears when the hypothesis space is not expressive enough and that patterns displayed by $\boldsymbol{f}^*$ cannot be imitated by any function from the hypothesis space.

In some other cases, the hypothesis space may contain functions that are too complex compared to $\boldsymbol{f}^*$. For instance, if we aim at imitating a quadratic function with the set of all polynomials based on a very small train set, then there is an infinite number of functions that achieve a zero empirical risk. However, most of those models will generalize poorly: although they achieve a minimal empirical risk by returning exactly the right value of y for any x that is in the train set, their intrinsic complexity prevents them from generalizing to new data. This phenomenon is called *overfitting* and is illustrated by the right part of Figure 2.1. It occurs when models "learn by heart" the train set, but are unable to generalize.

Overfitting is a major issue in ML and can be mitigated by various techniques [108]. Nevertheless, the remainder of the chapter will focus on devising a class of models that are expressive enough to imitate any continuous pattern so as to avoid underfitting.

2.2 Artificial Neural Networks

The problem of minimizing the empirical risk requires to first define an hypothesis space. While the domain of ML encompasses various methods such as decision trees [15], k-nearest neighbors [16], linear regression [17], naive bayes [18] or support vector machines [19], the present work only considers the case of neural networks. This Section

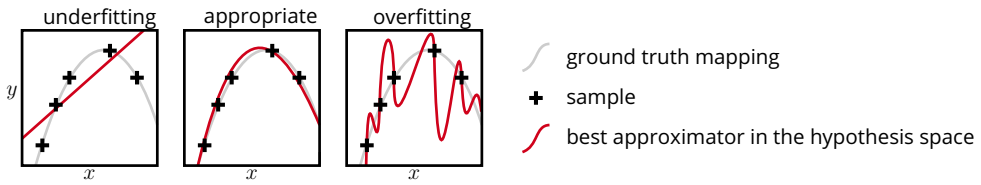


Figure 2.1: Illustration of the problems of underfitting and overfitting. They occur when an hypothesis space contains only functions that are too simple or only functions that are too complex with regards to the function to be modelled f^*. Train set is represented by the black dots, in red is shown the best function of three distinct hypothesis spaces: linear function (left), quadratic functions (middle) and all polynomials (right).

first introduces the concept of *artificial neuron* as a simple model of biological neurons. It then explains how multiple artificial neurons can be combined to form Single-Layer Perceptrons (SLPs) and then Multi-Layer Perceptrons (MLPs), which are a class of highly expressive functions.

2.2.1 Artificial Neurons

Artificial neurons have initially been designed as a simplistic model of biological neurons. Neurons – or nerve cells – are the main components of nervous tissues, and have the ability to communicate with each others through the transmission of electrical excitations, as illustrated in Figure 2.2. Actual neurons receive multiple excitations coming from multiple other neurons through their dendrites. Messages are summed in the cell body (soma), so as to obtain a single excitatory message. Finally, they transmit the resulting excitation to other neurons through the axons and terminal buttons.

Artificial neurons process information in a similar fashion[2]. Multiple scalar inputs $x_1, x_2, \ldots$ are received, weighted and then summed. A scalar quantity called *bias* is then added. Finally, the result goes through a non-linear mapping called *activation function*, and is sent to other neurons or as an output. Artificial neuron output a scalar quan-

[2]Other models such as "spiking neurons" mimic more realistically actual neurons. The artificial neurons considered here are only a very rough approximation.

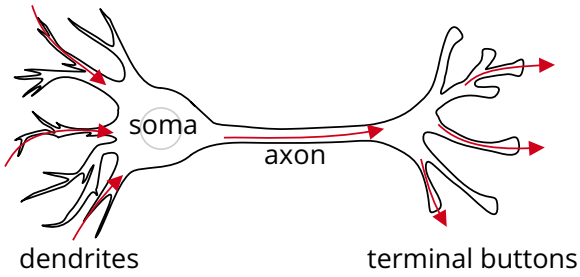


Figure 2.2: Schematic representation of a biological neuron.

tity in 1D, defined by the following equation:

$$\hat{y} = \sigma(w^\intercal x + b) \tag{2.5}$$

where $x \in \mathbb{R}^{d^x}$ is the vector of all input scalars, $w \in \mathbb{R}^{d^x}$ is a vector of weights, $b \in \mathbb{R}$ is the scalar bias, and $\sigma : \mathbb{R} \to \mathbb{R}$ is an activation function. Activation functions are usually continuous, monotonous and differentiable. Some mappings such as the Rectified Linear Unit (ReLU) illustrated in Figure 2.4 are not differentiable, but one can still define a subderivative[3]. Figure 2.3 shows the structure of a single neuron.

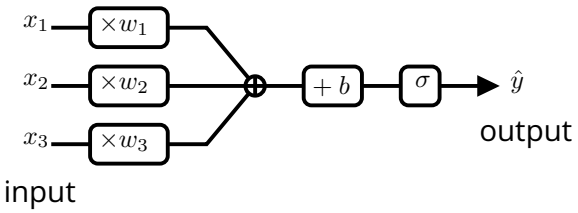


Figure 2.3: Schematic representation of an artificial neuron. Vector data $x = (x_1, x_2, x_3)$ is fed to the neuron. Each input is multiplied by a weight, and then summed. A non-linear activation function is then applied. A neuron can only return a scalar quantity $\hat{y} \in \mathbb{R}$.

2.2.2 Single-Layer Perceptrons (SLP)

In order to output vectors in multiple dimensions, one can stack multiple neurons to form a Single-Layer Perceptron (SLP). All neurons receive the same input, but process it differently, as they all have different weights and biases. It is common to consider that all neurons share the same activation function, although one may choose not to. There

[3] a generalization of the gradient for convex functions which are not differentiable.

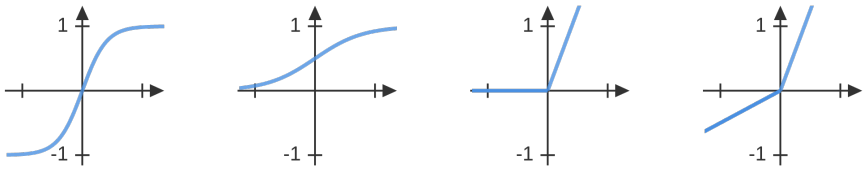


Figure 2.4: Commonly used activation functions. From left to right : hyperbolic tangent (tanh), sigmoid, Rectified Linear Unit (ReLU) and leaky ReLU.

is one neuron for each of the d^y dimensions of the output space. The action of an SLP over an input x is written as follows:

$$\hat{y} = \sigma(W.x + b) \tag{2.6}$$

where $W \in \mathbb{R}^{d^y \times d^x}$ (resp. $b \in \mathbb{R}^{d^y}$) is the concatenation of weights (resp. biases) of all neurons, and σ activation function. The action of σ is element-wise.

The main issue with SLPs is that they are not very expressive: they can only model functions that are almost linear. However, data encountered in real life are usually non-linear, and require much more expressive models.

2.2.3 Multi-Layer Perceptrons (MLP)

In order to improve the expressivity of neural networks, it is possible to stack multiple layers of SLPs, each layer receiving as input the output of the previous layer. Resulting models are called Multi-Layer Perceptrons (MLPs). An MLP with T layers is defined by the following equations:

$$h(0) = x \tag{2.7}$$

$$\forall t \in \{0, \dots, T-1\}, \quad h(t+1) = \sigma_t(W_t.h(t) + b_t) \tag{2.8}$$

$$\hat{y} = h(T) \tag{2.9}$$

where $(W_t)_{t=0,\dots,T-1}$, $(b_t)_{t=0,\dots,T-1}$ and $(\sigma_t)_{t=0,\dots,T-1}$ are respectively the weight matrices, bias vectors and activation functions of each layer. Variables $(h(t))_{t=1,\dots,T-1}$ are often referred to as *hidden layers*, *hidden variables*, or *latent variables*. We denote by $(d^t)_{t=1,\dots,T-1}$ their respective dimensions, and use the conventions $d^0 = d^x$ and $d^T = d^y$. Figure 2.5 compares an MLP to an SLP and an artificial neuron, and shows that hidden layers need not be of the same sizes.

In the case of a sigmoid activation, Cybenko proved in 1989 that a 2-layers perceptron is a Universal Approximator [109]. This property states that the class of all 2-layers perceptrons is dense in the space of continuous functions. In other words, any continuous function can be approximated with an arbitrarily low error by a 2-layers perceptron. Hornik then bounded the approximation error as a function of the amount of neurons in the hidden layer [110]. MLPs are thus very expressive mappings. By stacking multiple layers of neurons, they can build gradually more abstract latent representations of the input data, and thus be used for problems that require a high level of abstraction.

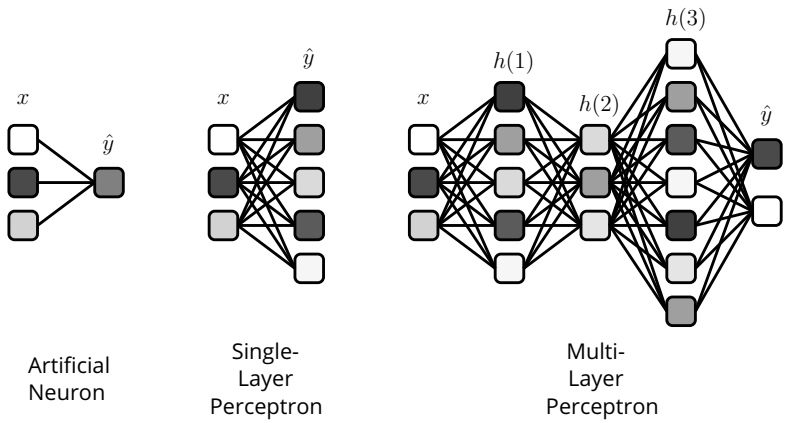


Figure 2.5: Schematic representations of an artificial neuron (left), a Single-Layer Perceptron (middle), and a Multi-Layer Perceptron (right). One can notice that not all hidden variables in an MLP need to be of the same size.

2.3 Training Neural Networks

MLPs are a class of highly expressive mappings whose parameters usually fall in one of these two categories:

- Parameters with regards to which the model can be derived: weights $(W_t)_{t=0,\dots,T-1}$ and biases $(b_t)_{t=0,\dots,T-1}$.
- Parameters for which no gradient can be computed: amount of layers T, hidden dimensions $(d_t)_{t=1,\dots,T-1}$ and activation functions $(\sigma_t)_{t=0,\dots,T-1}$.

This first category is called *trainable parameters*, and the second *hyperparameters*. Additional parameters that define the training process of trainable parameters also fall into the category of hyperparameters.

They will be defined below. Although both types of parameters are included in θ, it is common to denote only the trainable parameters by θ, and consider hyperparameters as implicit.

This section is devoted to explaining how one may search through the sets of trainable parameters, and how hyperparameters are tuned.

2.3.1 Learning trainable parameters

For now, let us consider that hyperparameters are fixed. It is common in the DL literature to rely on first-order gradient descent in the space of trainable parameters, so as to minimize the empirical risk. This optimization process is also referred to as "learning". We review some of the techniques involved in the learning of parameters.

We recall that the empirical risk over a single datapoint (x_m, y_m) is given by $L(\theta; x_m, y_m)$. The gradient of the loss with regards to trainable parameters is simply written as $\nabla_\theta L(\theta; x, y)$.

In the following, we detail methods to explore the space of trainable parameters using Stochastic Gradient Descent. We also detail the idea behind the back-propagation algorithm [111] which allows for an efficient computation of gradients.

Gradient descent

The simplest way to perform a gradient descent consists in computing the exact gradient of the empirical risk, and update weights iteratively until a good empirical risk is achieved:

$$\theta \leftarrow \theta - \eta \times \frac{1}{|M|} \sum_{m \in M} \nabla_\theta L(\theta; x_m, y_m) \tag{2.10}$$

where $\eta > 0$ is an hyperparameter called the *learning rate*, which controls the size of steps taken in the set Θ.

Minibatch gradient descent Computing the gradient over the whole dataset can be computationally exhausting, and one may accelerate it by estimating the gradient over minibatches $M_{batch} \subset M$ sampled from the train set [112]:

$$\theta \leftarrow \theta - \eta \times \frac{1}{|M_{batch}|} \sum_{m \in M_{batch}} \nabla_\theta L(\theta; x_m, y_m) \tag{2.11}$$

This provides an unbiased estimate of the gradient and can drastically reduce the computational time. At each update, a different minibatch M_{batch} is sampled from M.

Momentum The minibatch approach may also be quite slow in the case of small or noisy gradients. It can thus be accelerated by incorporating some momentum into the exploration of Θ [113]. This method introduces an additional variable v which plays the role of a velocity that accumulates an exponentially moving average of previous gradients.

$$v \leftarrow \alpha v - \eta \times \frac{1}{|M_{batch}|} \sum_{m \in M_{batch}} \nabla_\theta L(\theta; x_m, y_m) \tag{2.12}$$

$$\theta \leftarrow \theta + v \tag{2.13}$$

where $\alpha \in [0, 1]$ is the exponential decay rate of the velocity. Momentum can also be improved using Nesterov's accelerated gradient method [114], which consists in computing the gradient at $\theta + \alpha v$ instead of estimating it at θ.

Adaptative learning rates All the above methods are highly dependent on the choice of the learning rate. Research over the past decade has been focused on trying to dynamically adapt the gradient for each coordinate of θ, which gave rise to multiple optimization methods: AdaGrad [115], RMSProp [116], Adam [117], AdaMax and NAdam [118]. Among all the existing methods, Adam is probably the most widely used. It relies on estimating both the first order and second order moments of the gradients using an exponential averages controlled by parameters β_1 and β_2.

$$g \leftarrow \frac{1}{|M_{batch}|} \sum_{m \in Batch} \nabla_\theta L(y_m, \boldsymbol{f_\theta}(x_m)) \tag{2.14}$$

$$s \leftarrow \rho_1 s + (1 - \rho_1) g \qquad r \leftarrow \rho_2 r + (1 - \rho_2) g \odot g \tag{2.15}$$

s captures an estimation of the gradient, while r captures its norm according to all dimensions of θ. At the beginning of the training process, those estimators have a high bias. This is corrected by considering the following unbiased estimators:

$$\hat{s} \leftarrow \frac{s}{1 - \rho_1^t} \qquad \hat{r} \leftarrow \frac{r}{1 - \rho_2^t} \tag{2.16}$$

where t represents here the time step of the learning process (and should not be confused with the latent layer index commonly used in the present document). The trainable parameter θ is then updated as follows:

$$\theta \leftarrow \theta - \eta \frac{\hat{s}}{\sqrt{\hat{r}} + \delta} \tag{2.17}$$

where δ is usually set to a very small value. Default values of the present method are $\eta = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\delta = 10^{-8}$.

Computing the gradient using back-propagation

All the above optimizers rely on the ability to compute the derivative of models with regards to their trainable parameters. Closed-form equations of these gradients can be tedious to obtain. Thankfully, even very deep neural networks are nothing but a combination of small and simple differentiable operations, as shown in equations (2.7) to (2.9). The *chain rule* formula relates the derivative of a composition of functions with the derivatives of the said functions, allowing to write gradient estimation as a combination of easily computable terms. For instance, consider the two hidden layers neural network defined by

$$h(1) = \sigma_0(W_0.x) \qquad h(2) = \sigma_1(W_1.h(1)) \qquad \hat{y} = \sigma_2(W_2.h(2)) \tag{2.18}$$

Biases are disregarded to simplify notations. Gradients of the output with regard to the model parameters are given by the following equations:

$$\nabla_{W_2} loss = \nabla_{W_2}\hat{y}.\nabla_{\hat{y}} loss \tag{2.19}$$
$$\nabla_{W_1} loss = \nabla_{W_1}h(2).\nabla_{h(2)}\hat{y}.\nabla_{\hat{y}} loss \tag{2.20}$$
$$\nabla_{W_0} loss = \nabla_{W_0}h(1).\nabla_{h(1)}h(2).\nabla_{h(2)}\hat{y}.\nabla_{\hat{y}} loss \tag{2.21}$$

Computing gradients amounts to multiplying terms that are easy to compute, as illustrated in Figure 2.6. Moreover, one can observe that equations (2.20) and (2.21) share a common term. It appears that estimating the gradient of the output with regards to each of the trainable parameters can be done in a computationally efficient manner: the back-propagation algorithm [111] reverses the computational structure of the neural network to have gradient flow from the output to the weights of the model, as illustrated by the red arrows in Figure 2.6.

Weight initialization

Another key aspect of the training process concerns weights initialization. A poor initialization will very likely have the optimizer get stuck in a bad local optimum. Currently, there is no theoretically grounded rule that prescribes weight initialization, but in the case of sigmoid or hyperbolic tangent activation functions, it is common to use the *Normalized Xavier* [119] heuristic, which is defined by :

$$w_t \sim \mathcal{U}\left(\left[-\sqrt{\frac{6}{d^t + d^{t+1}}}, \sqrt{\frac{6}{d^t + d^{t+1}}}\right]\right) \tag{2.22}$$

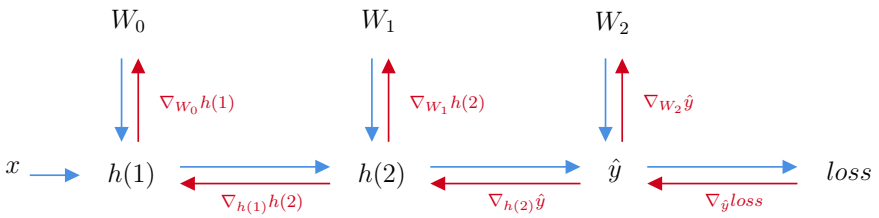


Figure 2.6: Back-propagation of the gradient – During a forward pass (blue arrows), operations go from the input and weights to the output. During the back-propagation (red arrows), gradient flows from the output to the trainable weights.

where $\mathcal{U}$ is the uniform distribution, and d^t, d^{t+1} are dimensions of the previous and current hidden layers. In the case of ReLU activation functions, it is recommended to use another heuristic called *He initialization* [120]. An overview of weight initialization techniques is provided in [121].

2.3.2 Hyperparameter tuning

Hyperparameters are the subset of parameters for which no gradient can be computed. They include both parameters that define the neural network architecture and parameters that define the learning process of trainable parameters (optimizer, learning rate, minibatch size, weight initialization strategy, etc.).

Validation set

Just like we cannot use the test set to learn trainable parameters of the neural network, we cannot use it to select the best set of hyperparameters. Hyperparameters have to be selected against a third dataset called called *validation set* $D_{val} = \{(x_m, y_m)\}_{m \in M_{val}}$. For each set of hyperparameters we want to evaluate, we fully train a model using the train set, then compute its error over the validation set. We then select the best set of hyperparameters with regards to their respective performance over the validation set. Only now can we estimate the generalization error of the resulting model over the test set. The test set can only be used to estimate the quality of a fully defined model. Other methods such as cross validation exist, but are not reviewed in the present document.

Hyperparameter optimization

There exists multiple techniques to explore the set of possible hyperparameters, which we propose to succinctly review.

Grid search This widely used method consists in selecting a few possible values for each hyperparameter, and to successively try all possible combinations. It explores the space of hyperparameters through a regular grid, which may be computationally exhausting if too many dimensions of the hyperparameter space are investigated.

Random search This method randomly selects sets of hyperparameters [122]. It typically outperforms grid search in the case where only a few hyperparameters affect the actual performance of the model.

Bayesian optimization It outperforms both grid and random searches by making educated guesses about which regions of the hyperparameter space to investigate [123]. It balances exploration (choosing configurations that are far from previous experiments) and exploitation (choosing configurations close to the best past experiments) to explore the set of hyperparameters in a thoughtful manner.

Other techniques such as early stopping [124] or evolutionary methods also provide additional tools to choose the best set of hyperparameters.

2.4 Convolutional Neural Networks

Previous sections have introduced basic MLPs, and how to train them. However, most recent successes of neural networks come from the fact that it is possible to design complex architectures that intrinsically encode some invariants or assumptions about the process to be modelled. The adequacy between the data structure and the neural network architecture is one main reason for most recent achievements of the DL domain. This principle being at the heart of the reflection underlying this thesis, we propose to illustrate it in its most renown application, namely Convolutional Neural Networks (CNNs).

CNNs [125] are a type of neural network architecture that specializes in processing data structures that are laid out as a regular grid [126]. They are commonly applied to time series (1 temporal dimension), pictures (2 spatial dimensions) and even videos (2 spatial and 1 temporal dimensions), and have achieved various successes in image

recognition [22, 127], video analysis [128, 129], natural language processing [130, 131], anomaly detection [132], drug discovery [133], Go game [134] and time series forecasting [135, 136].

2.4.1 Learning convolutions

CNNs rely on the use of convolution layers, which scan the input data and locally apply trainable filters to detect local patterns, and pooling layers, which compute local statistics over the data so as to reduce its size. Both types of layers are critical to the aforementioned successes, but we will solely discuss principles of the convolution layer, as some of the underlying ideas will prove essential in the next chapter.

Let us consider the case of 2d images. Each sample is denoted by $x \in \mathbb{R}^{d^h \times d^w}$, where d^h is the height and d^w is the width of the image. Each pixel is denoted by x_i where $i \in \{1, \dots, d^h\} \times \{1, \dots, d^w\}$ is a multi-index. Coordinates of the pixel in the 2d discrete space are given by $i = (i_1, i_2)$. It is common to consider images as the discretization of a multivariate function f, in which case we have $x_i = f(i)$, as shown in Figure 2.7.

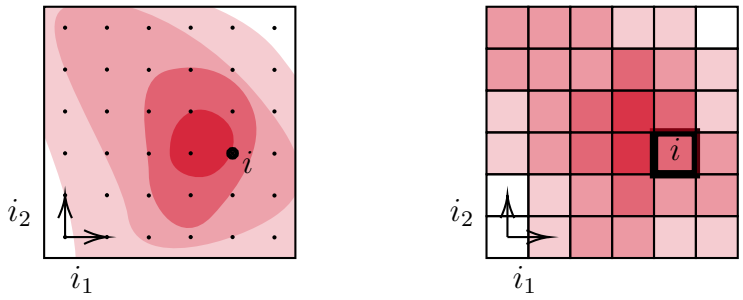


Figure 2.7: Discretization of a multivariate function to create a 2d image – The function f is sampled at evenly spread points.

Natural images are known to be equivariant per translation: one can translate a picture without altering its meaning. Thus, there has to be a way to represent pictures such that this representation will not be altered by translations. The mathematical operation called *convolution* provides such a tool, and will have us make a short detour to the *Signal Processing* domain.

Let f and g be two integrable functions. One can define the convolution of f and g as:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau) g(t - \tau) d\tau \tag{2.23}$$

The resulting function measures, for each value of shift t, how similar f and g actually are. If f is the signal of interest, and g is a small elementary pattern (with a comparatively small support), then $f * g$ will tell if this elementary pattern appears in f and where.

The above formula can be applied to discrete images as follows:

$$h_i = \sum_j f(j) \times g(i - j) \tag{2.24}$$

where j is also a multi-index. Using the change of variable $k = i - j$, introducing $w_k := g(k)$, and recalling that $f(j) = x_j$ leads to the following equation:

$$h_i = \sum_k x_{i-k} \times w_k \tag{2.25}$$

This equation describes the convolution operation used in CNNs : a small filter $w = (w_k)$ is swept across the input picture. It is multiplied element-wise with each small portion of the input picture, thus generating another picture[4].

An example of convolution for a simple 6×6 picture and a 3×3 filter is shown in Figure 2.8. The more similar the considered portion of the picture is to the filter, the higher the value of the resulting pixel. The blue part (top left) of the input data x is perfectly identical to the filter w, so the generated pixel has a large value. The orange portion (middle) of the picture is however very different from the filter, so the resulting pixel will have a low value. The generated filtered data h (left part of the figure) is a representation of where the elementary pattern encoded in the filter actually appears in the input data.

This convolution operation is done simultaneously with multiple filters. Some filters can be sensitive to horizontal lines, some to diagonals, etc. By stacking multiple layers of convolutions, the neural network is able to first detect elementary patterns, then detect patterns of elementary patterns, and then patterns of patterns of patterns, and so on. The neural network successively increases its abstraction level by building representations on top of each others.

Those filters are key to detect elementary features that appear in the previous layer. Although these filters may be manually designed, it is possible to train them as regular weights of a neural network. By doing so, the CNN can learn filters that are adapted to the problem at hand.

[4]the resulting picture is smaller, because of edge effect, but methods such as zero padding exist to account for that.

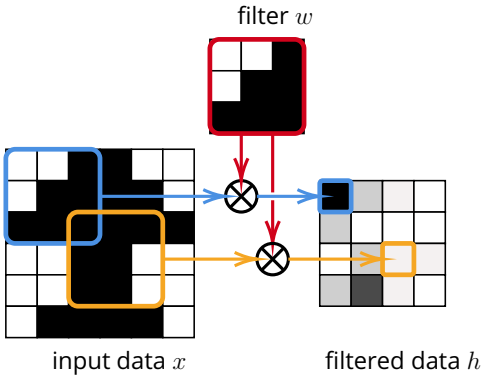


Figure 2.8: Convolution operation – The filter w is swept across the input data x, to generate a filtered picture h, which quantifies areas of the input that are the most similar to the filter.

2.4.2 Encoding invariants

A part of the success of CNNs can be attributed to their ability to encode data invariants directly into their architecture. As highlighted by Goodfellow, Bengio and Courville in their book *Deep Learning* [20], convolution layers exploit three important ideas : *sparse interactions, parameter sharing and equivariant representations.* These ideas allow to both increase the model efficiency, and decrease the amount of weights of the architecture.

First, the convolution operation is intrinsically local, contrarily to fully connected neural networks shown in Figures 2.5, where each neuron of a given layer is connected to each neuron of the next layer. It allows the neural network to focus on detecting local fundamental patterns. This is achieved by considering only reasonably small filters w (or equivalently by keeping the support of g small enough).

Second, weights are shared across the image. Filters are applied identically everywhere, regardless of the spatial coordinates. Thus the trained filters have to be relevant for the whole grid. They are trained simultaneously on every part of the input grid.

Finally, convolutions create *translation-equivariant* representations. If one were to translate the input data, the resulting filtered data would also be translated. Instead of working in the raw set of all possible pictures, it is as if the dataset was reduced to a set of equivalent classes. This can be seen as a way to fold the definition domain $\mathcal{X}$ along symmetry axes, bringing together dissimilar datapoints whose equivalent classes are actually very close.

Properly encoding invariants will prove crucial in the next chapter, which is devoted to another type of data: graphs.

Chapter 3

Graph Neural Networks

Power grids such as introduced in Chapter 1 have a quite atypical structure. They are made of the interconnection of a series of objects of various classes, thus forming a network. Modelling these complex structures without altering them is a difficult task, and constitutes a contribution of this PhD thesis deferred to Chapter 4. For now, we may as a first approximation frame power grids as graphs composed of vertices on the one hand (aggregating buses, loads, generators and shunts together), and edges on the other hand (aggregating transmission lines and transformers together), as illustrated in Figure 3.2.

In this chapter, we describe graphs, detail some of their properties, and explore how the recent domain of GNNs can handle such data structures. This work solely focuses on a specific type of GNNs, and we refer interested readers to the review paper written by Wu et al. [87] and to the book *Graph Representation Learning* by William L. Hamilton [88] for a more exhaustive presentation of the domain.

3.1 Graph data

In the previous chapter, we defined regression problems as trying to find a mapping $\boldsymbol{f_\theta}$ that best approximates a target mapping $\boldsymbol{f}^* : \mathcal{X} \to \mathcal{Y}$, when x follows the distribution $p(x)$. In the present chapter, we instantiate both input and output spaces $\mathcal{X}$ and $\mathcal{Y}$ as sets of graphs, and state the hypothesis that the target mapping $\boldsymbol{f}^*$ preserves the graph structure of its input.

3.1.1 Notations

We consider graphs that are made of two parts: a discrete network structure, and a set of continuous features. A typical instance of such

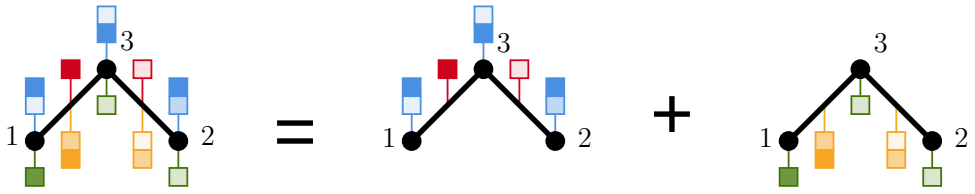
a graph is shown in Figure 3.1, and the conversion of a power grid into an homogeneous graph is displayed in Figure 3.2.

Structure The structure of a graph [137] is denoted by $(\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a set of $n \in \mathbb{N}$ vertices, and $\mathcal{E}$ is a set of directed edges. We consider the case where vertices are ordered, and use the simplifying notation $[n] := \{1, \dots, n\}$. As a consequence, we have that $\mathcal{V} = [n]$ and $\mathcal{E} \subseteq [n]^2$. We usually denote vertices by their index $i \in [n]$, and edges by their multi-index $(i, j) \in [n]^2$.

Features In addition to the discrete structure, continuous features are associated to each graph. Features are located at vertices or edges. Some can be considered as *input* features (*i.e.* directly observable) and encapsulated in x, while others are referred to as *output* features (*i.e.* that should be predicted) and encapsulated in y.

We denote by $x^v = (x_i^v)_{i \in [n]}$ input features located at vertices, and $x^e = (x_{ij}^e)_{(i,j) \in \mathcal{E}}$ input features located at edges, such that $x = (x^v, x^e)$. Similar notations are used for output features $y = (y^v, y^e)$. The dimension of input (resp. output) features at vertices is denoted by $d^{v,x}$ (resp. $d^{v,y}$), while the dimension of input (resp. output) features at edges is denoted by $d^{e,x}$ (resp. $d^{e,y}$).

We denote by $\mathcal{X}_{n,\mathcal{E}}$ and $\mathcal{Y}_{n,\mathcal{E}}$ the set of input and output features that are defined over the graph structure $([n], \mathcal{E})$. Moreover, we denote by $\mathcal{X}$ and $\mathcal{Y}$ the sets of all input and output features defined over any graph structure.



(x, y) $\quad x = (x^v,\ x^e)$ $\quad y = (y^v, y^e)$

Figure 3.1: Example of a graph (x, y), with $V = \{1, 2, 3\}$ and $\mathcal{E} = \{(1, 3), (2, 3)\}$. Input (resp. output) features are in $d^{v,x} = 2$ (resp. $d^{v,y} = 1$) dimensions at vertices and in $d^{e,x} = 1$ (resp. $d^{e,y} = 2$) dimensions at edges. We use the convention that output features are represented above the graph, while output features are displayed below it.

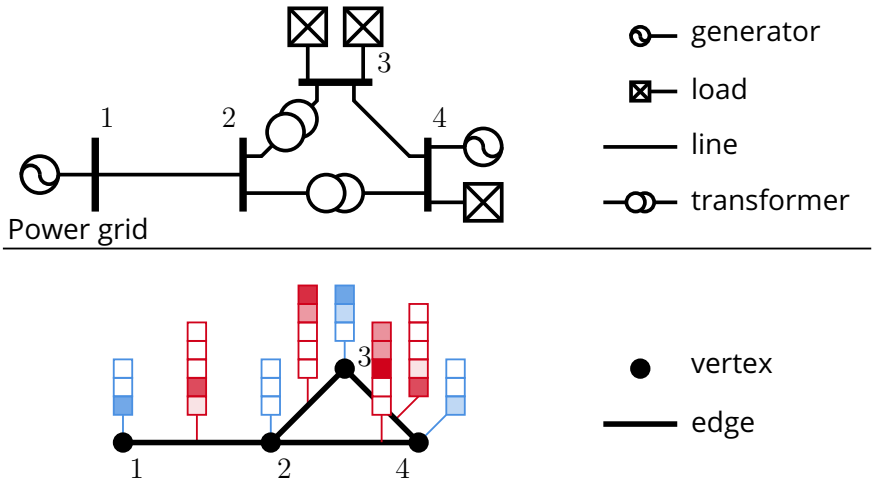


Figure 3.2: Power grid instance and its conversion into an homogeneous graph. Input features are in the following dimensions: $d^{v,x} = 3, d^{e,x} = 5$. Generators and loads are aggregated together on the one hand, and lines and transformers on the other hand. For the sake of readability, only input features are considered.

Dense and sparse representations Edge features can be numerically represented in two distinct ways: dense or sparse. Figure 3.3 presents the two approaches

In the dense representation, all possible edges in $[n]^2$ are represented. Thus, edge features are concatenated in a $n \times n \times d^{e,x}$ tensor, and it is required to associate a default feature for edges that are not in $\mathcal{E}$ (usually 0). Moreover, it makes it impossible to consider parallel edges that share the same vertex connections, but not necessarily the same features. Nevertheless, we use this representation in some figures of the present document (see Figure 3.1), because it makes the impact of vertex ordering clearer.

The sparse representation on the other hand consists in treating edge features as a list of tuples $(i, j, x^e_{ij})_{(i,j)\in\mathcal{E}}$. Thus, edge features x^e are represented as a $m \times (2 + d^{e,x})$ matrix. When considering graphs that have relatively few edges (*i.e.* $m \ll n^2$), the sparse representation provides a consequent gain in terms of memory. Moreover, it allows to consider parallel edges that are connected to the same vertices.

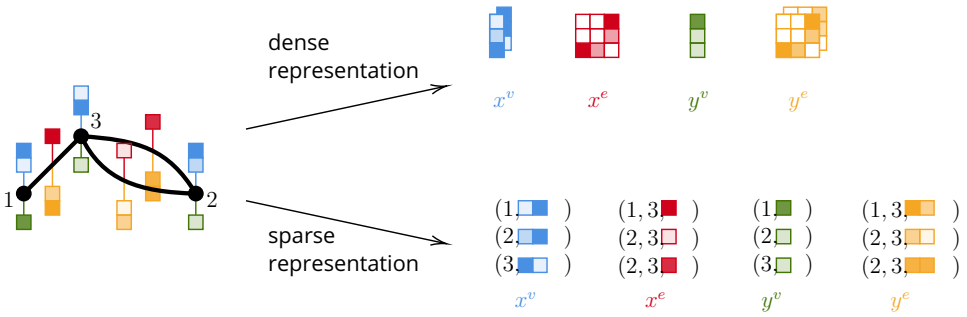


Figure 3.3: Dense and sparse representations of a graph. The dense representation associates a default value to edges that are not in $\mathcal{E}$, while the sparse representation only considers existing edges. Sparse representation can also directly handle parallel edges, while the dense representation has to convert them into single edges, which requires a deep understanding of the problem at hand. Edges directions are not displayed for the sake of readability.

3.1.2 Invariance & equivariance under permutations

The above definition of graphs relies on the ordering of vertices $[n]$. However, in the general case, there is no unique or *natural* way to order vertices: Why should the vertex corresponding to Brussels be ranked above the vertex corresponding to Paris (or the other way around)? Still, the numerical representation of graph data heavily depends on the choice of vertex ordering, while the underlying structure remains unaltered.

Permutations (bijective mappings from $[n]$ to $[n]$) allow to switch from one vertex ordering to another, consequently changing the numerical representation. When applied to a graph input x, a permutation $\sigma \in \Sigma_n$ changes the vertex ordering, while preserving the features:

$$\sigma \star x^v = (x^v_{\sigma^{-1}(i)})_{i \in [n]} \tag{3.1}$$

$$\sigma \star x^e = (x^e_{\sigma^{-1}(i)\sigma^{-1}(j)})_{(i,j) \in \mathcal{E}} \tag{3.2}$$

Permutations over outputs y are analogous. Figure 3.4 presents the impact of a permutation over a small graph, and over its numerical representation.

While the numerical representation of (x, y) is inevitably altered by permutations, there exist functions that are able to withstand such perturbations.

Definition 1. *Let $d \in \mathbb{N}$ and $\boldsymbol{f} : \mathcal{X} \to \mathbb{R}^d$. $\boldsymbol{f}$ is said to be invariant per*

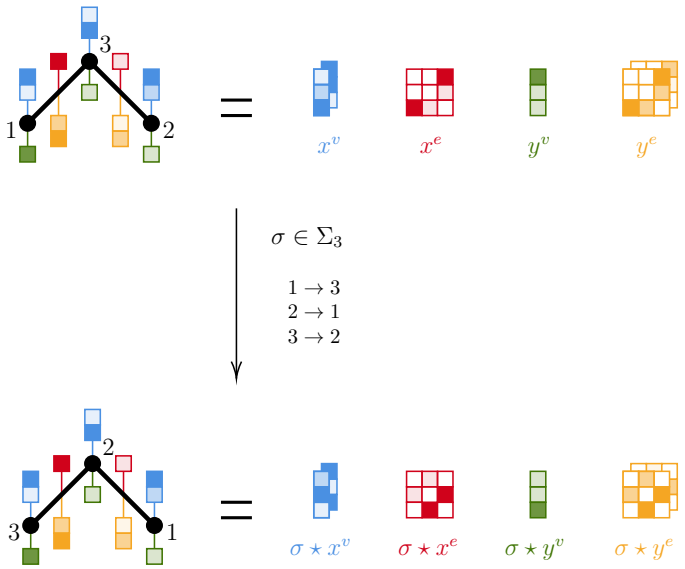


Figure 3.4: Permutation of a graph (x, y). The vertex ordering is altered, thus modifying the numerical representation of the data. However, the underlying graph (structure and attached features) remains untouched, as highlighted by the left part of the figure.

permutation if

$$\forall n \in \mathbb{N}, \forall \mathcal{E} \subseteq [n]^2, \forall x \in \mathcal{X}_{n,\mathcal{E}}, \forall \sigma \in \Sigma_n, \boldsymbol{f}(\sigma \star x) = \boldsymbol{f}(x) \tag{3.3}$$

A typical instance of such a mapping would be a function that outputs some global statistics about the graph structure (amount of vertices, amount of edges, diameter, etc.): regardless of the vertex ordering, the output is strictly identical. The dimension of the output space is not critical. See Figure 3.5 for a graphical representation of permutation-invariance.

Definition 2. *Let $\boldsymbol{f} : \mathcal{X} \to \mathcal{Y}$. $\boldsymbol{f}$ is said to be equivariant per permutation if*

$$\forall n \in \mathbb{N}, \forall \mathcal{E} \subseteq [n]^2, \forall x \in \mathcal{X}_{n,\mathcal{E}}, \begin{cases} \boldsymbol{f}(x) \in \mathcal{Y}_{n,\mathcal{E}} \\ \forall \sigma \in \Sigma_n, \boldsymbol{f}(\sigma \star x) = \sigma \star \boldsymbol{f}(x) \end{cases} \tag{3.4}$$

The condition $\boldsymbol{f}(x) \in \mathcal{Y}_{n,\mathcal{E}}$ enforces that the function $\boldsymbol{f}$ preserves the graph structure of its input. A typical instance of such a mapping would be a function that outputs for each vertex its amount of neighbors. See Figure 3.6 for a graphical representation of permutation-equivariance.

The notions of *invariance* and *equivariance* will prove to be central in the remainder of this document.

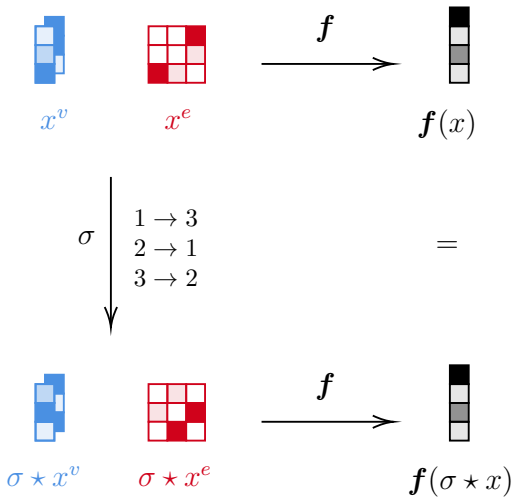


Figure 3.5: Permutation-invariant mapping. Regardless of the vertex ordering, the mapping $\boldsymbol{f}$ produces the exact same output. The dimension of the output of $\boldsymbol{f}$ is not critical.

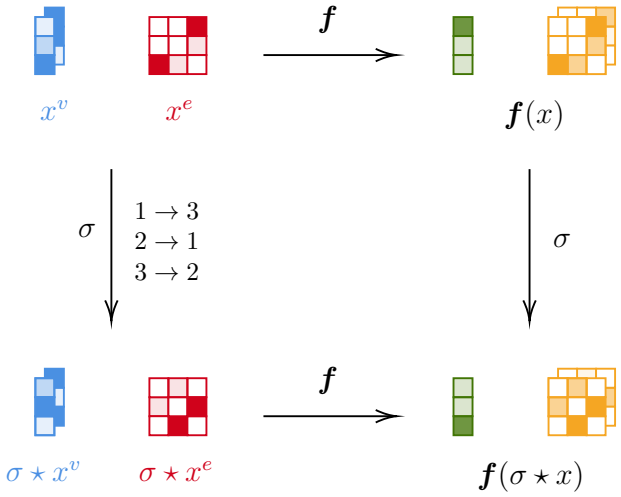


Figure 3.6: Permutation-equivariant mapping. The ordering of the output $\boldsymbol{f}(x)$ reflects the ordering of the input x. While the output of $\boldsymbol{f}$ is different when applied to x and $\sigma \star x$, the underlying graph is strictly identical, and the same permutation σ allows to switch from one output to the other.

3.1.3 Distributions & Hypotheses

In the case of power grid applications, graphs are sampled according to a distribution that makes the discrete structure $([n], \mathcal{E})$ and the continuous features vary altogether. We denote by $p(x, y)$ the joint probability

associated with the graph instance $(x, y) \in \mathcal{X} \times \mathcal{Y}$. Indeed, the support of this probability distribution is strictly restricted to pairs x and y that have the same graph structure. Additionally, all input and output features at vertices and edges should share the same dimensions. A typical distribution of graph data is illustrated in Figure 3.7.

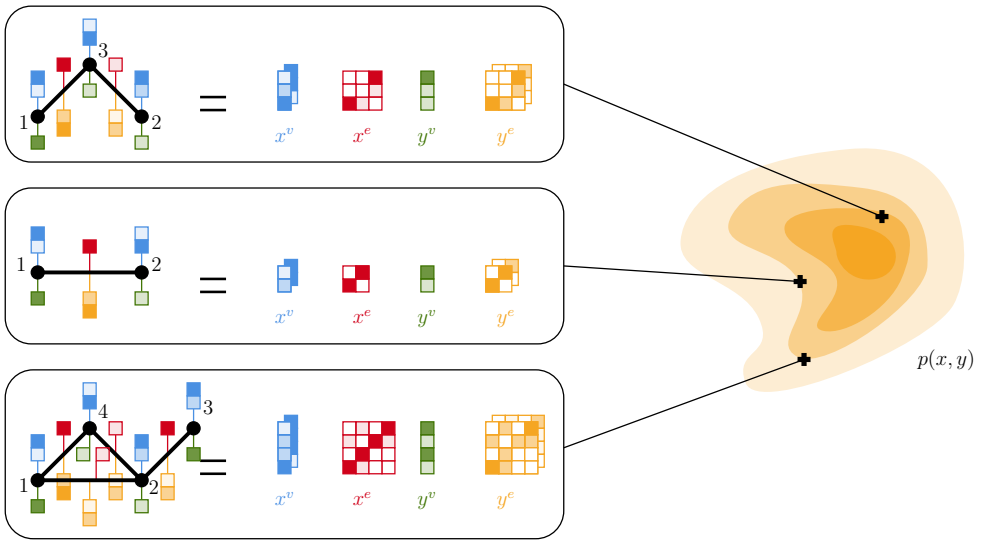


Figure 3.7: Representation of a graph probability distribution $p(x, y)$. Both the graph discrete structure and the continuous features vary.

Similarly to the hypothesis stated in equation (2.2), we assume that there is a functional relationship between inputs and outputs:

$$y = \boldsymbol{f}^*(x) \tag{3.5}$$

Moreover, we assume that the problems we are interested in are not bound to any vertex ordering, *i.e.* that $\boldsymbol{f}^*$ is permutation-equivariant. Our goal is to approximate this relationship using a function $\boldsymbol{f}_\theta$. The permutation equivariance property is extremely important: we do not want the quality of our prediction to be altered by a simple change of vertex ordering.

A possible approach could be to train a fully-connected neural network $\boldsymbol{f}_{\boldsymbol{\theta}}$ to be permutation-equivariant. This would require to perform some data-augmentation by considering all permuted versions of samples $(x, y) \sim p(x, y)$. While this strategy is not necessarily a bad option on small graphs, it becomes intractable for large ones (there are $n!$ permutations for graphs with n vertices). Moreover, regular neural networks are bound to the size of their input: it would be impossible

to use a trained model on a graph that is of different dimensions than the ones it was trained on.

A more viable solution lies in the growing field of Graph Neural Networks (GNNs), a class of neural networks that are built to process graph data: they are inherently permutation-equivariant, and output graphs that have the same structure as their input.

3.2 Graph Neural Networks

To sum things up, we are interested in imitating a mapping $\boldsymbol{f_\theta} : \mathcal{X} \to \mathcal{Y}$, that maps input graphs to output graphs, while preserving the structure in a permutation-equivariant way. Thankfully, the domain of GNNs defines a class of parameterized permutation-equivariant mappings, which thus provide good candidates for the imitation of $\boldsymbol{f_\theta}$. In this section we introduce the elementary operations upon which GNNs are built and provide an instance of such a neural network architecture.

3.2.1 Of images and graphs

Before diving into the proper definition of the GNN architecture, let us compare images and general graphs in the context of DL.

The convolution operation has proven to be extremely efficient for processing image data (or any data laid out as a regular grid). It relies on the fact that each pixel can naturally be associated with Euclidean coordinates in a low dimensional space. In such a case relative positions between pixels can be defined, as shown in equation (2.25). Moreover, pixels coordinates lie on a regular grid, thus enforcing that relative positions take a finite amount of values: instead of learning the convolution function over a continuous set, one only has to learn its values on a regular grid.

Working with general graphs is very different. First of all, there are no Euclidean coordinates associated with each vertex: there are no relative positions between vertices. Secondly, the only kind of distance we can work with stems from the notion of neighborhood. As explained in the previous section, there is no *intrinsic* way of ordering neighbors of a vertex. Moreover, vertices do not necessarily have the same amount of neighbors, as shown in Figure 3.8. In order to compute local statistics about the neighborhood of a vertex, one has to aggregate information using operations such as the *sum*, *mean*, *max*, etc.

As to the down-sampling operation commonly used in computer vision, it usually relies on the idea that one can aggregate local groups

of pixels together, so as to reduce the complexity of the data. Those groups are easy to select because of the regular structure of the input.

For general graphs, there is no *intrinsic* way of clustering vertices so as to create smaller and simpler graphs. Thus, all latent representation used by a neural network should stick to the graph structure of the input.

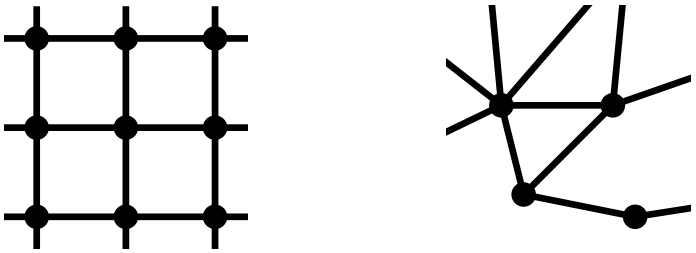


Figure 3.8: Structure of an image (left) vs. Structure of a graph (right). While both can be considered as graphs, the structure of images is a regular grid that can naturally be embedded in a low dimensional Euclidean space.

3.2.2 Graph Neural Network architecture

As mentionned in the Introduction, there exist multiple ways of implementing GNNs. Some rely on a fixed-point method, some on a spectral decomposition of the input graph, and others rely on a series of local operations. In this document, we only describe the latter method, which is called Spatial Graph Neural Network (GNN). For the sake of readability, we simply refer to it as Graph Neural Network (GNN).

This type of neural network architecture relies on an iterative process where vertices and edges of the input graph exchange messages between direct neighbors. It can be decomposed into three main steps:

- Encoding: each vertex and edge embeds its own input into a latent space.
- Message Passing: vertices and edges iteratively exchange information between direct neighbors.
- Decoding: each vertex and edge converts the result of the message passing into actual output values.

All those steps involve the use of multiple trainable neural networks, which are trained jointly.

In order to properly explain the main ideas to readers that are not familiar with this approach, we propose to explain it from multiple perspectives. First, we show the whole neural network architecture in the form of Algorithm 1. Then, each step is further detailed by both an explanatory paragraph and a figure. Finally, Figure 3.12 shows how information flows from the input x to the prediction $\hat{y}$ through the neural network architecture. Moreover, the latter figure outlines the way neural network blocks are laid out and shared across the architecture.

Algorithm 1 GNN forward pass

1: **procedure** $f_{\boldsymbol{\theta}}(x = ((x_i^v)_{i\in[n]}, (x_{ij}^e)_{(i,j)\in\mathcal{E}}))$
2:
3: ▷ Encoding
4: **for** $i \in [n]$ **do**
5: $h_i^v \leftarrow \boldsymbol{\phi}_{\boldsymbol{\theta}}^{\boldsymbol{v,encoder}}(x_i^v)$
6: **for** $(i,j) \in \mathcal{E}$ **do**
7: $h_{ij}^e \leftarrow \boldsymbol{\phi}_{\boldsymbol{\theta}}^{\boldsymbol{e,encoder}}(x_{ij}^e)$
8:
9: ▷ Message passing
10: **for** $t = 0, \ldots, T-1$ **do**
11: **for** $i \in [n]$ **do**
12: $h_i^v \leftarrow \boldsymbol{\phi}_{\boldsymbol{\theta}}^{\boldsymbol{v,t}}(h_i^v, \{h_{ij}^e\}_{(i,j)\in\mathcal{E}}, \{h_{ji}^e\}_{(j,i)\in\mathcal{E}})$
13: **for** $(i,j) \in \mathcal{E}$ **do**
14: $h_{ij}^e \leftarrow \boldsymbol{\phi}_{\boldsymbol{\theta}}^{\boldsymbol{e,t}}(h_{ij}^e, h_i^v, h_j^v)$
15:
16: ▷ Decoding
17: **for** $i \in [n]$ **do**
18: $\hat{y}_i^v \leftarrow \boldsymbol{\phi}_{\boldsymbol{\theta}}^{\boldsymbol{v,decoder}}(h_i^v)$
19: **for** $(i,j) \in \mathcal{E}$ **do**
20: $\hat{y}_{ij}^e \leftarrow \boldsymbol{\phi}_{\boldsymbol{\theta}}^{\boldsymbol{e,decoder}}(h_{ij}^e)$
21:
22: **return** $\hat{y} = ((\hat{y}_i^v)_{i\in[n]}, (\hat{y}_{ij}^e)_{(i,j)\in\mathcal{E}})$

Encoding *(See Algorithm 1, lines 3 to 7)* The first part of the GNN architecture consists in embedding features of the input graph into a latent space of dimension d – which is an hyperparameter of the GNN architecture. Each vertex $i \in [n]$ applies the exact same neural network $\boldsymbol{\phi}_{\boldsymbol{\theta}}^{\boldsymbol{v,encoder}}$ to its input x_i^v, thus creating a latent variable $h_i^v \in \mathbb{R}^d$. Similarly, each edge $(i,j) \in \mathcal{E}$ applies the exact same neural network

$\phi_{\theta}^{e,encoder}$ to its input x_{ij}^{e}, thus creating a latent variable $h_{ij}^{e} \in \mathbb{R}^{d}$. It is important to notice that those operations are performed in parallel:

- there is no exchange of information between vertices and / or edges;
- the vertex encoder $\phi^{v,encoder}$ is shared across vertices, and the edge encoder $\phi^{e,encoder}$ is shared across edges.

It is possible to choose different dimensions for vertex and edge latent spaces. However, we choose to use d for both so as to alleviate notations. This hyperparameter should be sufficiently large to allow vertices and edges to store and exchange enough information.

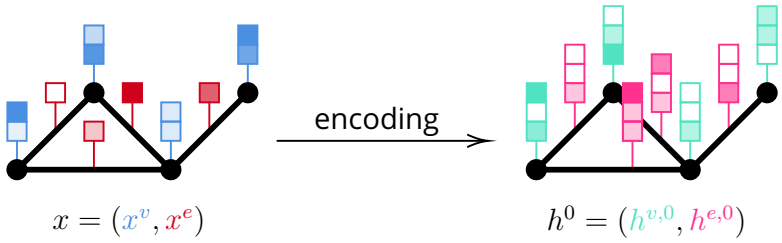


Figure 3.9: Encoding step: a vertex encoder is applied to all vertices in parallel, while an edge encoder is applied to all edges in parallel. The graph structure is indeed preserved.

Message Passing *(See Algorithm 1, lines 9 to 14)* The second part of the GNN architecture consists in iteratively exchanging information between vertices and edges[1]. Edges receive information from the vertices they are connected to, while vertices receive information from edges that are connected to them. As it appears, there is an asymmetry between edges and vertices: edges are connected to exactly two vertices, while vertices can be connected to a varying amount of edges. Vertices and edges are alternatively updated T times – where T is an hyperparameter of the GNN architecture. Vertices and edges are updated as follows :

- Each vertex $i \in [n]$ is updated using the trainable neural network $\phi_{\theta}^{v,t}$, which takes as input h_i^v and latent variables of edges that are connected to it. Since some edges are connected to i through their first index, and others through their second index, we make

[1]In this implementation, we only consider interactions between edges and vertices, so as to alleviate notations. It is however very common in the GNN literature to also have vertices that are connected through an edge to directly interact.

a distinction between both cases. Thus the latent state is updated according to $\{h_{ij}^e\}_{(i,j)\in\mathcal{E}}$ on one side and $\{h_{ji}^e\}_{(j,i)\in\mathcal{E}}$ on the other side. Those being unordered sets, they should be embedded into vectors, using basic operations such as the *sum*, *mean*, *product*, etc.

- Each edge $(i,j) \in \mathcal{E}$ is updated using the trainable neural network $\boldsymbol{\phi_\theta^{e,t}}$, which takes as input h_{ij}^e and the updated states of the two vertices to which it is connected: h_i^v and h_j^v.

In the GNN literature, $\boldsymbol{\phi_\theta^{e,t}}, \boldsymbol{\phi_\theta^{v,t}}$ can be different at each t, or exactly the same (creating a recurrent architecture).

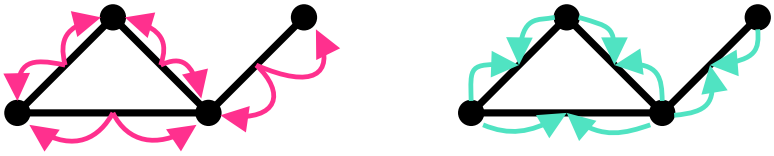


Figure 3.10: Message passing: first, vertices are updated depending on the latent variable of the edges to which they are connected (left), and then edges are updated depending on the latent variables of the vertices to which they are connected (right).

Decoding *(See Algorithm 1, lines 16 to 20)* The last past of the neural network architecture is akin to the first part. Trainable neural networks $\boldsymbol{\phi_\theta^{v,decoder}}$ and $\boldsymbol{\phi_\theta^{e,decoder}}$ are applied in parallel to the latest latent variable of all vertices and all edges. These mapping output vectors of dimension $d^{v,y}$ at vertices, and $d^{e,y}$ at edges. Moreover, since all previous operations also preserve the input graph structure, the final output is indeed in the right space: $\hat{y} \in \mathcal{Y}_{n,\mathcal{E}}$.

Training The $2 + 2T + 2$ ($2 + 2 + 2$ in the recurrent case) neural networks are commonly instantiated as standard MLPs, and are trained jointly. Each of these neural networks have their own hyperparameters: depth, width and activation function. In this regard, GNNs are not different to other deep learning architectures and do not require any specific algorithm with regards to training.

In this chapter, we have introduced a data formalism that can represent power grids. The relationship between the considered inputs and outputs is permutation-equivariant, which is a major issue

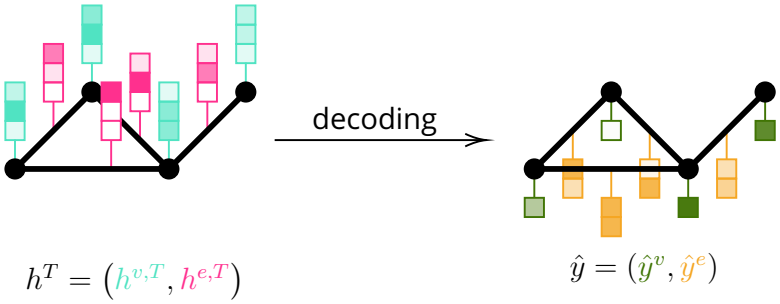


Figure 3.11: Decoding step: edge and vertex decoders are applied in parallel to all edges and vertices, so as to produce an actual prediction defined in $\mathcal{Y}_{n,\mathcal{E}}$.

for directly using regular neural networks. Then we have detailed a type of GNN architecture that is intrinsically permutation-equivariant, and that adapts the structure of its output to its input. Shortcomings of both the data formalism and the corresponding GNN architecture will be investigated in Chapter 4. This class of function is a natural candidate for modelling the functional relation between inputs and outputs of graph data. This intuition will be further entailed by our theoretical contribution in Chapter 6.

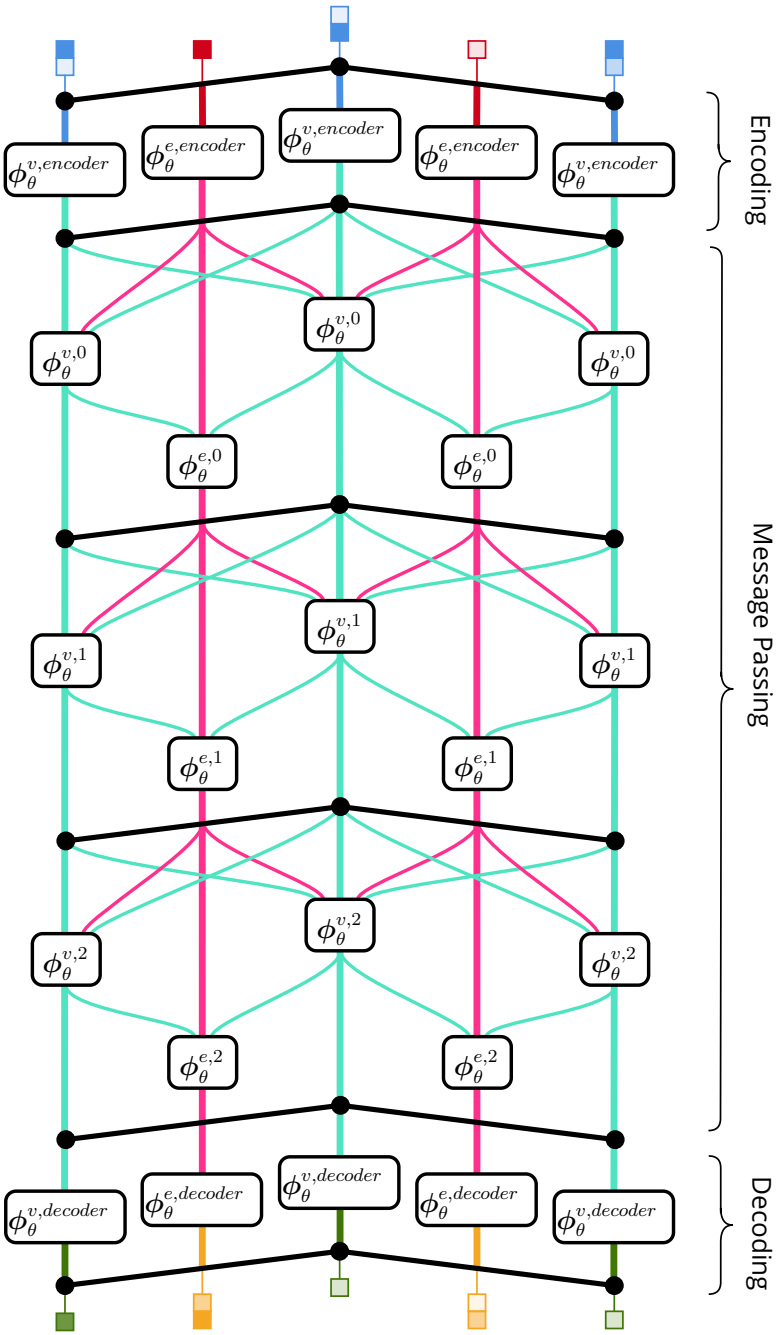


Figure 3.12: Example of a GNN applied to the graph shown in Figure 3.1. Information flows from the top (input) to the bottom (output).

Part II

The Deep Statistical Solver Approach

Chapter 4

Deep Statistical Solver Architecture

Traditional graphs introduced in Chapter 3 do not allow for a seamless representation of power grids. The former are exclusively composed of vertices and edges, while the latter are made of a series of objects of various classes. Even worse, it is impossible to consider multiple objects located at the same vertex in parallel. Unfortunately, it is extremely common in power grids to have multiple objects of the same class (generators, loads or shunts) connected to the exact same bus. In order to fit the traditional graph formalism, these can be aggregated together, which most likely implies some information loss. For instance, it is not equivalent to control two collocated shunts or a single shunt that is twice as large. This conceptual deadlock uncovers the limits of modelling power grids with traditional graphs, and pushes in favor of the definition of a graph formalism that is able to handle power grids as is.

In this chapter, we detail the notion of Hyper Heterogeneous Multi Graph (H2MG), a type of graph structure that represents power grids more easily. In addition, we propose a GNN architecture called the Hyper Heterogeneous Multi Graph Neural Network (H2MGNN), that draws inspiration from dynamical systems and the Neural Ordinary Differential Equation (NODE) literature. It is natively compatible with H2MGs. We also detail key ideas that help improve the stability and performance of the model.

4.1 Hyper Heterogeneous Multi Graphs

Power grids are made of a series of objects that can be organized into the following classes: buses, generators, loads, shunts, transmission

lines and transformers. Depending on its class, an object is defined by a varying amount of parameters of various units, and is located at one or two vertices (Section 9.2 investigates the case of objects located at four vertices).

In this section, we propose a data formalism called Hyper Heterogeneous Multi Graph (H2MG), which allows for a simpler integration with power grid data compared to traditional graphs, and that does not cause any loss of information.

Objects and vertices We consider a structure composed of objects on the one hand, and vertices on the other hand. Objects are connected to vertices, and therefore connected to each others via vertices: In such networks, vertices play the role of interfaces between objects. Moreover, objects that belong to the same class share some characteristics: They are connected to the exact same amount of vertices, and bear feature vectors of the same sizes. Meanwhile, vertices do not bear any feature vector, and only play the role of addresses (or interfaces).

The proposed formalism lies at the interface of the following kinds of graphs:

- Hyper-graphs: Graphs that have hyper-edges, which can be connected to any number of vertices [138].
- Heterogeneous graphs: Graphs that are made of multiple classes of objects [139].
- Multi-Graphs: Graphs that allow multiple objects to have the same addresses [137].

Hyper-edges and classes Let $n \in \mathbb{N}$, and $\mathcal{C}$ be the set of considered classes. We denote by $\mathcal{E}^c$ the set of hyper-edges of class c. All such hyper-edges are connected to the same amount of vertices through their ordered ports $\mathcal{O}^c$. Thus, $\mathcal{E}^c \subseteq [n]^{|\mathcal{O}^c|}$. Classes such that $|\mathcal{O}^c| = 1$ represent objects that are located at exactly one vertex (such as generators or loads in power grids). Classes such that $|\mathcal{O}^c| = 2$ represent objects that are located at exactly two vertices (such as transmission lines or transformers in power grids).

Multi objects Let $c \in \mathcal{C}$ and $e \in \mathcal{E}^c$. We denote by $\mathcal{M}_e^c$ the set of objects of class c that lie on hyper-edge e. Those objects may bear different feature vectors, and cannot be simply aggregated into an equivalent object.

Structure A H2MG is composed of a structure that defines the interconnection patterns of objects, and some features that are attached to each object. We denote the structure of a H2MG x as $(n, \mathcal{C}, \mathcal{E}, \mathcal{M})$, where $\mathcal{E} = (\mathcal{E}^c)_{c\in\mathcal{C}}$ and $\mathcal{M} = ((\mathcal{M}^c_e)_{e\in\mathcal{E}^c})_{c\in\mathcal{C}}$. Moreover, we use the following simplifying notation:

$$\mathcal{G}_x = \{(c, e, m) | c \in \mathcal{C}, e \in \mathcal{E}^c, m \in \mathcal{M}^c_e\} \tag{4.1}$$

Let $i \in [n]$. We call *hyper-edge neighborhood of a vertex* the set of hyper-edges that are connected to it.

$$\mathcal{N}_x(i) = \{(c, e, m, o) | (c, e, m) \in \mathcal{G}_x, o \in \mathcal{O}^c, e_o = i\} \tag{4.2}$$

One may observe that this set returns the class, the hyper-edge, the multi-object id and the port through which each object is connected to i.

Features Contrarily to standard graphs, H2MGs exclusively bear features at hyper-edges: vertices only play the role of addresses to which hyper-edges can be connected. In that sense, vertices should be seen as an interface between hyper-edges. The corresponding graph data can still be written as (x, y) where x is the input and y the output. Figure 4.1 shows how a power grid can seamlessly be framed as a H2MG.

$$x = (x^c_{e,m})_{(c,e,m)\in\mathcal{G}_x} \tag{4.3}$$

$$y = (y^c_{e,m})_{(c,e,m)\in\mathcal{G}_y} \tag{4.4}$$

All input and output features of hyper-edges of a given class $c \in \mathcal{C}$ are in $d^{c,x}$ and in $d^{c,y}$ dimensions. We denote by $\mathcal{X}_{n,\mathcal{C},\mathcal{E},\mathcal{M}}$ and $\mathcal{Y}_{n,\mathcal{C},\mathcal{E},\mathcal{M}}$ the set of input and output graphs defined over the graph structure $(n, \mathcal{C}, \mathcal{E}, \mathcal{M})$. We denote by $\mathcal{X}$ and $\mathcal{Y}$ the sets of all possible input and output graphs defined over all possible structures.

We define a pair of H2MG (x, y) to be *compatible*, if they share the same graph structure $(n, \mathcal{C}, \mathcal{E}, \mathcal{M})$.

Permutations The impact of permutations over H2MGs is analogous to that on standard graphs:

$$\sigma \star x = \left(x^c_{\sigma^{-1}(e),m}\right)_{(c,e,m)\in\mathcal{G}_x} \tag{4.5}$$

$$\sigma \star y = \left(y^c_{\sigma^{-1}(e),m}\right)_{(c,e,m)\in\mathcal{G}_x} \tag{4.6}$$

with $\sigma^{-1}(e) = (\sigma^{-1}(e_o))_{o\in\mathcal{O}^c}$. Definitions of permutation-invariant and permutation-equivariant mappings are also analogous to Definitions 1 and 2 introduced in Chapter 3.

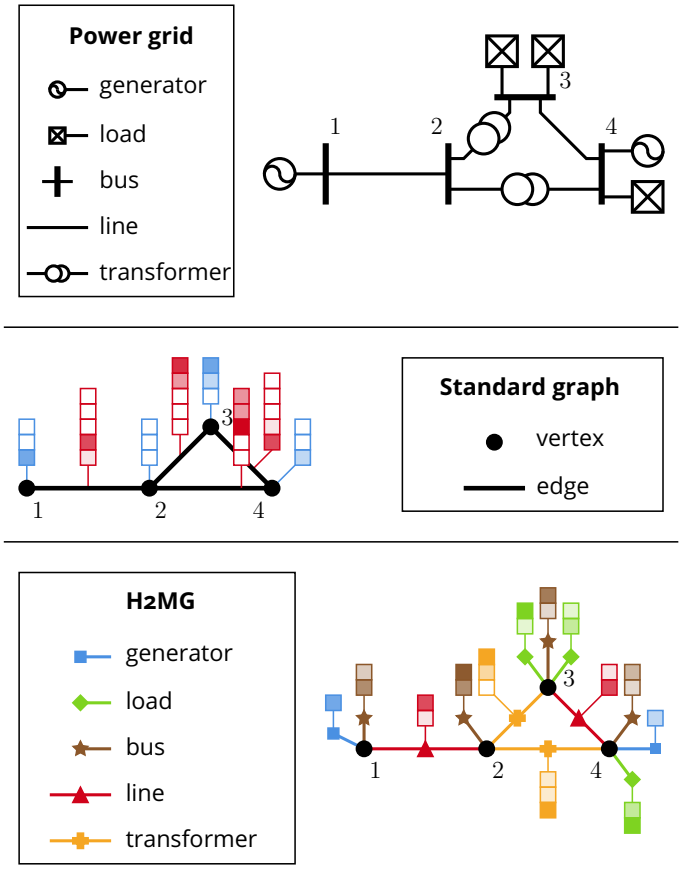


Figure 4.1: Power grid instance and its conversions into a H2MG, and into a H2MG. Standard graphs require to aggregate together vertex-like objects on the one hand and edge-like objects on the other hand. Meanwhile, H2MG allow to seamlessly represent power grids, without any information loss. In this example $\mathcal{C} = \{\text{generator}, \text{load}, \text{line}, \text{transformer}\}$. Input features are in the following dimensions: $d^{\text{gen},x} = 1, d^{\text{load},x} = 2, d^{\text{line},x} = 2, d^{\text{transfo},x} = 3$. Lines and transformers are of order 2, while generators and loads are of order 1. For the sake of readability, only input features are considered.

In addition to permutations that affect vertices, one may also consider permutations over classes $\mathcal{C}$, over ports $\mathcal{O}^c$ and over collocated objects of the same class $\mathcal{M}_e^c$. Those permutations are all commutative as they all affect orthogonal aspects of the data representation. However, these are not critical aspects of the problem. In the following, for the sake of simplicity, we shall only consider permutations over vertices.

Connections with standard graphs H2MGs allow for a more natural modelling of power grids and explicit the fact that they are composed of multiple classes of objects. The process of converting an heterogeneous graph into an homogeneous one constitutes a surjective but not injective mapping: aggregating multiple objects of the same class together causes information loss.

4.2 H2MGNN Architecture

In this section, we introduce the Hyper Heterogeneous Multi Graph Neural Network (H2MGNN) architecture to handle Hyper Heterogeneous Multi Graphs, and detail the main differences with the standard GNN introduced in Chapter 3. First, we provide a short introduction to the key ideas behind the Neural Ordinary Differential Equation (NODE) literature [140], that was inspirational for the H2MGNN architecture. We then define the H2MGNN architecture as a system of interacting entities. And finally, we detail the actual implementation of the proposed model.

4.2.1 Neural Ordinary Differential Equations

Before diving into the H2MGNN architecture, let us introduce some important concepts from the recent domain of NODE [140]. Consider a fully-connected neural network with T hidden layers, such that each hidden layer $h(t)$ is defined by the following recurrence equation:

$$h(t+1) = \boldsymbol{\phi}_{\boldsymbol{\theta}}^{\boldsymbol{t}}(h(t)) \tag{4.7}$$

where $(\boldsymbol{\phi}_{\boldsymbol{\theta}}^{\boldsymbol{t}})_{t=0,\dots,T-1}$ are trainable neural networks. The variable t represents the iteration step, and shall also abusively be referred to as time. If T is too large, the architecture is prone to vanishing gradient issues, and a possible solution is to use "skip connections" (ResNet) [23], that transform the previous equation into:

$$h(t+1) = h(t) + \boldsymbol{\phi}_{\boldsymbol{\theta}}^{\boldsymbol{t}}(h(t)) \tag{4.8}$$

This closely resembles a Euler scheme [141] used to solve the following differential equation over the interval $[0, T]$:

$$\frac{dh}{dt} = \boldsymbol{\phi}_{\boldsymbol{\theta}}(t, h(t)) \tag{4.9}$$

The key idea of NODEs is thus to rephrase deep neural networks as dynamical systems of latent variables. Inference amounts to solving the

differential equation (4.9) for the interval $[0, T]$. For the sake of simplicity, we shall equivalently consider the resolution of this differential equation for the interval $[0, 1]$, using a step size $\Delta t = 1/T$. In the NODE literature, the neural network mapping $\boldsymbol{\phi_\theta}$ is trained using an *adjoint sensitivity method* [142], which falls off the scope of the present work.

While our approach cannot be considered as part of the NODE literature, it still inherits its formulation and underlying ideas from this domain, as shown below.

4.2.2 Dynamical system

Inspired by the NODE literature, we propose to describe the H2MGNN architecture as a dynamical system of interacting entities. This continuous time dynamical system will then be discretized (see following subsection) so as to form the actual H2MGNN architecture.

We consider the following time-dependent variables:

- Vertex latent variables $(h_i^v)_{i\in[n]}$;
- Hyper-edge latent variables $(h_{e,m}^c)_{(c,e,m)\in\mathcal{G}_x}$;
- Hyper-edge outputs $(\hat{y}_{e,m}^c)_{(c,e,m)\in\mathcal{G}_x}$

Latent variables are in d dimensions – which is a hyperparameter of the architecture – while predictions are already in the required output dimensions.

Since latent variables have no intrinsic meaning, they are initialized with zero values. In contrast, predictions have a meaning with regards to the problem at hand, so it makes sense to initialize them at a trainable and class-dependent value $(\hat{y}_\theta^c)_{c\in\mathcal{C}}$.

$$\forall i \in [n], \qquad h_i^v(0) = [0, \dots, 0] \qquad \in \mathbb{R}^d \tag{4.10}$$

$$\forall (c, e, m) \in \mathcal{G}_x, \qquad h_{e,m}^c(0) = [0, \dots, 0] \qquad \in \mathbb{R}^d \tag{4.11}$$

$$\hat{y}_{e,m}^c(0) = \hat{y}_\theta^c \qquad \in \mathbb{R}^{d^{c,y}} \tag{4.12}$$

Latent variables interact according to the following differential system, involving trainable mappings ϕ_θ:

$$\forall i \in [n], \qquad \frac{dh_i^v}{dt} = \sum_{(c,e,m,o)\in\mathcal{N}_x(i)} \boldsymbol{\phi_\theta^{c,o}}(t, h_e^v, h_{e,m}^c, \hat{y}_{e,m}^c, x_{e,m}^c) \tag{4.13}$$

$$\forall (c, e, m) \in \mathcal{G}_x, \quad \frac{dh_{e,m}^c}{dt} = \boldsymbol{\phi_\theta^{c,h}}(t, h_e^v, h_{e,m}^c, \hat{y}_{e,m}^c, x_{e,m}^c) \tag{4.14}$$

$$\frac{d\hat{y}_{e,m}^c}{dt} = \boldsymbol{\phi_\theta^{c,y}}(t, h_e^v, h_{e,m}^c, \hat{y}_{e,m}^c, x_{e,m}^c) \tag{4.15}$$

where $h_e^v = (h_{e_o}^v)_{o\in\mathcal{O}^c}$ and should not be confused with $h_{e,m}^c$.

Vertices latent variables – equation (4.13) Vertices being connected at a varying amount of hyper-edges, their latent variables depend on a varying amount of unordered variables. We choose to aggregate their respective impacts by summing them. The action of each hyper-edge over its vertices depends on local information: its latent variable $h^c_{e,m}$, its prediction $\hat{y}^c_{e,m}$, its input $x^c_{e,m}$ and the latent variables of its vertices $h^v_e = (h^v_{e_o})_{o \in \mathcal{O}^c}$. Additionally, it is important to use a different trainable mapping depending on the hyper-edge class $c \in \mathcal{C}$ and on the port $o \in \mathcal{O}^c$ through which it is connected to the considered vertex. Thus, a set of trainable mappings $((\boldsymbol{\phi}^{\boldsymbol{c,o}}_{\boldsymbol{\theta}})_{o \in \mathcal{O}^c})_{c \in \mathcal{C}}$ is used in this step.

Hyper-edges latent variables – equation (4.14) The latent variable at hyper-edge (c, e, m) depends on itself $h^c_{e,m}$, on its local input $x^c_{e,m}$, on its local prediction $\hat{y}^c_{e,m}$, and on latent variables of its vertices $h^v_e = (h^v_{e_o})_{o \in \mathcal{O}^c}$. Thus, for a given class, all hyper-edges depend on the same amount of variables, which allows us to use a single trainable neural network $\boldsymbol{\phi}^{\boldsymbol{c,h}}_{\boldsymbol{\theta}}$.

Hyper-edges predictions – equation (4.15) Predictions are completely analogous to latent variables. The only difference is that their size is set by the problem at hand. Still, the formulation is analogous to equation (4.14), using this time the trainable neural network $\boldsymbol{\phi}^{\boldsymbol{c,y}}_{\boldsymbol{\theta}}$

4.2.3 Architecture

The H2MGNN architecture amounts to integrating the dynamical system of equations (4.10 - 4.15) over the interval $[0, 1]$. The final output serves as a prediction. Contrarily to NODE models, it relies on a simple Euler scheme by using a fixed time step size Δt, which implicitely define the amount of steps (given by $1/\Delta t$). Thus, our method is a recurrent and residual GNN architecture, which shall be trained using standard back-propagation methods. Implementing an actual NODE should be investigated in future work.

Algorithm 2 details the pseudo-code of the H2MGNN architecture. Trainable neural network blocks $((\boldsymbol{\phi}^{\boldsymbol{c,o}}_{\boldsymbol{\theta}})_{o \in \mathcal{O}^c}, \boldsymbol{\phi}^{\boldsymbol{c,h}}_{\boldsymbol{\theta}}, \boldsymbol{\phi}^{\boldsymbol{c,y}}_{\boldsymbol{\theta}})_{c \in \mathcal{C}}$ can be implemented as simple fully-connected neural networks. Contrarily to the architecture presented in Algorithm 1, the trainable mappings take the continuous time variable t as input. Hyperparameters of the architecture include the latent dimension d, the time step size Δt, and all the hyperparameters of each trainable neural network block. Unless explicitly stated otherwise, this architecture is used by default in the experiments.

Algorithm 2 Proposed Heterogeneous Graph Neural Network

1: **procedure** $\boldsymbol{f_\theta}(x = (x^c_{e,m})_{(c,e,m)\in\mathcal{G}_x})$
2:
3: ▷ Initialization
4: **for** $i \in [n]$ **do**
5: $h^v_i \leftarrow 0^d$
6: **for** $(c, e, m) \in \mathcal{G}_x$ **do**
7: $h^c_{e,m} \leftarrow 0^d$
8: $\hat{y}^c_{e,m} \leftarrow \hat{y}^c_\theta$
9:
10: ▷ Latent interaction
11: $t \leftarrow 0$
12: **while** $t < 1$ **do**
13: **for** $i \in [n]$ **do**
14: $h^v_i \leftarrow h^v_i + \Delta t \times \sum_{(c,e,m,o)\in\mathcal{N}_x(i)} \boldsymbol{\phi^{c,o}_\theta}(t, h^v_e, h^c_{e,m}, \hat{y}^c_{e,m}, x^c_{e,m})$
15: **for** $(c, e, m) \in \mathcal{G}_x$ **do**
16: $h^c_{e,m} \leftarrow h^c_{e,m} + \Delta t \times \boldsymbol{\phi^{c,h}_\theta}(t, h^v_e, h^c_{e,m}, \hat{y}^c_{e,m}, x^c_{e,m})$
17: $\hat{y}^c_{e,m} \leftarrow \hat{y}^c_{e,m} + \Delta t \times \boldsymbol{\phi^{c,y}_\theta}(t, h^v_l, h^c_{e,m}, \hat{y}^c_{e,m}, x^c_{e,m})$
18: $t \leftarrow t + \Delta t$
19:
20: **return** $\hat{y} = (\hat{y}^c_{e,m})_{(c,e,m)\in\mathcal{G}_x}$

In order to make things clearer, we propose to illustrate the algorithm on a simple case. Consider the input graph displayed in Figure 4.2. It has 2 vertices, and one instance of each of the following classes: $\mathcal{C} = \{\alpha, \beta, \gamma\}$. We observe that the class β is of order 2, while others are of order 1. Moreover, there are no collocated hyper-edges of the same class, we may thus drop the index m to simplify notations. An iteration of the algorithm is made of the three following steps:

- Figure 4.3 displays what happens when the latent variables of vertices is updated (line 14 of Algorithm 2). Each vertex is updated according to the information contained in the hyper-edges connected to it.
- Figure 4.4 displays the process of updating latent variables at hyper-edges. They are updated according to the information contained in the hyper-edge, and to the $|\mathcal{O}^c|$ ordered vertices to which the hyper-edge is connected.
- Figure 4.5 displays the process of updating output at hyper-

edges. This step is akin to the previous one, but uses a different trainable mapping.

These figures underline the asymmetry between vertices and hyper-edges. Vertices can be seen as interfaces through which hyper-edges can interact, thus propagating their respective influence to the whole graph.

4.2.4 Inference complexity

Assuming that each neural network block has a single hidden layer with dimension d, that d is much larger than input and output feature dimensions and denoting by u the average neighborhood size, one inference has computational complexity of order $\mathcal{O}(und^3/\Delta t)$, scaling linearly with n. Furthermore, many problems involve very local interactions, resulting in small u. However, one should keep in mind that hyperparameters $1/\Delta t$ and d should be chosen according to the charateristics of distribution $p(x)$, as detailed in Chapter 6. For instance, the step size Δt should be appropriate with regards to the diameters of considered graphs, so as to allow information to propagate enough times.

4.3 Implementation considerations

The H2MGNN architecture requires additional attention in order to actually work seamlessly. Because both inputs and outputs of the neural networks actually have a physical meaning, and can thus have atypical distributions, it is important to add some pre and post processing. Moreover, if no precautions are taken, the architecture is prone to numerical instability, which we address by using normalization layers.

4.3.1 Input pre-processing

Neural networks are known to be especially sensitive to input data distribution. It is thus generally recommended to pre-process the data so that the values are approximately contained in the range $[-1, 1]$. We thus propose to use a pre-processing function $\zeta : \mathcal{X} \to \mathcal{X}$.

Input of this pre-processing mapping are physically meaningful quantities. Thus, normalizing them distorts the input space and shatters their physical meaning. Figure 4.6 details how the pre-processing function ζ is included in the whole process.

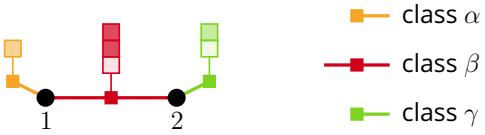


Figure 4.2: Instance of an input heterogeneous graph x.

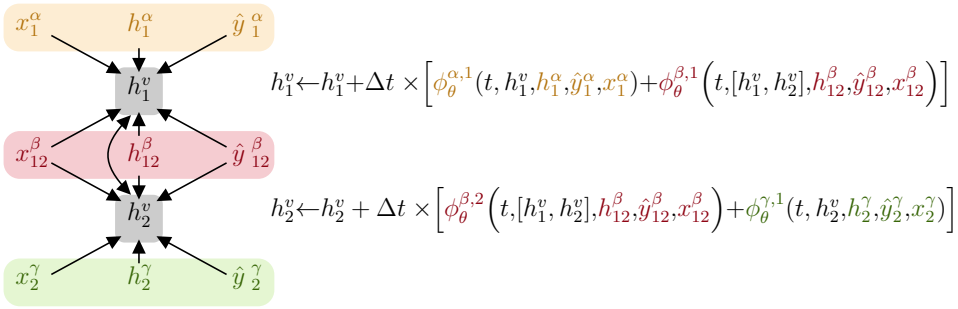


Figure 4.3: Updating vertex latent variables.

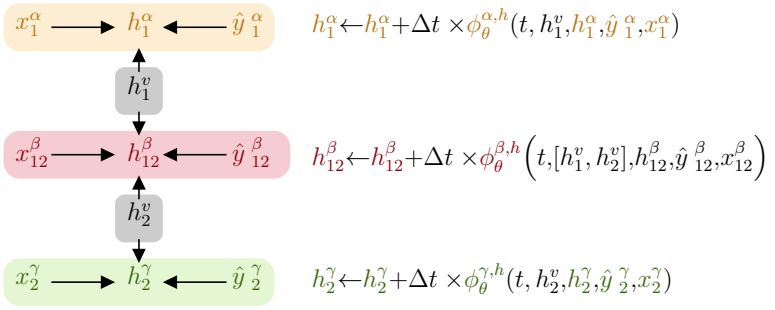


Figure 4.4: Updating hyper-edge latent variables.

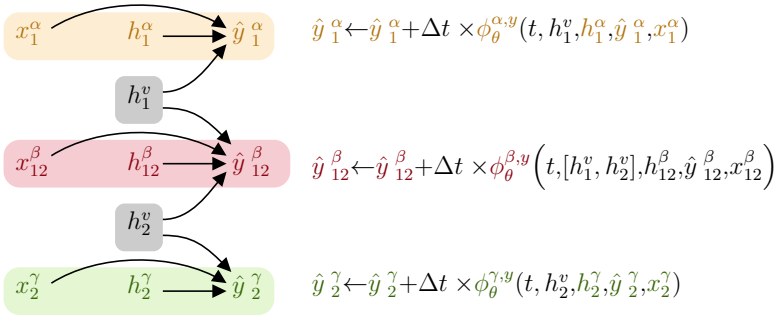


Figure 4.5: Updating hyper-edge outputs.

Since this mapping serves as a pre-processing step to a permutation-equivariant structure, it is preferable to have $\boldsymbol{\zeta}$ also be permutation-equivariant, so as not to break the input graph structure. We propose to decompose it as $\boldsymbol{\zeta} = (\boldsymbol{\zeta}^c)_{c\in\mathcal{C}}$, such that each class has its own pre-processing function. Such class-specific functions are then applied identically to each instance of each class. We denote by $\tilde{x}$ the pre-processed version of the input data x, which is then fed to the neural network.

$$\tilde{x} = (\boldsymbol{\zeta}^c(x^c_{e,m}))_{(c,e,m)\in\mathcal{G}_x} \tag{4.16}$$

This function is built before the start of the training process and so that distributions of all input features of all classes are well-distributed in the training set.

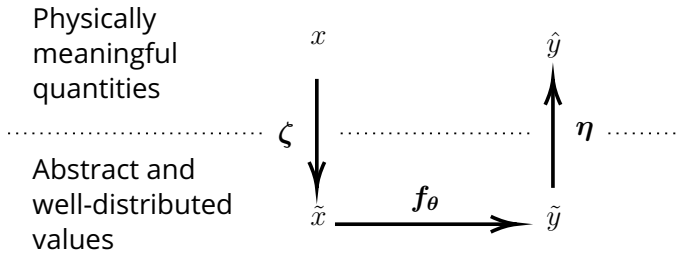


Figure 4.6: Data pre-processing $\boldsymbol{\zeta}$ and post-processing $\boldsymbol{\eta}$ allows to convert physically meaningful and poorly-distributed data, to abstract and better-distributed (see text) quantities. $\boldsymbol{\zeta}$ and $\boldsymbol{\eta}$ being permutation-equivariant, we have that $\boldsymbol{\eta} \circ \boldsymbol{f_\theta} \circ \boldsymbol{\zeta}$ is also permutation-equivariant.

4.3.2 Output post-processing

Neural networks also have difficulties predicting quantities that have either too large or too small orders of magnitudes. Thus, manually scaling the output of the neural network $\boldsymbol{f_\theta}$ can greatly facilitate the learning process. We propose to apply another function $\boldsymbol{\eta} : \mathcal{Y} \to \mathcal{Y}$ to convert the abstract output of neural networks into physically meaningful quantities. Figure 4.6 shows of the post-processing function $\boldsymbol{\eta}$ fits into the whole framework.

Similarly to the pre-processing mapping, it is important to have a permutation-invariant function. We decompose the post-processing as $\boldsymbol{\eta} = (\boldsymbol{\eta}^c)_{c\in\mathcal{C}}$, such that each class has its own post-processing function. Such mappings are applied identically to each instance of each class. We denote by $\tilde{y} = \boldsymbol{f_\theta}(\tilde{x})$ the raw output of the neural network,

and by $\hat{y}$ the actual post-processed output data, which is defined by:

$$\hat{y} = (\boldsymbol{\eta}^{\boldsymbol{c}}(\tilde{y}^c_{e,m}))_{(c,e,m)\in\mathcal{G}_x} \tag{4.17}$$

The post-processing function can also be used to initialize the output of the neural network at the beginning of the training, or even to manually force predictions to remain within an acceptable range of values.

This function is built before the start of the training process, and requires some knowledge about the problem at hand.

4.3.3 Numerical stability

Regardless of pre and post-processing considerations, the proposed architecture is prone to exploding or vanishing gradient issues. The recurrent formulation tends to create numerical instabilities. We normalize all abstract latent variables before feeding them to neural network using the following mapping[1]:

$$u \mapsto \frac{u}{\|u\| + 1} \tag{4.18}$$

This allows neural networks inputs to remain in a reasonable range. Contrarily to other approaches such as batch-normalization [143], this mapping is local and does not require to compute statistics over the whole batch of data. It preserves the permutation-equivariant property of the GNN architecture.

This simple trick, combined with suitable pre and post processing functions significantly improve the stability and efficiency of the training process.

Throughout this chapter, we have introduced a formalism that can model power grids without any information loss, and then detailed a neural network architecture called H2MGNN that is able to seamlessly process such structures. In the next chapter, we develop a methodology to train such neural network mappings to solve optimization problems.

[1] as suggested by our colleague Dr. Victor Berger

4.A Historical DSS architecture

The data formalism and neural network architecture introduced in the present chapter result from a process of iterative refinements, . While experiments from Sections 7.1 and 8.2 rely on the latest version of the approach, Sections 7.2, 8.1 and 9.1 detail experiments conducted using an earlier version of both the data formalism and neural network architecture, which we detail in the present appendix section.

Graph data Previous versions used to consider traditional homogeneous graphs as input and output. Moreover, only vertex features were considered as output. Using similar notations as Chapter 3, the data was denoted by $x = (x^v, x^e)$, and the output by $y = (y^v)$. The notion of neighborhood was quite straightforward and denoted by $\mathcal{N}_x(i) := \{j \in [n] | (i,j) \in E \text{ or } (j,i) \in E\}$. Moreover, we used the following convention: $\mathcal{N}_x^*(i) = \mathcal{N}(i;x) \backslash i$.

Architecture Previously used GNN architectures were not recurrent: they iteratively updated latent variables using each time a different set of trainable neural network mappings $(\boldsymbol{\phi}^{\boldsymbol{t}}_{\rightarrow,\boldsymbol{\theta}}, \boldsymbol{\phi}^{\boldsymbol{t}}_{\leftarrow,\boldsymbol{\theta}}, \boldsymbol{\phi}^{\boldsymbol{t}}_{\circlearrowleft,\boldsymbol{\theta}}, \boldsymbol{\psi}^{\boldsymbol{t}}_{\boldsymbol{\theta}}, \boldsymbol{\xi}^{\boldsymbol{t}}_{\boldsymbol{\theta}})_{t=1,\dots,T}$, as detailed in Algorithm 3. Instead of producing a single output similarly to the current version, it used to output T different outputs, which are then all used during training, as illustrated in Figure 4.7. The coef-

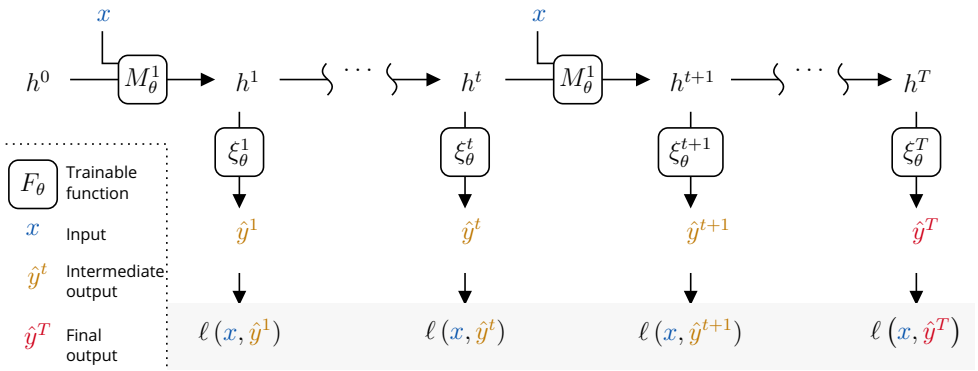


Figure 4.7: Previously used neural network architecture. The operator M^t_θ encompasses all the message passing operations performed in lines (10-13) of Algorithm 3. All intermediate predictions appear in the loss of equation 4.19, while the current H2MGNN only penalizes the final prediction.

Algorithm 3 Previously used Graph Neural Network architecture

1: **procedure** $\boldsymbol{f_\theta}(x = (x^v, x^e))$
2:
3: ▷ Initialization
4: **for** $i \in [n]$ **do**
5: $h_i^0 \leftarrow 0^d$
6:
7: ▷ Message passing
8: **for** $t = 1, \dots, T$ **do**
9: **for** $i \in [n]$ **do**
10: $\phi^t_{\rightarrow,i} \leftarrow \sum_{j \in \mathcal{N}^*_x(i)} \boldsymbol{\phi^t_{\rightarrow,\theta}}(h_i^{t-1}, x^e_{ij}, h_j^{t-1})$
11: $\phi^t_{\leftarrow,i} \leftarrow \sum_{j \in \mathcal{N}^*_x(i)} \boldsymbol{\phi^t_{\leftarrow,\theta}}(h_i^{t-1}, x^e_{ji}, h_j^{t-1})$
12: $\phi^t_{\circlearrowleft,i} \leftarrow \boldsymbol{\phi^t_{\circlearrowleft,\theta}}(h_i^{t-1}, x^e_{ii})$
13: $h_i^t \leftarrow h_i^{t-1} + \alpha \times \boldsymbol{\psi^t_\theta}(h_i^{t-1}, x_i, \phi^t_{\rightarrow,i}, \phi^t_{\leftarrow,i}, \phi^t_{\circlearrowleft,i})$
14: $\hat{y}_i^t \leftarrow \boldsymbol{\xi^t_\theta}(h_i^t)$
15:
16: **return** $(\hat{y}^t)_{t=1,\dots,T}$

ficient α in line 13 of Algorithm 3 is a scaling factor that improved the numerical stability.

Training The training loss consists in a discounted sum over the T different costs of intermediate predictions[2], using a discount factor $\gamma \in [0, 1]$:

$$\sum_{t=1}^{T} \gamma^{T-t} \ell(x, \hat{y}^t) \tag{4.19}$$

This loss, combined with the use of gradient clipping and a reasonably small α, allowed to mitigate a recurring problem of gradient explosion. Since the latest H2MGNN version does not suffer from such issues thanks to the tricks introduced in Section 4.3, there is little benefit in using a discounted sum.

[2] ℓ is the cost function that is to be minimized by the neural network mapping, as will be further detailed and justified in Chapter 5

Chapter 5

Statistical Solver Problems

Power grid operation involves various forms of optimization processes: power flows are estimated by solving equations derived from Kirchhoff's laws, generators voltage setpoints and interconnection patterns are controlled by dispatchers to optimize a security objective, power generation dispatch is dictated by a market equilibrium, etc. Whether these quantities are obtained through a well-posed optimization program, or left to human decisions, there is always some form of optimization, which creates a coupling between various quantities throughout the grid.

In this chapter, we propose a method to train neural networks such as introduced in Chapter 4 to solve instances of an optimization problem – referred to as the "target" problem – through the definition of a Statistical Solver Problem (SSP). Instead of learning to imitate another resolution method, the proposed methodology amounts to training a neural network in an unsupervised fashion, by directly minimizing the cost function of the target problem. In that sense, the proposed approach, which we call Deep Statistical Solver (DSS), is a full-fledged optimization process. It relies on similar ideas as those underlying the Physics-Informed Neural Network (PINN) literature.

In a first section, we consider the case of single-level unconstrained optimization problems. For instance, it includes the case of the AC Power Flow (AC-PF) problem, which consists in computing power flows throughout the grid, given power injections and transmission lines interconnections. In a second section, we consider the case of bilevel optimization problems, of which the issue of controlling the voltage in power grids is a typical instance. It consists in choosing the voltage setpoints of generators while anticipating that this decision has a non trivial impact over the power flows.

5.1 Single-level unconstrained optimization

Let us consider the case of single-level unconstrained optimization target problems, through the example of the AC-PF problem. It consists in computing the voltage magnitudes and phase angles (which can then be used to compute power flows), given the power injections and the transmission lines interconnections. This problem can be schematically represented by Figure 5.1.

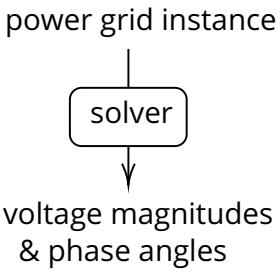


Figure 5.1: Single level AC-PF problem

In current power grid operations, the solver that solves the AC-PF relies on a Newton-Raphson method. Inspired by the PINN approach to the resolution of partial differential equation [95], we propose an alternative solution to the classical Newton-Raphson solver using neural networks mapping such as introduced in Chapter 4 to solve this optimization problem without the need for any existing solution of sample instances. Such general approach could be used for any permutation-equivariant optimization problem defined on graphs, including the ones for which there is no existing satisfying solver.

Thus, we propose to convert the target optimization problem as a learning problem, which we refer to as a Statistical Solver Problem (SSP). We derive this neural network training procedure from an elementary density estimation principle. We call a deep neural network trained using this approach a Deep Statistical Solver (DSS).

Throughout this section, we will use the AC-PF as running example, so as to make things concrete. The present theoretical section is further supported by experimental results presented in both Chapters 7 and 8.

5.1.1 Target optimization problem

Let $\mathcal{X}$ and $\mathcal{Y}$ be two sets, and $\boldsymbol{\ell} : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}$ be a cost function. We assume that for a given $x \in \mathcal{X}$, $\boldsymbol{\ell}(x, y)$ has a unique minimum $y^*(x)$

defined by the following target optimization problem:

$$y^*(x) = \arg\min_{y \in \mathcal{Y}} \boldsymbol{\ell}(x, y) \tag{5.1}$$

The cost function $\boldsymbol{\ell}$ being fixed, we refer to x as an instance of the optimization problem.

In the AC-PF problem, x denotes a H2MG input (see Section 4.1) that encapsulates the power production and consumption, and transmission lines interconnections (see Figure 4.1 for instance); and y denotes the voltage magnitudes and phase angles at all buses. The cost function ℓ is the violation of Kirchhoff's laws. This target optimization problem is usually solved using the Newton-Raphson method.

5.1.2 Probabilistic relaxation

Traditional optimization does not use any probability distribution on the instance space. On the contrary, statistical learning tools are designed to exploit any such distribution. However, we first need to rephrase the optimization problem in probabilistic terms, and then rely on a density estimation approach to give rise to a statistical learning problem.

Target distribution

In power grid operations, each possible power grid instance x is associated with a certain probability density $p(x)$. Some instances might be more probable than others. In such a case, rare but critical instances may be neglected by our training algorithm, which may result in our model being unreliable in abnormal situations. Still, one could choose to distort the real probability density by overweighting critical instances of optimization problems, so as to make the model robust to rare events. The issue of choosing the appropriate probability distribution p in regard with industrial security constraints is not addressed in this document, but should definitely be considered in future work.

The conditional probability distribution $q(y|x)$ that connects problem instances x to their solutions is a Dirac measure located at $y^*(x)$:

$$q(y|x) = \delta_{y^*(x)}(y) \tag{5.2}$$

The target distribution is as follows:

- $x \sim p(x)$: sampling an instance of optimization problem x ;
- $y \sim q(y|x)$: solving the instance of optimization problem x.

Writing the optimization problem in probabilistic terms allows us to derive a training loss based on density estimation principles and to learn probability q, i.e., to learn a mapping between instances of optimization problems x and their solutions $y^*(x)$. Indeed, q is not directly reachable, and we will now derive rigorously such a training loss that allows us to learn it.

Deriving a learning problem

We consider a set of distributions $q_\theta(y|x)$ parameterized by $\theta \in \Theta$. We need to find θ^* such that $p(x)q_{\theta^*}(y|x)$ is as close as possible to the unknown true generative process $p(x)q(y|x)$.

It is common in the statistical learning literature [107] to minimize the Kullback-Leibler divergence between $p(x)q_\theta(y|x)$ and $p(x)q(y|x)$.

$$D_{KL}(pq_\theta \| pq) := \int_{x\in\mathcal{X}} \int_{y\in\mathcal{Y}} p(x)q_\theta(y|x) \log\left(\frac{q_\theta(y|x)}{q(y|x)}\right) \tag{5.3}$$

The KL-divergence is not a distance per-se, but is a commonly used metrics for the similarity between two probability distributions. Unfortunately, this quantity is not defined in our case. The support of $p(x)q(y|x)$ – also denoted by $supp(p(x)q(y|x)) := \{(x,y) \mid p(x)q(y|x) > 0\}$ – is limited to a manifold strictly included in the support of $p(x)q_\theta(y|x)$. As a consequence we have that $D_{KL}(pq_\theta\|pq) = +\infty$, which cannot be minimized.

In the following, we propose to consider a relaxation of the target distribution for which the KL-divergence can be written. This relaxation is controlled by a parameter $\beta \in \mathbb{R}^+\backslash\{0\}$. We first consider the problem of learning intermediate distributions, and we then investigate the limit of such intermediate learning problems when we make the relaxation disappear.

Assuming that for any fixed x, $\boldsymbol{\ell}$ has a unique minimum, and that for any β we have $\int_{y'\in\mathcal{Y}} e^{-\beta\boldsymbol{\ell}(x,y')} < +\infty$, then q can be written as the weak limit of a set of Gibbs measures [144]:

$$q(y|x) = \lim_{\beta\to+\infty} q_\beta(y|x) \qquad ; \qquad q_\beta(y|x) = \frac{e^{-\beta\boldsymbol{\ell}(x,y)}}{\int_{y'\in\mathcal{Y}} e^{-\beta\boldsymbol{\ell}(x,y')}} \tag{5.4}$$

We observe that for any $\beta > 0$, $supp(p(x)q_\beta(y|x)) = supp(p(x)) \times \mathcal{Y}$. Thus, the KL-divergence between the parameterized generative process and each intermediate Gibbs distribution is well defined:

$$D_{KL}(pq_\theta||pq_\beta) = -\,\mathbb{E}_{x\sim p(x)}\left[H_y(q_\theta(y|x))\right] + \beta\,\mathbb{E}_{\substack{x\sim p(x)\\ y\sim q_\theta(y|x)}}\left[\boldsymbol{\ell}(x,y)\right] + C(\beta) \tag{5.5}$$

where $H_y(.)$ is the Shannon entropy of a distribution *w.r.t.* variable y, and C a function of β independent from θ. For a given β, we consider the following intermediate statistical learning problem:

$$\Theta^*(\beta) = \arg\min_{\theta\in\Theta} D_{KL}(pq_\theta||pq_\beta) \tag{5.6}$$

Theorem 1. *Let $(\beta_k)_{k\in\mathbb{N}}$ be a positive sequence such that $\beta_k \underset{k\to+\infty}{\longrightarrow} +\infty$. If q_θ is continuous w.r.t. θ, Θ is compact, and $\theta \mapsto \mathbb{E}_{\substack{x\sim p(x)\\ y\sim q_\theta(y|x)}} [\boldsymbol{\ell}(x,y)]$ has a unique minimizer θ^*, then any sequence $\theta_k \in \Theta^*(\beta_k)$ converges to θ^*.*

Proof. Let us denote by $\boldsymbol{g_k}$ the mapping

$$\boldsymbol{g_k} : \theta \mapsto -\frac{1}{\beta_k}\mathbb{E}_{x\sim p(x)}\left[H_y(q_\theta(y|x))\right] + \mathbb{E}_{\substack{x\sim p(x)\\ y\sim q_\theta(y|x)}}[\boldsymbol{\ell}(x,y)] \tag{5.7}$$

Its $\arg\min$ is exactly $\Theta^*(\beta_k)$. We observe that it converges uniformly to the mapping f:

$$\boldsymbol{g} : \theta \mapsto -\mathbb{E}_{\substack{x\sim p(x)\\ y\sim q_\theta(y|x)}}[\boldsymbol{\ell}(x,y)] \tag{5.8}$$

θ^* is the minimizer of $\boldsymbol{g}$ and is assumed to be unique. Consider any convergent subsequence of $\theta_k \in \Theta^*(\beta_k)$, and let us denote θ' its limit. Then for any $\theta \in \Theta$ we have that $\boldsymbol{g_k}(\theta_k) \leq \boldsymbol{g_k}(\theta)$. Since $\boldsymbol{g_k}$ converges uniformly towards $\boldsymbol{g}$ and θ_k converges to θ', we obtain that

$$\boldsymbol{g_k}(\theta_k) \to \boldsymbol{g}(\theta') \qquad\qquad \boldsymbol{g_k}(\theta) \to \boldsymbol{g}(\theta) \tag{5.9}$$

As a result, we have that $\boldsymbol{g}(\theta') \leq \boldsymbol{g}(\theta)$, which holds for any θ. By unicity of θ^*, we have that $\theta' = \theta^*$. Thus any convergent subsequence of $\theta^k \in \Theta^*(\beta_k)$ converges to θ^*, and hence the whole sequence converges to θ^* as well. □

Theorem 1 states that if $\theta \mapsto \mathbb{E}_{\substack{x\sim p(x)\\ y\sim q_\theta(y|x)}}[\boldsymbol{\ell}(x,y)]$ has a unique minimizer θ^*, then the limit of the intermediate statistical learning problems $\Theta^*(\beta)$ as β tends to infinity amounts to solving the following problem:

$$\theta^* = \arg\min_{\theta\in\Theta} \mathbb{E}_{\substack{x\sim p(x)\\ y\sim q_\theta(y|x)}}[\boldsymbol{\ell}(x,y)] \tag{5.10}$$

Even though we were unable to compute the KL-divergence between our target distribution and our parameterized distribution, we managed to take an intermediate path to derive a proper learning problem.

Statistical Solver Problem The resulting learning problem is referred to as an SSP, and can be summed up as follows: given spaces $\mathcal{X}$ and $\mathcal{Y}$, distribution p defined over $\mathcal{X}$ and cost function ℓ defined on $supp(p) \times \mathcal{Y}$, we wish to find the distribution $q_\theta(y|x)$ that minimizes the cost function $\boldsymbol{\ell}$, when x follows $p(x)$. It is schematically represented by Figure 5.2.

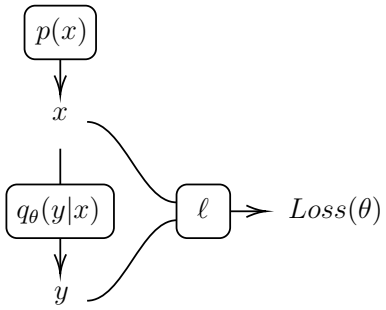


Figure 5.2: Single level SSP

5.1.3 Training algorithm

The loss function associated with the SSP can be expressed as follows:

$$Loss(\theta) = \mathbb{E}_{\substack{x\sim p(x) \\ y\sim q_\theta(y|x)}} \left[\boldsymbol{\ell}(x, y)\right] \tag{5.11}$$

Indeed, we do not have a direct access to the distribution $p(x)$, and have to rely on a empirical dataset $D = \{x_m\}_{m\in M}$. The empirical loss over the dataset D is given by:

$$Loss(\theta; D) = \frac{1}{|M|} \sum_{m\in M} \mathbb{E}_{y\sim q_\theta(y|x_m)} \left[\boldsymbol{\ell}(x_m, y)\right] \tag{5.12}$$

In the AC-PF problem, the cost function ℓ is the violation of Kirchhoff's laws, which is differentiable with regards to y. Moreover, in the experiments we choose $q_\theta(y|x)$ to be a deterministic mapping, instantiated as a trainable H2MGNN function $\boldsymbol{f_\theta(x)}$ such as introduced in Chapter 4. The loss shall thus be rephrased as

$$Loss(\theta; D) = \frac{1}{|M|} \sum_{m\in M} \boldsymbol{\ell}(x_m, \boldsymbol{f_\theta}(x_m)) \tag{5.13}$$

The gradient $\nabla_\theta Loss(\theta; D)$ can be estimated using automatic differentiation. The H2MGNN $\boldsymbol{f_\theta}$ is trained through a standard gradient-descent method, as described in Algorithm 4. Indeed, advanced gradient descent techniques introduced in Chapter 2 can also be used. This approach is experimentally validated in Chapters 7 and 8.

Algorithm 4 Training a single-level Statistical Solver

procedure Train($D = \{x_m\}_{m \in M}, N_{epochs}$)
  Initialize θ
  for $epoch = 1, \dots, N_{epochs}$ **do**
    $\nabla_\theta Loss(\theta; D) \leftarrow \frac{1}{|M|} \sum_{m \in M} \nabla_\theta \boldsymbol{\ell}(x_m, \boldsymbol{f_\theta}(x_m))$
    $\theta \leftarrow \theta - \alpha \nabla_\theta Loss(\theta; D)$
  return $\boldsymbol{f_\theta}$

5.2 Bilevel optimization

Bracken and McGill [145] define bilevel optimization as

> *"mathematical programs that contain an optimization problem in the constraints."*

This type of optimization process occurs when two interacting entities make decisions sequentially. The first to make a decision is called the "leader". The second is called the "follower", and knows the leader's decision. Thus, in order to optimize its objective function, the leader has to anticipate the impact of its own decision over the follower's behavior.

In power grids operation, a typical instance of such a structure arises in the voltage control problem. In order to keep the power grid in security, dispatchers can control voltage setpoints of certain generators. Still, the actual outcome of their actions is not trivial, and is given by the solution of the AC-PF. In order to optimize their own objectives, dispatchers have to anticipate the impact of their actions over the outcome of another optimization problem. This bilevel optimization problem is schematically represented in Figure 5.3, where the term "controller" denotes the dispatcher. We thus wish to train a neural network such as introduced in Chapter 4 to control generators voltage setpoints so as to optimize a security-related objective.

In line with the previous section, we propose to derive a training procedure from the aforementioned target optimization problem. It relies on the training of two distinct neural networks, one playing the role of the controller, and the other playing the role of the solver. This training strategy is in some way similar to the training of the two models that are used in Generative Adversarial Networks (GANs) [24]. We instantiate this section on the voltage control problem to keep things concrete, although the approach is as general as possible and could be applied to a wide variety of domains. Chapter 9 presents preliminary experiments on a rather small power grid.

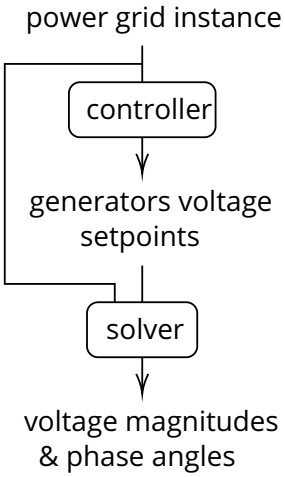


Figure 5.3: Bilevel voltage control problem

5.2.1 Target optimization problem

Let $\mathcal{X}, \mathcal{Y}, \mathcal{Z}$ be three sets, and $\boldsymbol{\ell}, \boldsymbol{\ell'}$ be two cost functions $\mathcal{X} \times \mathcal{Y} \times \mathcal{Z} \to \mathbb{R}$. We consider the problem of finding the element of $\mathcal{Y}$ that minimizes the cost function $\boldsymbol{\ell}$ for a certain element $x \in \mathcal{X}$. Similarly to the previous section, we assume that there are no equality or inequality constraints other than the nested optimization problem. We assume that for any fixed x, y, z, $y' \mapsto \boldsymbol{\ell}(x, y', z)$ and $z' \mapsto \boldsymbol{\ell'}(x, y, z')$ both have a unique minimum. We consider the following bilevel optimization problem:

$$y^*(x) = \underset{y \in \mathcal{Y}}{\arg\min}\, \boldsymbol{\ell}(x, y, z) \tag{5.14}$$

$$\text{subject to } z = z^*(x, y) = \underset{z \in \mathcal{Z}}{\arg\min}\, \boldsymbol{\ell'}(x, y, z) \tag{5.15}$$

Equation (5.14) defines the "upper level" problem, while equation (5.15) defines the "lower level" problem.

In our voltage control example, we denote by x the heterogeneous graph containing power production and consumption everywhere on the grid, as well as transmission lines interconnections, by y the voltage setpoints at some buses controlled by the dispatcher, and by z the voltage magnitudes and phase angles at each bus. $\boldsymbol{\ell}$ computes a security-related objective (for instance minimizing Joule losses while keeping voltage magnitudes in an acceptable range), and $\boldsymbol{\ell'}$ computes the AC-PF by minimizing the violation of Kirchhoff's laws. The problem amounts to maximizing the security of the grid, by anticipating the (non-linear) impact of the dispatcher's decision over the actual state of the system. The dispatcher is the leader, and "physics" is the follower.

Target distribution

Similarly to the previous section, we associate a probability distribution $p(x)$ to the set $\mathcal{X}$. The conditional probability $q(y|x)$ maps instances of the upper level optimization problems to their solution, while $r(z|x,y)$ maps instances of the lower level optimization problem to their solutions.

$$q(y|x) = \delta_{y^*(x)}(y) \tag{5.16}$$
$$r(z|x,y) = \delta_{z^*(x,y)}(z) \tag{5.17}$$

The target distribution is the following:

- $x \sim p(x)$: sampling an optimization problem instance x;
- $y \sim q(y|x)$: solving the upper level optimization problem;
- $z \sim r(z|x,y)$: solving the lower level optimization problem.

We now proceed as in Section 5.1.

Deriving a learning problem

Our only goal here is to learn an approximation of the distribution $q(y|x)$, using a parameterized distribution $q_\theta(y|x)$. Similarly to the previous section, we propose to consider the Kullback-Leibler divergence between pq_θ and intermediate Gibbs measures pq_β. Assuming that for any fixed x, $\boldsymbol{\ell}$ has a unique minimum, and $\forall \beta, \int_{y' \in \mathcal{Y}} e^{-\beta \boldsymbol{\ell}(x,y')} < +\infty$, then q can be written as the weak limit of a set of Gibbs measures:

$$q(y|x) = \lim_{\beta \to +\infty} q_\beta(y|x) \quad ; \quad q_\beta(y|x) = \frac{\int_{z \in \mathcal{Z}} r(z|x,y) e^{-\beta \boldsymbol{\ell}(x,y,z)}}{\int_{y' \in \mathcal{Y}} \int_{z \in \mathcal{Z}} r(z|x,y') e^{-\beta \boldsymbol{\ell}(x,y',z)}} \tag{5.18}$$

As it appears, the intermediate distribution q_β relies on the lower level distribution r. Following the same reasoning as in the previous section, and using the fact that r is a Dirac measure (equation 5.17), we obtain the following upper-level Statistical Solver Problem (SSP):

$$\theta^* = \arg\min_{\theta \in \Theta} \mathbb{E}_{\substack{x \sim p(x) \\ y \sim q_\theta(y|x) \\ z \sim r(z|x,y)}} [\boldsymbol{\ell}(x,y,z)] \tag{5.19}$$

Estimating θ^* requires the actual lower level distribution r. Unfortunately, it is neither possible to sample from it, nor to estimate it.

We propose to learn an approximation of $r(z|x,y)$ using a distribution $r_\omega(z|x,y)$ parameterized by $\omega \in \Omega$. During its training, q_θ is very

likely to sample values that are far from the support of q. Even for those "erroneous" predictions, r_ω should remain a valid approximation of r. Thus, one should choose a distribution $\overline{q}_\theta(y|x)$ that covers q_θ. In the experiments, we define $\overline{q}_\theta(y|x)$ by simply adding a Gaussian noise to $q_\theta(y|x)$. We consider the following lower-level Statistical Solver Problem:

$$\omega^* = \arg\min_{\omega \in \Omega} \mathbb{E}_{\substack{x\sim p(x) \\ y\sim \overline{q}_\theta(y|x) \\ z\sim r_\omega(z|x,y)}} [\boldsymbol{\ell'}(x, y, z)] \tag{5.20}$$

Equations (5.19) and (5.20) define the bilevel Statistical Solver Problem, and Figure 5.4 illustrate it.

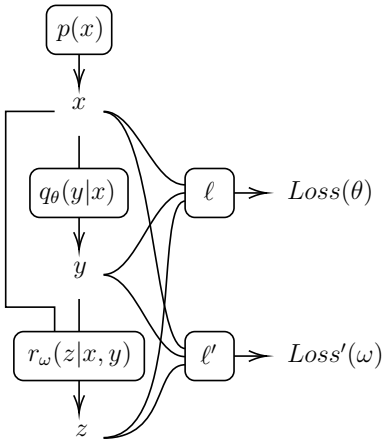


Figure 5.4: Bilevel SSP

5.2.2 Training algorithm

We consider the following loss functions:

$$Loss(\theta) = \mathbb{E}_{\substack{x\sim p(x) \\ y\sim q_\theta(y|x) \\ z\sim r(z|x,y)}} [\boldsymbol{\ell}(x, y, z)] \tag{5.21}$$

$$Loss'(\omega) = \mathbb{E}_{\substack{x\sim p(x) \\ y\sim \overline{q}_\theta(y|x) \\ z\sim r_\omega(z|x,y)}} [\boldsymbol{\ell'}(x, y, z)] \tag{5.22}$$

Under the assumption that $r \approx r_\omega$, we can use the following approximation:

$$Loss(\theta) \approx \mathbb{E}_{\substack{x\sim p(x) \\ y\sim q_\theta(y|x) \\ z\sim r_\omega(z|x,y)}} [\boldsymbol{\ell}(x, y, z)] \tag{5.23}$$

Using the same notations as in the previous section, we define the two empirical losses over the dataset $D = \{x_m\}_{m \in M}$ as:

$$Loss(\theta; D) = \frac{1}{|M|} \sum_{m \in M} \mathbb{E}_{\substack{y \sim q_\theta(y|x_m) \\ z \sim r_\omega(z|x_m, y)}} [\boldsymbol{\ell}(x_m, y, z)] \tag{5.24}$$

$$Loss'(\omega; D) = \frac{1}{|M|} \sum_{m \in M} \mathbb{E}_{\substack{y \sim \overline{q}_\theta(y|x_m) \\ z \sim r_\omega(z|x_m, y)}} [\boldsymbol{\ell'}(x_m, y, z)] \tag{5.25}$$

In the voltage control problem, both cost functions $\boldsymbol{\ell}, \boldsymbol{\ell'}$ are differentiable. Moreover, we choose to use deterministic mappings for both q_θ and r_ω using the two H2MGNN mappings $\boldsymbol{f_\theta}$ and $\boldsymbol{f'_\omega}$, and to define $\overline{q}_\theta$ to be Gaussian noise $\epsilon \sim \mathcal{N}(\boldsymbol{f_\theta}, \sigma I)$ (where σ is an hyperparameter). Thus, the two losses shall be rephrased as follows:

$$Loss(\theta; D) = \frac{1}{|M|} \sum_{m \in M} \boldsymbol{\ell}(x_m, \boldsymbol{f_\theta}(x), \boldsymbol{f'_\omega}(x, \boldsymbol{f_\theta}(x))) \tag{5.26}$$

$$Loss'(\omega; D) = \frac{1}{|M|} \sum_{m \in M} \mathbb{E}_{\epsilon \sim \mathcal{N}(0, \sigma I)} [\boldsymbol{\ell'}(x_m, \boldsymbol{f_\theta}(x_m) + \epsilon, \boldsymbol{f'_\omega}(x_m, \boldsymbol{f_\theta}(x_m) + \epsilon))] \tag{5.27}$$

Gradients $\nabla_\theta Loss(\theta; D)$ and $\nabla_\omega Loss'(\omega; D)$ can be estimated by automatic differentiation. The H2MGNN mappings $\boldsymbol{f_\theta}$ and $\boldsymbol{f'_\omega}$ are trained through a standard gradient-descent method, as described in Algorithm 5. Indeed, advanced gradient descent techniques introduced in Chapter 2 can be also used. Preliminary experiments using this approach are presented in Chapter 9.

5.2.3 Discussion

This section discusses issues regarding the bilevel SSP that justify the choices made in the proposed algorithm.

Deterministic approach

In early experiments, $\overline{q}_\theta$ was chosen to be deterministic and defined by the mapping $\boldsymbol{f_\theta}$, just like q_θ. We experimentally observed that $\boldsymbol{f_\theta}$ irremediably ended up predicting a constant value, regardless of the input x[1]. As a consequence, r_ω only observed a single value for y during its training, which lead it to consider that z is not a function of y. Thus, the gradient back-propagation through r_ω is stopped, which prevents q_θ from learning any further.

[1]This phenomenon is well known in the GAN literature under the name of "mode collapse" [146].

Algorithm 5 Training a bilevel Statistical Solver

procedure Train($D = \{x_m\}_{m \in M}, N_{epochs}, \alpha, \alpha'$)
  Initialize θ and ω
  for $epoch = 1, \dots, N_{epochs}$ **do**
    ▷ Optimize the controller
    for $m \in M$ **do**
      $y_m \leftarrow \boldsymbol{f_\theta}(x_m)$
      $z_m \leftarrow \boldsymbol{f'_\omega}(x_m, y_m)$
    $\nabla_\theta Loss(\theta; D) \leftarrow \frac{1}{|M|} \sum_{m \in M} \nabla_\theta \boldsymbol{\ell}(x_m, y_m, z_m)$
    $\theta \leftarrow \theta - \alpha \nabla_\theta Loss(\theta; D)$
    ▷ Optimize the solver
    for $m \in M$ **do**
      $y_m \leftarrow \boldsymbol{f_\theta}(x_m)$
      $\epsilon_m \sim \mathcal{N}(0, \sigma I)$
      $y_m \leftarrow y_m + \epsilon_m$
      $z_m \leftarrow \boldsymbol{f'_\omega}(x_m, y_m)$
    $\nabla_\omega Loss'(\omega; D) \leftarrow \frac{1}{|M|} \sum_{m \in M} \nabla_\omega \boldsymbol{\ell'}(x_m, y_m, z_m)$
    $\omega \leftarrow \omega - \alpha' \nabla_\omega Loss'(\omega; D)$
  return f_θ

Considering a probabilistic $\overline{q}_\theta$ allows to avoid this pitfall by forcing r_ω to be accurate for various values of y, thus allowing to properly back-propagate gradient through it to learn q_θ.

Training q_θ and r_ω separately

The probability distribution r_ω only serves as an approximation of r when learning q_θ. Thus, it seems reasonable to first train r_ω, and then use it to train q_θ. However, we experimentally observed that the training distribution q_θ tends to exploits weaknesses of r_ω by prioritizing areas where it is a poor approximation of r. Training both models jointly (alternating iterations on one and the other) prevents this type of behavior by constantly updating r_ω so as to always be accurate in areas that are relevant to q_θ.

Learning a joint probability for y and z

A possible avenue for future work could be to consider the Lagrangian relaxation of the bilevel problem:

$$(\mathcal{Y} \times \mathcal{Z})^*(x) = \underset{y \in \mathcal{Y}, z \in \mathcal{Z}}{\arg\min}\ \boldsymbol{\ell}(x, y, z) + \lambda \boldsymbol{\ell'}(x, y, z) \tag{5.28}$$

where $\lambda \in \mathbb{R}^{+*}$. Although not equivalent to the initial bilevel problem, this single optimization problem could still provide a good approximation of the actual solution if λ is carefully adjusted (or updated during the search). We argue that in such a framework, the resulting distribution of y may be altered by the lower-level cost function ℓ', as the optimal solution is not necessarily the same as in the initial bilevel optimization problem. Our two-model framework enforces that the distribution of y is not biased in any way by the lower-level problem.

In this chapter, we have detailed how to transform both single level and bilevel optimization problems into Statistical Solver Problems. This allows to train a neural network mapping to minimize a cost function for a distribution of problem instances. The resulting deep neural network is called a Deep Statistical Solver. In the following, Chapter 6 proves that the H2MGNN introduced in Chapter 4 can approximate the actual solution of an SSP with arbitrary precision. Then, Chapters 7, 8 and 9 show the application of the DSS approach to several use cases.

Chapter 6

Proving that global continuous problems can be solved through local operations

In the present work, we consider optimization problems defined over graphs that are invariant by permutation of vertices and by permutation of collocated objects of the same class, *i.e.* for which the cost function does not depend on any specific ordering. To give a power grid example, the violation of physical laws should be minimized at every bus of a power grid, which does not depend on any specific object ordering. Moreover, we propose to approximate the mapping from problem instances to their respective solutions by using a GNN architecture that relies on local message passing.

The goal of the present chapter is to prove that the H2MGNN architecture is indeed able to approximate with an arbitrary precision the actual target mapping of problem instances to their respective solutions[1]. The main theoretical finding of the present PhD thesis is that under certain assumptions, optimization problems over graphs can be solved using a series of local operations. This holds even for problems that involve coupling of variables over long range.

In Section 6.1, we formalize this finding as Theorem 2, after having carefully introduced assumptions about the data distribution and the structure of the optimization problem. In Section 6.2, we dive into the details of the proof.

This universal approximation theorem is non-constructive, i.e., it does not offer any guarantee of convergence toward an ideal solver. But there hardly exist such convergence guarantees in the field of DL. However, this non-trivial result provides a solid theoretical ground to

[1]All density results are with regards to the uniform norm $\|.\|_\infty$

the proposed approach by proving its consistency.

This theoretical contribution is a joint work with Zhengying Liu (Université Paris-Saclay, INRIA).

6.1 Universal approximation theorem

We first recall some notations about the considered graph data, and introduce some assumptions about their associated probability distributions. Then we further motivate the assumption about the permutation-invariance of the cost function, and refine our definition of the Hypothesis space. Finally, the main theoretical contribution of this PhD thesis is stated.

6.1.1 Data and distribution

With the notations of Chapter 4, let $x \in \mathcal{X}$ be a H2MG of size n, and denote by $(n, \mathcal{C}, \mathcal{E}, \mathcal{M})$ its structure. Two vertices are considered to be linked if there exists an hyper-edge of any class that is connected to both. This allows us to define the notion of *geodesic distance* between two vertices i and i' as the shortest number of hops between linked vertices going from i to i'. The *diameter* of x is then defined as the largest geodesical distance between any two of its vertices and is denoted by $diam(x)$. In the case where x is composed of multiple disjoint components, there exist pairs of vertices that cannot be joined by leaps between linked vertices, implying that $diam(x) = +\infty$.

In the following, we are interested in graphs sampled according to a probability distribution $p(x)$. We introduce the following set of hypotheses (H), over the graphs in the support of p, $supp(p) := \{x \in \mathcal{X} | p(x) > 0\}$.

(H1) *Uniqueness of hyper-edges.* For any $x \in supp(p)$ of structure $(n, \mathcal{C}, \mathcal{E}, \mathcal{M})$, any class c and hyper-edge e, $|\mathcal{M}_e^c| = 1$;

(H2) *Permutation-invariance.* For any $x \in supp(p)$ of structure $(n, \mathcal{C}, \mathcal{E}, \mathcal{M})$, and $\sigma \in \Sigma_n$, $\sigma \star x \in supp(p)$;

(H3) *Compactness.* $supp(p)$ is a compact subset of $\mathcal{X}$;

(H4) *Connectivity.* For any $x \in supp(p)$, x has a single component (*i.e.* $diam(x) < +\infty$);

(H5) *Separability of input features.* There exists $\delta > 0$ such that for any $x \in supp(p)$, any class $c \in \mathcal{C}$ and any pair of hyper-edges $e \neq e'$, $\|x_e^c - x_{e'}^c\| \geq \delta$.

The uniqueness of hyper-edges hypothesis (H1) means that $supp(p)$ only contains H2MGs that have no collocated objects of the same class. We believe that this hypothesis could be alleviated, although we leave this task to future work. For now, we may drop indices m and simply denote the structure of H2MGs as $(n, \mathcal{C}, \mathcal{E})$.

The compactness hypothesis (H3) implies in particular that there is an upper bound n_{max} over the sizes of the H2MGs in $supp(p)$. Also, these hypotheses imply that there is a finite upper bound (denoted by D) on the diameters of all graphs in $supp(p)$.

The technical hypothesis (H5) is necessary to ensure that local message passing operations can distinguish between any two non-isomorphic graphs[2], which allows to bypass one major limitation of local message passing [147, 148]. Moreover, this hypothesis is justified by our real-life power grid application where it is highly unlikely that two objects have exactly the same features.

6.1.2 The cost function

The context is that of a single level optimization problem, as defined in Section 5.1. Let $\mathcal{Y}$ be a space of H2MGs and let ℓ be a real-valued cost function defined for all compatible pairs $(x, y) \in \mathcal{X} \times \mathcal{Y}$ (*i.e.* that share the same graph structure, see Section 4.1). Furthermore, we introduce the following set of hypotheses over the cost function:

(H6) For any $x \in supp(p)$, there exists a unique minimizer $y^*(x) = \underset{y \in \mathcal{Y}}{\arg\min}\ \ell(x, y)$, and the mapping $x \mapsto y^*(x)$ is continuous (as in Section 5.1).

(H7) ℓ is permutation-invariant.

The uniqueness hypothesis (H6) will prove necessary in order to prove the expected theoretical result. However, experiments on the AC-PF (see Section 8.1) for which it does not hold nevertheless provide excellent results.

The permutation-invariance hypothesis (H7) is motivated by the fact that in most power grid related problems, quantities that should be minimized do not rely on any specific vertex or object ordering. For instance, the cost function considered in the AC-PF is the sum over all vertices of local power mismatches, which is independent of any specific vertex ordering.

An immediate consequence of the hypotheses above is given by:

[2]Two graphs are isomorphic if they are permuted versions of one another.

Property 1. *Under hypotheses (H1), (H6) and (H7), the mapping $x \mapsto y^*(x)$ is permutation-equivariant.*

Proof. Let $x \in \mathcal{X}$ be a graph of size n, and $\sigma \in \Sigma_n$ be a permutation.

$$\begin{aligned}
\ell(\sigma \star x, \sigma \star y^*(x)) &= \ell(x, y^*(x)) && \text{by invariance of } \ell \\
&= \min_{y \in \mathcal{Y}} \ell(x, y) && \text{by definition of } y^*(x) \\
&= \min_{y \in \mathcal{Y}} \ell(\sigma \star x, \sigma \star y) && \text{by invariance of } \mathcal{Y} \\
&= \min_{y \in \mathcal{Y}} \ell(\sigma \star x, y) && \text{by invariance of } supp(p) \\
&= \ell(\sigma \star x, y^*(\sigma \star x)) && \text{by definition of } y^*(x)
\end{aligned}$$

Moreover, the assumed uniqueness of the solution ensures that $y^*(\sigma \star x) = \sigma \star y^*(x)$, which concludes the proof. □

6.1.3 The Main Theorem

The Statistical Solver Problem we are tackling here is defined by ($\mathcal{X}$, $\mathcal{Y}$, p, ℓ), where p is a probability distribution over $\mathcal{X}$, and ℓ a cost function defined on compatible pairs in $supp(p) \times \mathcal{Y}$. The goal is to find, for any $x \in supp(p)$ the minimizer $y^*(x) = \arg\min_{y \in \mathcal{Y}} \ell(x, y)$.

Let $\mathcal{C}_{eq.}(supp(p), \mathcal{Y})$ be the set of continuous and permutation-equivariant functions (see Definition 2) from $supp(p)$ to the output space $\mathcal{Y}$. We recall from Chapter 4 that the equivariance of a function implies that it is only defined on compatible pairs (x, y).

Let $\mathcal{H}^D$ be the set of all H2MGNN architectures defined by Algorithm 2 of Section 4.2 with step size $1/\Delta t \leq D+1$ and latent dimension $d \in \mathbb{N}$, and for which the update mappings are MLPs with appropriate input and output dimensions.

The Universal Approximation Theorem is the following:

Theorem 2. *Let ($\mathcal{X}$, $\mathcal{Y}$, p, ℓ) be an SSP for which hypotheses (H1 – H7) above hold. Then $\mathcal{H}^D$ is dense in $\mathcal{C}_{eq.}(supp(p), \mathcal{Y})$ with regards to the uniform norm $\|.\|_\infty$.*

Equivalently, this theorem states that for any $\epsilon > 0$, there exists a H2MGNN $f_\theta \in \mathcal{H}^D$ such that:

$$\forall x \in supp(p), \|f_\theta(x) - y^*(x)\| \leq \epsilon \tag{6.1}$$

There always exists a H2MGNN as defined in Algorithm 2 that can approximate with an arbitrary precision the actual solution of the optimization problem. In other words, even global problems that involve

long-range coupling of variables can be solved by using enough local operations.

Moreover, the theorem relates the required amount of local message-passing operations with the maximum diameter D of H2MGs in $supp(p)$. This can be interpreted as follows. Each hyper-edge is required to guess its own output, by only communicating with hyper-edges that share a vertex with it. Since its own output is very likely to depend on the input and output of every other hyper-edge (of all classes) in the graph, it should gather information about every other object in the graph in order to make an educated guess. Since it takes at most D steps to go from any vertex to any other, it makes sense that we need a quantity higher than D to gather information about every other object.

6.1.4 Sketch of the proof

This subsection gives a very brief idea of the proof of Theorem 2. The full proof is given in next Section 6.2, but can be skipped by readers mainly interested in the results of the algorithms on practical applications, from toy problems to real-world Power Networks, given in Part III of this document, Chapters 7 and 8.

The basic idea behind the proof is to follow the approach of Keriven & Peyré [149]. However, their approach only considers mappings that output vectors at vertices, while our mappings output vectors at hyper-edges. The transfer from our context into their framework is done in the following 3 steps:

Step 1 – Lemma 1 All mappings in $\mathcal{C}_{eq.}(supp(p), \mathcal{Y})$ are decomposed in two distinct parts: the first part outputs vectors at vertices, while the second part locally converts these vertex quantities into hyper-edges quantities. The first part falls into the framework of Keriven & Peyré [149], while the second part is made of local continuous operators and can be treated easily.

Step 2 – Lemma 2 Similarly, we consider the subset of H2MGNNs that can be decomposed into a first part that performs multiple message passing steps and returns vectors at vertices, and a second part that maps this vertex information onto hyper-edges.

Step 3 – Lemma 7, Theorem 3 & Lemma 3 We show that both parts of the decompositions of the H2MGNNs are respectively dense in the corresponding part of the decompositions of $\mathcal{C}_{eq.}(supp(p), \mathcal{Y})$.

Furthermore, all parts of the decompositions of the H2MGNNs being Lipschitz, we may combine densities and prove that $\mathcal{H}^D$ is dense in $\mathcal{C}_{eq.}(supp(p), \mathcal{Y})$.

6.2 Proof of the Universal Approximation theorem

Hypotheses Let us consider in this section an SSP ($\mathcal{X}$, $\mathcal{Y}$, p, ℓ) that satisfies hypotheses (H1 – H7).

Notations The notations are the ones defined in Section 6.1.

Let us first recall that the set of all MLPs is dense in the set of continuous functions [109, 110], meaning that they can approximate with an arbitrary precision any continuous function with proper input and output dimensions.

6.2.1 Step 1: Decomposition of equivariant functions

In order to be able to fit into Keriven & Peyré's framework, we decompose all mappings in $\mathcal{C}_{eq.}(supp(p), \mathcal{Y})$ in three distinct parts: the first part outputs vectors at vertices and the second part converts these vertex quantities into hyper-edges quantities.

Let us first define the functional spaces corresponding to these decompositions.

Spaces $\mathcal{G}^d$ We consider the set of continuous and permutation-equivariant functions that associates H2MGs of size n with a series of d-dimensional vectors defined at each of the n vertices. Since $supp(p)$ may contain H2MGs of varying sizes, we introduce $\mathcal{V}^d = \bigcup_{n\in\mathbb{N}}(\mathbb{R}^d)^n$, and we denote by $\mathcal{G}^d := \mathcal{C}_{eq.}(supp(p), \mathcal{V}^d)$ the aforementioned set of functions.

$$g : \quad supp(p) \to \mathcal{V}^d \tag{6.2}$$

$$(x_e^c)_{(c,e)\in\mathcal{G}_x} \mapsto (g(x)_i)_{i\in[n]} \tag{6.3}$$

Spaces $\mathcal{R}^d$ We denote by $\mathcal{R}^d := \mathcal{C}(\mathcal{V}^d \times supp(p), \mathcal{Y})$ the set of continuous functions that can be written as:

$$r : \quad supp(p) \times \mathcal{V}^d \to \mathcal{Y} \tag{6.4}$$

$$(x_e^c)_{(c,e)\in\mathcal{G}_x}, (\beta(x)_i)_{i\in[n]} \mapsto (r^c(x_e^c, \beta_e))_{(c,e)\in\mathcal{E}^c} \tag{6.5}$$

where r^c are continuous mappings of appropriate dimensions. It combines vectors located at hyper-edges and vectors located at vertices to output vectors at hyper-edges. The β_e on the right hand side correspond to the β_i of the left-hand side that are neighbors of hyper-edge e. For a given class c, the mapping r^c is applied simultaneously at every hyper-edge.

The composition of such mappings is then defined by:

$$r \circ g : \quad supp(p) \to \mathcal{Y} \tag{6.6}$$

$$(x_e^c)_{(c,e)\in\mathcal{G}_x} \mapsto (r^c(x_e^c, g(x)_e))_{(c,e)\in\mathcal{G}_x} \tag{6.7}$$

Such decomposition fulfills the goal of the needed decomposition: to have mappings that first output quantities solely located at vertices, and then locally convert this message and the local input into an actual prediction at a hyper-edge. We can now prove the following Lemma:

Lemma 1. $\exists d \in \mathbb{N}, \mathcal{C}_{eq.}(supp(p), \mathcal{Y}) \subseteq \mathcal{R}^d \circ \mathcal{G}^d$

Proof. Let $f \in \mathcal{C}_{eq.}(supp(p), \mathcal{Y})$, $x \in supp(p)$ with structure $(n, \mathcal{C}, \mathcal{E})$.

Setting the scene Let us consider a class $c \in \mathcal{C}$, and a specific port $o \in \mathcal{O}^c$. Let $i \in [n]$ be a vertex of x. Let us use the following notation: $\mathcal{N}_x(i; o, c) = \{e \in \mathcal{E}^c | e_o = i\}$. We introduce the following multivariate polynomial:

$$\forall \tilde{x} \in \mathbb{R}^{d^{c,x}}, \forall \tilde{y} \in \mathbb{R}^{d^{c,y}}, P_{x,c,o,i}(\tilde{x}, \tilde{y}) = \prod_{e\in\mathcal{N}_x(i;o,c)} (\|\tilde{x} - x_e^c\|_2^2 + \|\tilde{y} - f(x)_e^c\|_2^2) \tag{6.8}$$

The separability hypothesis (H5) implies that:

$$\forall o \in \mathcal{O}^c, f(x)_e^c = \underset{\tilde{y}\in\mathbb{R}^{d^{c,y}}}{\arg\min}\, P_{x,c,o,e_o}(x_e^c, \tilde{y}) \tag{6.9}$$

In other words, $f(x)_e^c$ can be retrieved from the knowledge of the polynomial P_{x,c,o,e_o}, and of the hyper-edge feature x_e^c. This holds for any $\forall o \in \mathcal{O}^c$, which means that the information is equivalently stored in all vertices to which the object is connected.

We choose to denote $\beta(x)_{c,o,i}$ the series of coefficients of the above polynomial (arbitrarily ordered), and propose the equivalent notations:

$$P_{x,c,o,i}(\tilde{x}, \tilde{y}) := P(\tilde{x}, \tilde{y}; \beta(x)_{c,o,i}) \tag{6.10}$$

Because $supp(p)$ is compact, we know that there is an upper bound over the amount of coefficients of all polynomials $P_{x,c,o,i}$. We can store for a single input x and a vertex index $i \in [n]$ all the coefficients of polynomials such that:

$$\beta(x)_i := ((\beta(x)_{c,o,i})_{o \in \mathcal{O}^c})_{c \in \mathcal{C}} \tag{6.11}$$

We observe that classes and ports can be arbitrarily ordered, without breaking any permutation-equivariance w.r.t. the ordering of the nodes themselves. Thus $\beta(x)_i$ can be represented as a vector in a unique manner. By compacity of $supp(p)$, there exists an upper bound over the dimensions required to represent $\beta(x)_i$, which we denote by d^β. For the sake of simplicity, we embed all vectors $\beta(x)_i$ for $x \in supp(p)$ in a d^β-dimensional space, padding with zeros when required.

Defining g We introduce $g : x \mapsto (\beta(x)_i)_{i \in [n]}$. It maps input graphs to vectors of coefficient of polynomials at each vertex. It is continuous with regards to x because each coefficient can be written as a product of continuous features of x and of $f(x)$, which is assumed to be a continuous function of x. To prove that it is permutation-equivariant, it is sufficient to prove that for any $x \in supp(p)$, any vertex $i \in [n]$, any class $c \in \mathcal{C}$, any port $o \in \mathcal{O}^c$, and any permutation $\sigma \in \Sigma_n$, we have $\beta(\sigma \star x)_{c,o,i} = \beta(x)_{c,o,\sigma^{-1}(i)}$. Since two polynomials are equal if and only if their coefficients are equal, it also amounts to proving that $P_{\sigma\star x,c,o,i} = P_{x,c,o,\sigma^{-1}(i)}$. Let $\tilde{x} \in \mathbb{R}^{d^{c,x}}$ and $\tilde{y} \in \mathbb{R}^{d^{c,y}}$,

$$\begin{aligned}
P(\tilde{x}, \tilde{y})_{\sigma\star x,c,a,i} &= \prod_{e \in \mathcal{N}_{\sigma\star x}(i;o,c)} (\|\tilde{x} - (\sigma \star x)^c_e\|_2^2 + \|\tilde{y} - f(\sigma \star x)^c_e\|_2^2) \\
&= \prod_{e \in \mathcal{N}_{\sigma\star x}(i;o,c)} (\|\tilde{x} - (\sigma \star x)^c_e\|_2^2 + \|\tilde{y} - (\sigma \star f(x))^c_e\|_2^2) \\
&= \prod_{e \in \mathcal{N}_{\sigma\star x}(i;o,c)} (\|\tilde{x} - x^c_{\sigma^{-1}(e)}\|_2^2 + \|\tilde{y} - f(x)^c_{\sigma^{-1}(e)}\|_2^2) \\
&= \prod_{\sigma^{-1}(e) \in \mathcal{N}_x(\sigma^{-1}(i);o,c)} (\|\tilde{x} - x^c_{\sigma^{-1}(e)}\|_2^2 + \|\tilde{y} - f(x)^c_{\sigma^{-1}(e)}\|_2^2) \\
&= \prod_{e \in \mathcal{N}_x(\sigma^{-1}(i);o,c)} (\|\tilde{x} - x^c_e\|_2^2 + \|\tilde{y} - f(x)^c_e\|_2^2) \\
&= P(\tilde{x}, \tilde{y})_{x,c,o,\sigma^{-1}(i)}
\end{aligned}$$

Thus $g \in \mathcal{G}^{d^\beta}$.

Defining r For each $c \in \mathcal{C}$, we introduce r^c as

$$\forall \tilde{x} \in \mathbb{R}^{d^{c,x}}, \forall \beta \in \mathbb{R}^d, r^c(\tilde{x}, \beta) = \underset{\tilde{y} \in \mathbb{R}^{d^{c,y}}}{\arg\min}\, P(\tilde{x}, \tilde{y}; \beta) \tag{6.12}$$

We denote by r the mapping that relies on mappings $(r^c)_{c \in \mathcal{C}}$ as shown in equation (6.5). We observe that mappings r^c are not continuous. However, the mapping $r \circ g$ is continuous over the compact set $supp(p)$ because of the separability assumption (H5): two terms of the product in equation 6.8 cannot be simultaneously zero. Thus, we may take continuous extensions r'^c of r^c over $\mathcal{X}$ such that $\forall x \in supp(p), r' \circ g(x) = r \circ g(x)$ (Tietze theorem). We have that $r' \in \mathcal{R}^{d^\beta}$, and, by construction (Eq. 6.9):

$$\forall x \in supp(p), r' \circ g(x) = f(x) \tag{6.13}$$

This reasoning is true for any $f \in \mathcal{C}_{eq.}(supp(p), \mathcal{Y})$, and hence $\mathcal{C}_{eq.}(supp(p), \mathcal{Y}) \subseteq \mathcal{R}^{d^\beta} \circ \mathcal{G}^{d^\beta}$, which concludes the proof of Lemma 1. □

In the remainder of this proof, we simply denote d^β as d, $\mathcal{V}^{d^\beta}$ as $\mathcal{V}$, and $\mathcal{R}^{d^\beta}, \mathcal{G}^{d^\beta}$ as $\mathcal{R}, \mathcal{G}$.

6.2.2 Step 2: Decomposition of H2MGNNs

Similarly, we consider the subset of H2MGNNs that can be decomposed into a first part that locates all information at vertices, and a second part that maps this vertex information only onto hyper-edges.

We consider two distinct classes of H2MGNN architectures. The first is detailed in Algorithm 6: we denote by $\mathcal{H}^D_{\mathcal{G}}$ the set of all such H2MGNNs that satisfy $1/\Delta t \leq D + 1$ updates, and that output d-dimensional vectors at vertices. The second class is detailed in Algorithm 7, and we denote by $\mathcal{H}_{\mathcal{R}}$ the set of all such neural networks. We propose to combine functions from these two sets similarly to equation (6.7), and denote by $\mathcal{H}_{\mathcal{R}} \circ \mathcal{H}^D_{\mathcal{G}}$ such mappings.

The following result is true by construction of $\mathcal{H}_{\mathcal{R}}$ and $\mathcal{H}^D_{\mathcal{G}}$, considering a zero initialization and appropriate neural network blocks:

Lemma 2. $\mathcal{H}_{\mathcal{R}} \circ \mathcal{H}^D_{\mathcal{G}} \subseteq \mathcal{H}^D$

6.2.3 Step 3: Composition of densities

We show that the two parts of the decompositions of the H2MGNNs are respectively dense in the two parts of the decompositions of $\mathcal{C}_{eq.}(supp(p), \mathcal{Y})$.

Algorithm 6 First part of the decomposition

procedure $g_{\boldsymbol{\theta}}(x)$
  $t \leftarrow 0$
  while $t < 1$ **do**
    for $i \in [n]$ **do**
      $h_i^v \leftarrow h_i^v + \sum_{(c,e,o)\in\mathcal{N}_x(i)} \boldsymbol{\phi}_{\boldsymbol{\theta}}^{\boldsymbol{c,o}}(t, h_e^v, h_e^c, x_e^c)$
    for $(c,e) \in \mathcal{G}_x$ **do**
      $h_e^c \leftarrow h_e^c + \boldsymbol{\phi}_{\boldsymbol{\theta}}^{\boldsymbol{c,h}}(t, h_e^v, h_e^c, x_e^c)$
    $t \leftarrow t + \Delta t$
  return $h^v = (h_i^v)_{i\in[n]}$

Algorithm 7 Second part of the decomposition

procedure $\boldsymbol{r}_{\boldsymbol{\theta}}(x, h^v)$
  for $(c,e) \in \mathcal{G}_x$ **do**
    $\hat{y}_e^c \leftarrow \boldsymbol{\phi}_{\boldsymbol{\theta}}^{\boldsymbol{c,y}}(h_e^v, x_e^c)$
  return $\hat{y} = (\hat{y}_e^c)_{(c,e)\in\mathcal{G}_x}$

Moreover, the second parts of the decompositions of the H2MGNNs being Lipschitz, we may combine densities and prove that $\mathcal{H}^D$ is dense in $\mathcal{C}_{eq.}(supp(p), \mathcal{Y})$.

With the notations introduced in Step 1 and Step 2 above, this amounts to prove that $\mathcal{H}_{\mathcal{G}}^D$ is dense in $\mathcal{G}$ (Theorem 3), and to prove that $\mathcal{H}_{\mathcal{R}}$ is dense in $\mathcal{R}$ and contains only Lipschitz functions (Lemma 3). The latter is straightforward:

Lemma 3. *$\mathcal{H}_{\mathcal{R}}$ is dense in $\mathcal{R}$, and contains only Lipschitz mappings.*

Proof. This was proven by Cybenko in 1989 [109], and further detailed by Hornik [110]. □

The last missing piece of the demonstration of the Universal Approximation Theorem is the following:

Theorem 3. *$\mathcal{H}_{\mathcal{G}}^D$ is dense in $\mathcal{G}$*

The proof of this theorem closely follows the approach of [149]. We first prove a modified version of the Stone-Weierstrass Theorem (Theorem 4), and then verify that all involved spaces indeed satisfy the conditions of this Theorem by proving Lemma 4.

Notations Let $\mathcal{X}_{eq.} \subseteq \mathcal{X}$ be a compact and permutation-invariant set of H2MGs such that there are no collocated objects of the same class (we may thus drop indices m). The compactness implies that there exists $\overline{n} \in \mathbb{N}$, upper bound of the size of all H2MGs in $\mathcal{X}_{eq.}$. Let $\mathcal{C}_{eq.}(\mathcal{X}_{eq.}, \mathcal{V})$ be the space of continuous and permutation-equivariant functions from $\mathcal{X}_{eq.}$ to $\mathcal{V}$ that associate to a H2MG $x = (x_e^c)_{(c,e)\in\mathcal{G}_x}$ a vector $(\beta_i)_{i\in[n]}$ at each of its vertices i. $(\mathcal{C}_{eq.}(\mathcal{X}_{eq.}, \mathcal{V}), +, \cdot, \odot)$ is a unital $\mathbb{R}$-algebra, where $(+, \cdot)$ are the usual addition and multiplication by a scalar, and $\odot$ is the Hadamard product defined by $(g \odot g')(x)_i = g(x)_i \cdot g'(x)_i$. Its unit is the constant function $\mathbf{1} = (1, \dots, 1)$.

We can now state the following Theorem:

Theorem 4. *(Modified Stone-Weierstrass Theorem for equivariant functions.) Let $\mathcal{A}$ be a unital subalgebra of $\mathcal{C}_{eq.}(\mathcal{X}_{eq.}, \mathcal{V})$, (i.e. contains the unit function $\mathbf{1}$) and assume both following properties hold:*

- *(Separability) For all $x, x' \in \mathcal{X}_{eq.}$, with number of vertices n and n' such that x is not isomorphic to x', and for all $i \in [n]$, $i' \in [n']$, there exists $f \in \mathcal{A}$ such that $f(x)_i \neq f(x')_{i'}$;*
- *(Self-separability) For all $n \leq \overline{n}, I \subseteq [n], x \in \mathcal{X}_{eq.}$ with n vertices, such that no isomorphism of x exchanges at least one index between I and I^C, and for all $i \in I$, $i' \in I^C$, there exists $f \in \mathcal{A}$ such that $f(x)_i \neq f(x)_{i'}$.*

Then $\mathcal{A}$ is dense in $\mathcal{C}_{eq.}(\mathcal{X}_{eq.}, \mathcal{V})$ with respect to the uniform metric.

This proof of Theorem 4 is almost identical to that of Theorem 4 in [149], with the following differences.

1. For the input space, we consider h2mg of the form $([n], (x_e^c)_{(c,e)\in\mathcal{G}_x})$ where $x_e^c \in \mathbb{R}^{d^{x,c}}$ for all $c \in \mathcal{C}$ and $e \in \mathcal{E}^c$, instead of hyper-graphs defined in $\mathbb{R}^{n^b}$ for some $b \in \mathbb{N}$. The corresponding metrics are different, although the difference is not critical for the proof;
2. Similarly, we consider as output space $(\mathbb{R}^d)^n$ instead of $\mathbb{R}^n$;
3. We only assume $\mathcal{X}_{eq.} \subseteq \mathcal{X}$ to be compact and permutation-invariant, whereas [149] explicitely builds a space $\mathcal{X}_{eq.} := \{x \in \mathbb{R}^{n^d} | n \leq n_{\max}, \|x\| \leq R\}$ (this makes Theorem 4 above more general).

Let us now discuss how to overcome these differences in order to mimic the proofs of [149].

1. The only properties of the input space involved in [149] are the number of vertices, action of permutation and the metric (with the corresponding topology). For the first two points, everything is still applicable in our setting. For the topology, the difference is not critical either since we are actually considering the product of several metric spaces defined in [149] and all corresponding properties follow.

2. The case $d = 1$ is as in [149], and the general case amounts to stacking the resulting function d times. This works seamlessly with Hadamard product and all properties related to density.

3. There is actually no dependency on the explicit form of $\mathcal{X}_{eq.}$ or $\mathcal{X}$ in [149] (as for the case in 1). And the proof only relies on the upper bound on the number of vertices. So the generalization can be naturally obtained.

The detailed proof of Theorem 4 then follows the exact same procedure as that of Theorem 4 in [149], and we shall omit it here, referring the reader to [149] for all details.

Applying Theorem 4 Our goal is to prove that Theorem 4 can be applied to $\mathcal{H}_{\mathcal{G}}^{D}$. Because $\mathcal{H}_{\mathcal{G}}^{D}$ is not obviously an algebra (see Lemma 9), let us consider $\mathcal{H}_{\mathcal{G}}^{D,\odot}$, the algebra generated by $\mathcal{H}_{\mathcal{G}}^{D}$ with respect to the Hadamard product. More formally:

$$\mathcal{H}_{\mathcal{G}}^{D,\odot} = \left\{ \sum_{s=1}^{S} \bigodot_{u=1}^{U_s} c_{su} g_{su} | S \in \mathbb{N}, U_s \in \mathbb{N}, c_{su} \in \mathbb{R}, g_{su} \in \mathcal{H}_{\mathcal{G}}^{D} \right\}. \tag{6.14}$$

Note that the Hadamard product among g_{su}'s is well-defined since for a fixed input x, all output values $g_{su}(x)$ take the same dimension – the size of x.
$(\mathcal{H}_{\mathcal{G}}^{D,\odot}, +, \cdot, \odot)$ is obviously a unital sub-algebra of $(\mathcal{X}_{eq.}, +, \cdot, \odot)$ (the constant function $(1, \ldots, 1)$ trivially belongs to $\mathcal{H}_{\mathcal{G}}^{D,\odot}$). In order to apply Theorem 4 to $\mathcal{H}_{\mathcal{G}}^{D,\odot}$, one needs to prove that it satisfies both separability hypotheses.

Proving the separability Let us first notice that the self-separability property is a straightforward consequence of the hypothesis of separability of external inputs on $supp(p)$. Hence we only need to prove the separability property:

Lemma 4. *$\mathcal{H}_{\mathcal{G}}^{D,\odot}$ satisfies the separability property of Theorem 4.*

The proof consists of 3 steps.

- (Lemma 5) We prove that for all $x, x' \in supp(p)$ that are not isomorphic, there exists a vertex i^* in x whose hyper-edge neighborhood never appears in x'.

- (Lemma 6) We build a continuous function $g^\dagger$ on $\mathcal{X}$ that returns an indicator of the presence of this sequence in the input graph.

- (Lemma 8) We prove that there exists a function $g_\theta \in \mathcal{H}_\mathcal{G}^D \subseteq \mathcal{H}_\mathcal{G}^{D,\odot}$ that approximates well enough $g^\dagger$.

Lemma 5. *Let x and x' two non isomorphic H2MGs in $supp(p)$, of respective sizes n and n'. There exists $i \in [n]$, such that for any $i' \in [n]$,*

$$\{(x_e^c, c, o) | (c, e, o) \in \mathcal{N}_x(i)\} \neq \{(x_e'^c, c, o) | (c, e, o) \in \mathcal{N}_{x'}(i')\} \tag{6.15}$$

Proof. (of Lemma 5) This lemma relies on the separability hypothesis (H5) which states that there exists $\delta > 0$ such that for all $x \in supp(p)$ and $c \in \mathcal{C}$, for any $e \neq e' \in \mathcal{E}^c$, $\|x_e^c - x_{e'}^c\| \leq \delta$.

We shall use proof by contradiction: assume that for any $i \in [n]$, there exists $\alpha(i) = i' \in [n']$ such that $\{(x_e^c, c, o) | (c, e, o) \in \mathcal{N}_x(i)\} = \{(x_e'^c, c, o) | (c, e, o) \in \mathcal{N}_{x'}(i')\}$. Two cases must be distinguished, depending on whether $n < n'$ or $n = n'$.

- If $n > n'$, then according to the pigeonhole principle, there exist two indices $i \in [n]$ and $j \in [n]$ that have the same image by α, $i' \in [n']$. Hence, we have $\{(x_e^c, c, o) | (c, e, o) \in \mathcal{N}_x(i)\} = \{(x_e^c, c, o) | (c, e, o) \in \mathcal{N}_x(j)\}$. This means that both vertices i and j are connected to an hyper-edge of the same class c and with the same feature x_e^c through the same port o. It implies that there are two distinct objects of the same class that have the exact same feature, which contradicts the separability hypothesis.

- If $n = n'$, according to the separability hypothesis (H5), there cannot exist $i \neq j \in [n]$ that are mapped to the same $i' \in [n']$. Thus α actually defines an injective mapping. Because $n = n'$, this mapping is also surjective and hence bijective, making it a permutation. Thus, using the separability hypothesis, for any $e \in \mathcal{E}^c$, $\alpha(e) \in E'^c$ and $x_e^c = x_{\alpha(e)}'^c$, which means that x and x' are isomorphic, contradicting the hypothesis, and thus completing the proof of Lemma 5.

□

For convenience, we shall use a continuous kernel function defined by

$$K_\epsilon(x) = \max(0, 1 - |x|/\epsilon) \tag{6.16}$$

for $\epsilon > 0$. Then we have $K_\epsilon(0) = 1$ and $K_\epsilon(x) = 0$ for $|x| > \epsilon$.

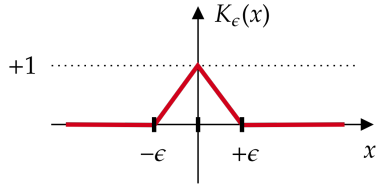


Figure 6.1: Kernel function

All intermediate functions of $\mathcal{H}^D_{\mathcal{G}}$, *i.e.* $((\boldsymbol{\phi}^{\boldsymbol{c,o}}_{\boldsymbol{\theta}})_{o\in\mathcal{O}^c})_{c\in\mathcal{C}}$ and $(\boldsymbol{\phi}^{\boldsymbol{c,h}}_{\boldsymbol{\theta}})_{c\in\mathcal{C}}$, live in function spaces that satisfy the Universal Approximation Property (UAP). So let us consider a space of continuous functions that share the same architecture, but in which all spaces of parameterized neural networks have been replaced by corresponding continuous function space. We denote this space by $\mathcal{H}^{D,\dagger}_{\mathcal{G}}$ (by convention, a dagger(†) added to a Neural Network block will refer to the corresponding continuous function space (*e.g.* $\boldsymbol{\phi}^{\boldsymbol{c,o},\dagger}_{\boldsymbol{\theta}}$). We are now in position to prove the following lemma.

Lemma 6. *For any $x, x' \in supp(p)$ that are not isomorphic, there exists a function $g^\dagger \in \mathcal{H}^{D,\dagger}_{\mathcal{G}}$ such that for any $i \in [n], i' \in [n']$, we have $g^\dagger(x)_i \neq g^\dagger(x')_{i'}$.*

Proof. (of Lemma 6) Without loss of generality, let us suppose $n \geq n'$. According to Lemma 5, there exists $i^* \in [n]$ such that for all $i' \in [n']$, $\{(x^c_e, c, o)|(c, e, o) \in \mathcal{N}_x(i)\} \neq \{(x'^c_e, c, o)|(c, e, o) \in \mathcal{N}_{x'}(i')\}$.

For any class c and port o, we denote by $\mathcal{X}^{*c,o} = \{x^c_e|e_o = i^*\}$. We can arbitrarily order classes, ports, and elements in $\mathcal{X}^{*c,o}$. Each tuple $c \in \mathcal{C}$, $o \in \mathcal{O}^c$ and $\tilde{x} \in \mathcal{X}^{*c,o}$ is associated with a unique integer $l(c, o, \tilde{x}) \in \mathbb{N}$. By denoting $L = \sum_{c\in\mathcal{C}} \sum_{o\in\mathcal{O}^c} |\mathcal{X}^{*c,o}|$, we can choose integers that do not overlap and lie in $\{1, \dots, L\}$.

Moreover, let us denote $\mathcal{X}^c := \{x^c_e|e \in \mathcal{E}^c\} \cup \{x'^c_e|e \in \mathcal{E}'^c\}$, and $\epsilon^c = \min_{\tilde{x}\neq\tilde{x'}\in\mathcal{X}^c} \|\tilde{x} - \tilde{x'}\|$. ϵ^c is thus the smallest non-zero distance between features of class c that appear in x and x'.

Besides, the compactness assumption implies that there exists a maximal number of hyper-edges connected to the same vertex. We denote by N this upper bound.

In the following, we construct a continuous function $g^\dagger : supp(p) \to \mathcal{V}$ that detects the presence of a vertex that has exactly the same neighborhood as i^* in x, and then propagate information to every other vertex of the graph. We have $g^\dagger(x) \geq (1, \dots, 1) \in \mathbb{R}^n$ (inequality is element-wise) and $g^\dagger(x') = (0, \dots, 0) \in \mathbb{R}^{n'}$ (thus proving Lemma 6).

Let us choose *continuous* functions $((\phi_{\boldsymbol{\theta}}^{\boldsymbol{c,o,\dagger}})_{o \in \mathcal{O}^c})_{c \in \mathcal{C}}$ (see line 5 of Algorithm 6) such that for any $x'' \in supp(p)$ with n'' vertices, and any vertex $i \in [n'']$, $h[x'']_i^v(\Delta t)$ is defined by:

$$h[x'']_i^v(\Delta t) = \sum_{(c,e,o) \in \mathcal{N}_{x''}(i)} \Big(\sum_{\tilde{x} \in \mathcal{X}^{*c,o}} K_{\epsilon^c}(\|x_e''^c - \tilde{x}\|) \times (N+1)^{l(c,o,\tilde{x})} + \sum_{\tilde{x} \in \mathcal{X}^c \setminus \mathcal{X}^{*c,o}} K_{\epsilon^c}(\|x_e''^c - \tilde{x}\|) \times (N+1)^{L+1} \Big) \tag{6.17}$$

One can observe that for both x and x', this quantity lies in $\mathbb{N}$, and is a sum of positive terms.

The second term of the expression enforces that if there is a single hyper-edge connected to i that is not in the hyper-edge neighborhood of i^* in x, then $h[x'']_i^v(\Delta t) \geq (N+1)^{L+1}$. The first term is basically a bijective base-$(N+1)$ numeration. Thus, in order to obtain exactly the value $\sum_{l \in [L]} (N+1)^l$ at vertex i, it is required that values of hyper-edges that are connected to i^* in x are present exactly one time. Therefore at every vertex i' of x', we have $h[x']_{i'}^v(\Delta t) \neq \sum_{l \in [L]} (N+1)^l$, and at vertex i^* of x, we have $h[x]_{i^*}^v(\Delta t) = \sum_{l \in [L]} (N+1)^l$.

We choose the following hyper-edge update:

$$h[x'']_e^c(\Delta t) = \sum_{o \in \mathcal{O}^c} K_1(|h[x'']_{e_o}^v(\Delta t) - \sum_{l \in [L]} (N+1)^l|) \tag{6.18}$$

which basically returns a quantity larger or equal to 1 at hyper-edges in x that are connected to i^*, and 0 to all hyper-edges in x'.

Then we use the two following series of updates:

$$h[x'']_i^v(t + \Delta t) = \sum_{(c,e,o) \in \mathcal{N}_{x''}(i)} h[x'']_e^c(t) \tag{6.19}$$

$$h[x'']_e^c(t + \Delta t) = \sum_{o \in \mathcal{O}^c} h[x'']_{e_o}^v(t + \Delta t) \tag{6.20}$$

which additively propagates information to all neighbors.

At $t = 2\Delta t$, at each vertex i that is a direct neighbor of i^* we have $h[x]_i^v(2\Delta t) \geq 1$. Then, at $t = 1$ it has propagated to every other vertex, meaning that $\forall i \in [n], h[x]_i^v(1) \geq 1$. Meanwhile, we have $\forall i' \in [n'], h[x']_{i'}^v(1) = 0$, which concludes the proof of Lemma 6 $\square$

In order to prove Lemma 8, we will need the following technical lemma:

Lemma 7. *Let X, Y, Z be three metric spaces. Let $\mathcal{F} \subseteq \mathcal{C}(X,Y)$ and $\mathcal{G} \subseteq \mathcal{C}(Y,Z)$ be two sets of continuous functions. And let $\mathcal{F}^\ell \subseteq \mathcal{F}, \mathcal{G}^\ell \subseteq \mathcal{G}$ be two subsets of Lipschitz functions that are dense in $\mathcal{F}$ and $\mathcal{G}$ respectively. Then $\mathcal{G}^\ell \circ \mathcal{F}^\ell := \{g \circ f | g \in \mathcal{G}^\ell, f \in \mathcal{F}^\ell\}$ is dense in $\mathcal{G} \circ \mathcal{F}$.*

Proof. (of Lemma 7) Let $g \circ f$ be a continuous function in $\mathcal{G} \circ \mathcal{F}$, $\epsilon > 0$. Due to the density of $\mathcal{G}^\ell$ in $\mathcal{G}$, there exists $g^\ell \in \mathcal{G}^\ell$ such that

$$\overline{d}(g, g^\ell) < \frac{\epsilon}{2}. \tag{6.21}$$

Let L_{g^ℓ} be the Lipschitz constant of g^ℓ, the density of $\mathcal{F}^\ell$ in $\mathcal{F}$ implies that there exists f^ℓ such that

$$\overline{d}(f, f^\ell) < \frac{\epsilon}{2L_{g^\ell}}. \tag{6.22}$$

Then we have

$$\begin{aligned} d_Z(g \circ f(x), g^\ell \circ f^\ell(x)) \leq & d_Z(g \circ f(x), g^\ell \circ f(x)) \\ & + d_Z(g^\ell \circ f(x), g^\ell \circ f^\ell(x)) && (6.23) \\ < & \frac{\epsilon}{2} + L_{g^\ell} d_Y(f(x), f^\ell(x)) && (6.24) \\ < & \frac{\epsilon}{2} + L_{g^\ell} \frac{\epsilon}{2L_{g^\ell}} = \epsilon && (6.25) \end{aligned}$$

for any $x \in X$. Thus $\overline{d}(g \circ f, g^\ell \circ f^\ell) < \epsilon$. Hence $\mathcal{G}^\ell \circ \mathcal{F}^\ell$ is dense in $\mathcal{G} \circ \mathcal{F}$. □

We may now proceed with Lemma 8.

Lemma 8. *$\mathcal{H}_{\mathcal{G}}^D$ is dense in $\mathcal{H}_{\mathcal{G}}^{D,\dagger}$.*

Proof. As functions in $\mathcal{H}_{\mathcal{G}}^D$ are composition of Lipschitz functions (neural network with linear transformation and Lipschitz activation as assumed), and all intermediate function spaces verify the Universal Approximation Property, we conclude immediately from using the definition of $\mathcal{H}_{\mathcal{G}}^{D,\dagger}$ and applying Lemma 7 consecutively. □

We are ready to prove Lemma 4, i.e., that $\mathcal{H}_{\mathcal{G}}^{D,\odot}$ satisfies the separability hypothesis of Theorem 4.

Proof. (of Lemma 4) It suffices to show the separability for $\mathcal{H}_{\mathcal{G}}^{D}$ since it is a subset of $\mathcal{H}_{\mathcal{G}}^{D,\odot}$.

Let $x, x' \in supp(p)$. According to Lemma 6, there exists $g^{\dagger} \in \mathcal{H}_{\mathcal{G}}^{D,\dagger}$ such that for any $i \in [n], i' \in [n']$, we have $g^{\dagger}(i)_i \neq g^{\dagger}(x')_{i'}$. According to Lemma 8, there exists $g \in \mathcal{H}_{\mathcal{G}}^{D}$ such that

$$\overline{d}(g^{\dagger}, g) < \frac{1}{3}. \tag{6.26}$$

Then for any $i \in [n], i' \in [n']$, we have $g(x)_i > \frac{2}{3}$ and $g(x')_{i'} < \frac{1}{3}$. This proves the separability of $\mathcal{H}_{\mathcal{G}}^{D}$. By observing that $\mathcal{H}_{\mathcal{G}}^{D} \subseteq \mathcal{H}_{\mathcal{G}}^{D,\odot}$, we obtain that $\mathcal{H}_{\mathcal{G}}^{D,\odot}$ respects the separability hypothesis of Theorem 4. □

Before being able to prove Theorem 2, we need the last following lemma.

Lemma 9. $\mathcal{H}_{\mathcal{G}}^{D} = \mathcal{H}_{\mathcal{G}}^{D,\odot}$.

Proof. (of Lemma 9) We shall prove this result by explicitly constructing an approximation function in $\mathcal{H}_{\mathcal{G}}^{D}$ for a given function in $\mathcal{H}_{\mathcal{G}}^{D,\odot}$.

Let $g^{\odot} \in \mathcal{H}_{\mathcal{G}}^{D,\odot}$. By definition of $\mathcal{H}_{\mathcal{G}}^{D,\odot}$ in eq. (6.14), there exists $S \in \mathbb{N}$, $\{U_s\}_{s\in\{1,\dots,S\}} \in \mathbb{N}^S$, as well as $\{c_{su}\} \in \mathbb{R}$ and $\{g_{su}\} \in \mathcal{H}_{\mathcal{G}}^{D}$ for all (s,u) with $s \in [S], u \in [U_s]$, such that :

$$g^{\odot} = \sum_{s=1}^{S} \bigodot_{u=1}^{U_s} c_{su} g_{su} \tag{6.27}$$

Thus, for any (s,u), there exists $1/\Delta t_{su} \leq D+1$, and $d_{su} \in \mathbb{N}$, such that g_{su} is composed of functions $((\boldsymbol{\Phi}_{\boldsymbol{\theta}}^{\boldsymbol{c,o,s,u}})_{o\in\mathcal{O}^c})_{c\in\mathcal{C}}$ and $(\boldsymbol{\Phi}_{\boldsymbol{\theta}}^{\boldsymbol{c,h,s,u}})_{c\in\mathcal{C}}$ applied $1/\Delta t_{su}$ times. d_{su} is the dimension of the latent state of *channel* g_{su}.

The different channels can have different number of propagation updates T_{su}, but they are all bounded by $D+1$. Without loss of generality, we can assume that all $1/\Delta t_{su}$ are equal to $D+1$ by padding, when needed, exactly $D+1-1/\Delta t_{su}$ null operations before the actual ones, scaling the input t appropriately, and scaling the updates by a factor $(D+1)/(\Delta t_{su})$.

Let $d = \sum_{s=1}^{S}\sum_{u=1}^{U_s} d_{su}$ be the cumulated dimensions of the different channels.

For each (s,u), we introduce the matrix $W_{su} \in \{0,1\}^{d_{su}\times d}$ which is defined by:

$$[W_{su}]_{ab} = \begin{cases} 1, & \text{if } \sum_{s'=1}^{s}\sum_{u'=1}^{U_{s'}} d_{s'u'} + \sum_{u'=1}^{u-1} d_{su'} + a = b \\ 0, & \text{otherwise.} \end{cases} \tag{6.28}$$

Thus $W_{su} = [0, \dots, 0, I_{d_{su}}, 0, \dots, 0]$. Basically, when given a vector of dimension d, W_{su} will be able to select exactly components that correspond to the channel (s, u), and will thus return a vector of dimension d_{su}.

Let us now define the functions $((\mathbf{\Phi}_{\boldsymbol{\theta}}^{\boldsymbol{c,o}})_{o \in \mathcal{O}^c})_{c \in \mathcal{C}}$ and $(\mathbf{\Phi}_{\boldsymbol{\theta}}^{\boldsymbol{c,h}})_{c \in \mathcal{C}}$, such that for any class $c \in \mathcal{C}$ and any port $o \in \mathcal{O}^c$ we have:

$$\mathbf{\Phi}_{\boldsymbol{\theta}}^{\boldsymbol{c,o}}(t, h_l^v, h_l^c, x_l^c) = \sum_{s=1}^{S} \sum_{u=1}^{U_s} W_{su}^\top . \mathbf{\Phi}_{\boldsymbol{\theta}}^{\boldsymbol{c,o,s,u}}(t, W_{su}.h_l^v, W_{su}.h_l^c, x_l^c) \quad (6.29)$$

$$\mathbf{\Phi}_{\boldsymbol{\theta}}^{\boldsymbol{c,h}}(t, h_l^v, h_l^c, x_l^c) = \sum_{s=1}^{S} \sum_{u=1}^{U_s} W_{su}^\top . \mathbf{\Phi}_{\boldsymbol{\theta}}^{\boldsymbol{c,h,s,u}}(t, W_{su}.h_l^v, W_{su}.h_l^c, x_l^c) \quad (6.30)$$

These functions used in Algorithm 6 define a mapping acting on a latent space of dimension d. Moreover, for any channel (s, u) and any vertex $i \in [n]$, we have $W_{su}.h_i^{T_{max}} = h_i^{T_{max},s,u}$.

We have thus built a function of $\mathcal{H}_{\mathcal{G}}$ that exactly replicates the steps performed on the different channels. By choosing as a last step function $\mathbf{\Phi}_{\boldsymbol{\theta}}^{\boldsymbol{c,a}}(1, \dots) = \sum_{s=1}^{S} \sum_{u=1}^{U_s} c_{su} \mathbf{\Phi}_{\boldsymbol{\theta}}^{\boldsymbol{c,a,s,u}}(1, \dots)$ we obtain a mapping $g \in \mathcal{H}_{\mathcal{G}}^D$ that can perfectly imitate $g^\odot \in \mathcal{H}_{\mathcal{G}}^{D,\odot}$. This concludes the proof of Lemma 9. □

We can now prove Theorem 3.

Proof. (of Theorem 3) According to the hypotheses of compactness (H3) and permutation-invariance (H2) on $supp(p)$, both conditions of Theorem 4 are satisfied by $supp(p)$. Consider the subalgebra $\mathcal{H}_{\mathcal{G}}^{D,\odot}$ defined by equation (6.14). According to the hypotheses of uniqueness of hyper-edges (H1), connectivity (H4) separability of input features (H5), and Lemma 4, $\mathcal{H}_{\mathcal{G}}^{D,\odot}$ satisfies the separability and self-separability conditions of Theorem 4. Applying Theorem 4, it comes that $\mathcal{H}_{\mathcal{G}}^{D,\odot}$ is dense in $\mathcal{G}$. Then according to Lemma 9, $\mathcal{H}_{\mathcal{G}}^{D} = \mathcal{H}_{\mathcal{G}}^{D,\odot}$. We conclude that $\mathcal{H}_{\mathcal{G}}^{D}$ is dense in $\mathcal{G}$ by the transitivity property of density. □

End of proof of Theorem 2 We now have all necessary ingredients to complete the proof of Theorem 2.

From Lemma 7, Theorem 3 and Lemma 3, it comes that $\mathcal{H}_{\mathcal{R}} \circ \mathcal{H}_{\mathcal{G}}^{D}$ is dense in $\mathcal{R} \circ \mathcal{G}$. But from Lemma 1 we know that $\mathcal{C}_{eq.}(supp(p), \mathcal{Y}) \subseteq \mathcal{R} \circ \mathcal{G}$. Thus $\mathcal{H}_{\mathcal{R}} \circ \mathcal{H}_{\mathcal{G}}^{D}$ is dense in $\mathcal{C}_{eq.}(supp(p), \mathcal{Y})$

And from Lemma 2 we know that $\mathcal{H}_{\mathcal{R}} \circ \mathcal{H}_{\mathcal{G}}^{D} \subseteq \mathcal{H}^D$. Hence $\mathcal{H}^D$ is dense in $\mathcal{C}_{eq.}(supp(p), \mathcal{Y})$. ■

Part III

Applications

Chapter 7

Toy Examples

Before applying the DSS methodology to power grids, we propose to illustrate it on a set of simpler optimization problems. In this chapter, we learn to solve linear systems[1] stemming from two distinct domains, namely a system of springs and the discretization of Poisson's equation. The goals of this chapter are as follows:

- Experimentally demonstrate the viability of our approach on a set of problems for which we have an efficient baseline (LU decomposition [150]);
- Scale up the approach to large graphs (up to $1,089$ nodes);
- Demonstrate the out-of-distribution generalization ability of trained DSSs to both larger and smaller graphs than those seen during training;
- Show that trained models generalize poorly to changes of orders of magnitude of the features;
- Visualize and interpret the behavior of latent variables in trained models.

The first section considers the problem of finding the equilibrium state of a system of springs, while the second focuses on systems stemming from the discretization of Poisson's equation (used in fluid dynamics, electrostatics, Newtonian gravity, etc.). Experiments conducted over the Poisson equation dataset are a joint work with Wenzhuo Liu (IRT SystemX).

[1] Knowing matrix A and vector b, the problem is to find u such that $Au = b$. However, u does not vary linearly with A, making the solution mapping $(A, b) \mapsto u$ nonlinear.

7.1 System of springs

The first application we propose to explore consists in finding the equilibrium position of a system of springs. Let us consider a series of nodes laid out as a horizontal regular 2D grid, linked by a series of springs with various stiffness parameters. Multiple weights are tied to randomly selected nodes. To counteract the downward forces applied by weights, several supports located at randomly selected nodes hold their node at a constant height.

This reasonably simple type of problem is a good test for our method, because the exact solution of the problem can be obtained through a matrix inversion, computed e.g., by the LU decomposition. Moreover, we can easily explore the out-of-distribution generalization ability of the trained models by considering various distributions distributions of systems of springs examples.

7.1.1 Problem & data generation

Let us now introduce the notations and the data generative process. We denote by x a system of springs. For the sake of simplicity, we only consider regular grids of $\sqrt{n} \times \sqrt{n}$ nodes, and physical coefficients are modulated by the dimensionless variable τ. Figure 7.1 displays the graph structures of samples drawn for 5 different values of $\sqrt{n}$. We may thus denote by $p(x; \sqrt{n}, \tau)$ the input data distribution of spring systems x that have $\sqrt{n} \times \sqrt{n}$ nodes and a feature coefficient τ. In our problem, there are exactly 4 object classes: $\mathcal{C} = \{springs, weights, supports, nodes\}$. As will be detailed thereafter, only springs, weights and supports have input features, and only nodes have output feature. Thus, $x = (x^{springs}, x^{weights}, x^{supports})$ and $y = (y^{nodes})$.

Springs In order to also have some diversity in terms of graph structure, we choose to cut open $\sqrt{n} - 2$ springs[2], making sure not to break the network into multiple components, as illustrated by Figure 7.2. Springs are defined by their respective stiffness constants k, which follow a uniform law $\mathcal{U}([\tau \times 1N.m^{-1}, \tau \times 10N.m^{-1}])$ (parameterized by τ), where N and m are SI base units. Thus, denoting $\mathcal{E}^{springs}$ the set of springs, their corresponding input is given by $x^{springs} = (k_e)_{e \in \mathcal{E}^{springs}}$. They apply symmetric forces to the nodes at their extremities, proportional to the stiffness coefficient and the height difference (assum-

[2] $\sqrt{n}$ is always chosen to be an integer

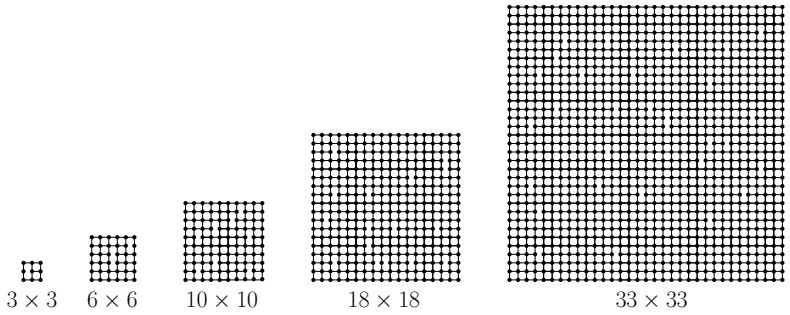


Figure 7.1: Top view of graph structure of input data x sampled using 5 distinct values for $\sqrt{n}$. From left to right, the parameter $\sqrt{n}$ is successively set to 3, 6, 10, 18 and 33. Exactly $\sqrt{n} - 2$ springs are cut open in each sample, so as to provide some topological variability.

ing that height displacement are much larger than horizontal displacement).

Weights Weights are located on 20% of all n nodes, uniformly sampled. They are defined by the constant downward force F that they apply to their nodes, which is sampled uniformly according to $\mathcal{U}([\tau \times 1N, \tau \times 10N])$. Thus, denoting $\mathcal{E}^{weights}$ the set of weights, their corresponding input is given by $x^{weights} = (F_e)_{e \in \mathcal{E}^{weights}}$.

Supports Similarly, supports are located at 20% of all n nodes, uniformly picked such that there is no overlap with weights. They are defined by the target height $\mathring{u}$ that they impose to their nodes. These altitudes are sampled uniformly according to $\mathcal{U}([-\tau \times 1m, \tau \times 1m])$. Thus, denoting $\mathcal{E}^{supports}$ the set of supports, their corresponding input is given by $x^{supports} = (\mathring{u}_e)_{e \in \mathcal{E}^{supports}}$.

Nodes While nodes do not bear any input feature, they do have an output $\hat{u}$, as we aim at finding for each node the altitude when the whole system is at the equilibrium. Thus, denoting $\mathcal{E}^{nodes}$ the set of nodes, their corresponding input is given by $y^{nodes} = (\hat{u}_e)_{e \in \mathcal{E}^{nodes}}$.

Cost function The optimization problem we consider consists in finding the equilibrium height of every node. However, two distinct behaviors can be observed, depending on whether a node is connected to a support or not. If there is a support, then the node is exactly at the same height as the support. We define $\mathbb{1}_e^{support}$, a binary indicator that equates to 1 if there is a support at node e, and 0 otherwise. The actual

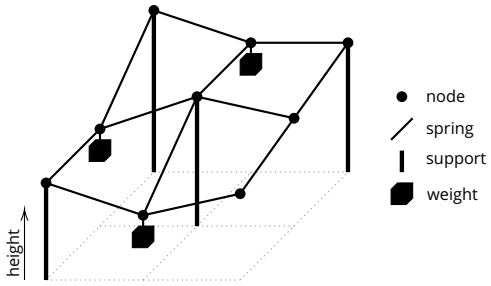


Figure 7.2: Instance of a spring system with $\sqrt{n} = 3$.

height of any node $e \in \mathcal{E}^{nodes}$ is given by:

$$u_e = \mathbb{1}_e^{support} \mathring{u}_e + (1 - \mathbb{1}_e^{support}) \hat{u}_e \tag{7.1}$$

It receives forces applied by springs and weights:

$$\Delta F_e = \sum_{e' \in \mathcal{E}^{springs}} (\mathbb{1}_{e'_1 = e} - \mathbb{1}_{e'_2 = e}) k_{e'} (u_{e'_1} - u_{e'_2}) + F_e \tag{7.2}$$

The cost function takes into account the force imbalance at nodes that have no support:

$$\ell(x, y) = \sum_{e \in \mathcal{E}^{nodes}} (1 - \mathbb{1}_e^{support}) \times |\Delta F_e|^2 \tag{7.3}$$

7.1.2 Experiments

Now that both the cost function ℓ and the data distributions $p(x; \sqrt{n}, \tau)$ are defined, let us detail the experimental setup.

Datasets We have chosen 5 distinct values for $\sqrt{n}$: $[3, 6, 10, 18, 33]$, and 5 distinct values for τ: $[0.1, 0.33, 1, 3.3, 10]$. For each of the 25 pairs of values $(\sqrt{n}, \tau)$, we have generated $100,000$ samples for the train sets, $10,000$ samples for the validation sets and $10,000$ samples for the test sets. We trained models for each of the following pairs: $(\sqrt{n} = 10, \tau = 0.1)$, $(\sqrt{n} = 10, \tau = 10)$, $(\sqrt{n} = 3, \tau = 1)$, $(\sqrt{n} = 33, \tau = 1)$. In a later subsection, the ability of trained models to generalize to all 25 datasets will be explored.

Baseline Since minimizing the imbalance at each node amounts to inverting a linear system given by equations 7.1 and 7.2, we compare the solution learned with DSS with that obtained with the LU method, by measuring the Pearson correlation (Corr), the normalised mean

absolute error (nMAE) and the normalised root mean squared error (nRMSE). The normalization is done by dividing the MAE and RMSE by the difference between the largest and smallest values (dividing by the mean for values centered around zeros does not make sense). The 10^{th}, 50^{th} and 90^{th} percentiles of the cost function ℓ are also displayed for both the LU and DSS models.

Model hyperparameters We used our own implementation of the H2MGNN architecture using TensorFlow [151] (no GNN framework were used). Hyperparameters have been tuned by trial and error using as validation set the $(\sqrt{n} = 10, \tau = 1)$ dataset. All four trained models then use the exact same hyperparameters. The step size and latent dimension are set to $\Delta t = 1/100$ and $d = 40$. All the neural networks $((\boldsymbol{\phi}_{\boldsymbol{\theta}}^{\boldsymbol{c,o}})_{o\in\mathcal{O}^c}, \boldsymbol{\phi}_{\boldsymbol{\theta}}^{\boldsymbol{c,h}}, \boldsymbol{\phi}_{\boldsymbol{\theta}}^{\boldsymbol{c,y}})_{c\in\mathcal{C}}$ that appear in the H2MGNN (Algorithm 2 in Section 4.2) are simple fully connected neural networks with 2 hidden layers, 80 neurons per hidden layers, and a hyperbolic tangent ($\tanh$) activation function.

Training All models have been trained using the Adamax [117] optimizer with parameters $(lr = 3 \times 10^{-3}, \beta_1 = 0.999, \beta_2 = 0.9999, \epsilon = 1 \times 10^{-12})$. Training was performed for 40 hours on a single NVIDIA TITAN Xp (163 epochs with batch size 50 for $\sqrt{n} = 3$; 118 epochs with batch size 50 for $\sqrt{n} = 10$; 10 epochs with batch size 2 for $\sqrt{n} = 33$). Significantly smaller batches were used for the dataset made of 33×33 nodes to avoid GPU saturation. However, results show that this has no obvious impact over the training quality.

Results Table 7.1 displays metrics of the quality of predictions of models over test sets stemming from the same distribution $p(x; \sqrt{n}, \tau)$ as they were trained on. In all four cases, our trained models are able to achieve an extremely good correlation with the LU method (even though the training is fully unsupervised), as evidenced by fact that $1 - Corr$ is systematically below 2e-4.

7.1.3 Out-of-distribution generalization

We now investigate the super-generalization abilities of each of the 4 trained models, recalling that they were all trained for a different pair $(\sqrt{n}, \tau)$. In Figure 7.3 we display the correlation for each model when used on each of the 25 test sets, one for each pair $(\sqrt{n}, \tau)$.

For a constant value of τ, we observe that models achieve good results when tested on both larger or smaller graphs. It achieves zero-

$(\sqrt{n}, \tau)$	$(10, 0.1)$		$(10, 10)$		$(3, 1)$		$(33, 1)$	
Method	DSS	LU	DSS	LU	DSS	LU	DSS	LU
1-Corr.	8e-6	-	4e-6	-	2e-4	-	3e-6	-
nRMSE	2e-4	-	5e-4	-	2e-3	-	2e-4	-
nMAE	5e-5	-	3e-4	-	3e-4	-	1e-4	-
ℓ 10^{th} p.	4e-13	5e-30	5e-1	3e-19	6e-9	4e-28	1e-8	2e-25
ℓ 50^{th} p.	9e-13	2e-29	2e0	9e-19	7e-8	2e-26	2e-8	4e-25
ℓ 90^{th} p.	3e-12	1e-28	5e0	3e-18	1e-6	1e-24	3e-8	1e-24

Table 7.1: For each of the 4 considered datasets, the DSS method is able to perform extremely accurate predictions, highly correlated with the LU method, while having been trained in a completely unsupervised manner.

shot learning [152] for problems of different topologies. This aspect of the out-of-distribution generalization is of primary importance, because in our main power grid application the topology tends to vary drastically. Moreover, we observe that a model trained on large graphs transfers better to small graphs than a model trained on small graphs transfers to large ones.

For a constant value of $\sqrt{n}$, we observe that changing the feature distribution drastically decreases the predictive power of trained models. This is not surprising as it completely changes the orders of magnitude of the input data, and feeds neural network with values that were never encountered during training. However, we argue that features orders of magnitude in power grids do not vary that much from one year to the other, so a model trained on one year should still be valid for the next one.

7.1.4 Visualization of latent variables

The H2MGNN architecture relies on the evolution of a series of latent variables, either defined at vertices, or at hyper-edges. Figure 7.4 displays the internal evolution of the first component of each variable w.r.t. parameter $t \in [0, 1]$. The non-trivial behavior and asynchronous oscillations that appear for some variables indicate that the model has learned how to have latent variables interact with each other. Figure 7.5 displays a 2D projection of latent variables trajectories by considering the first two dimensions. Similarly, we observe complex trajectories that result from interactions between latent variables.

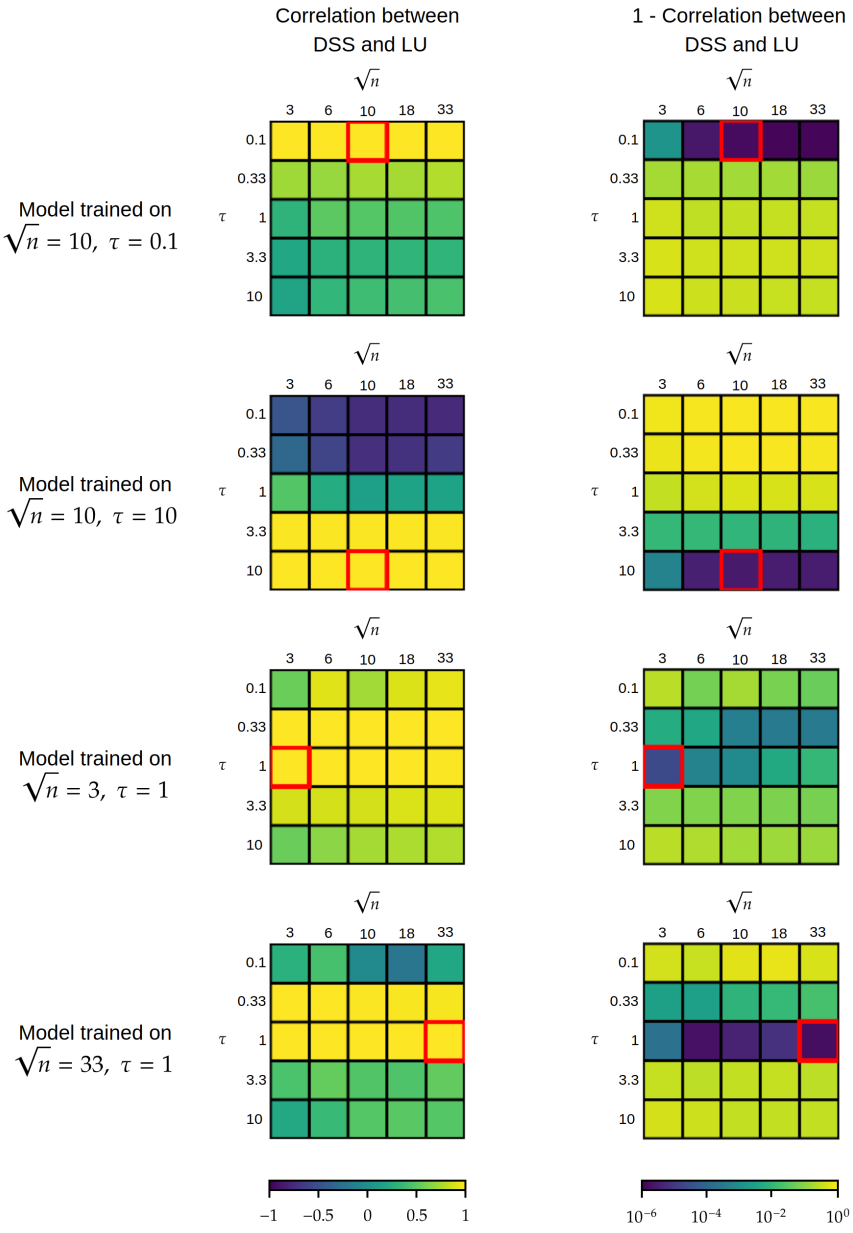


Figure 7.3: Correlations of the 4 trained models with LU solutions for the test sets of all 25 pairs $(\sqrt{n}, \tau)$. The second column displays $1 - Corr$ for more precise estimation of the prediction quality. The test sets stemming from the training distributions are highlighted in red. Trained models tend to generalize quite well to both larger and smaller graph sizes (controlled by $\sqrt{n}$), but have trouble generalizing to other orders of magnitude of features (controlled by τ).

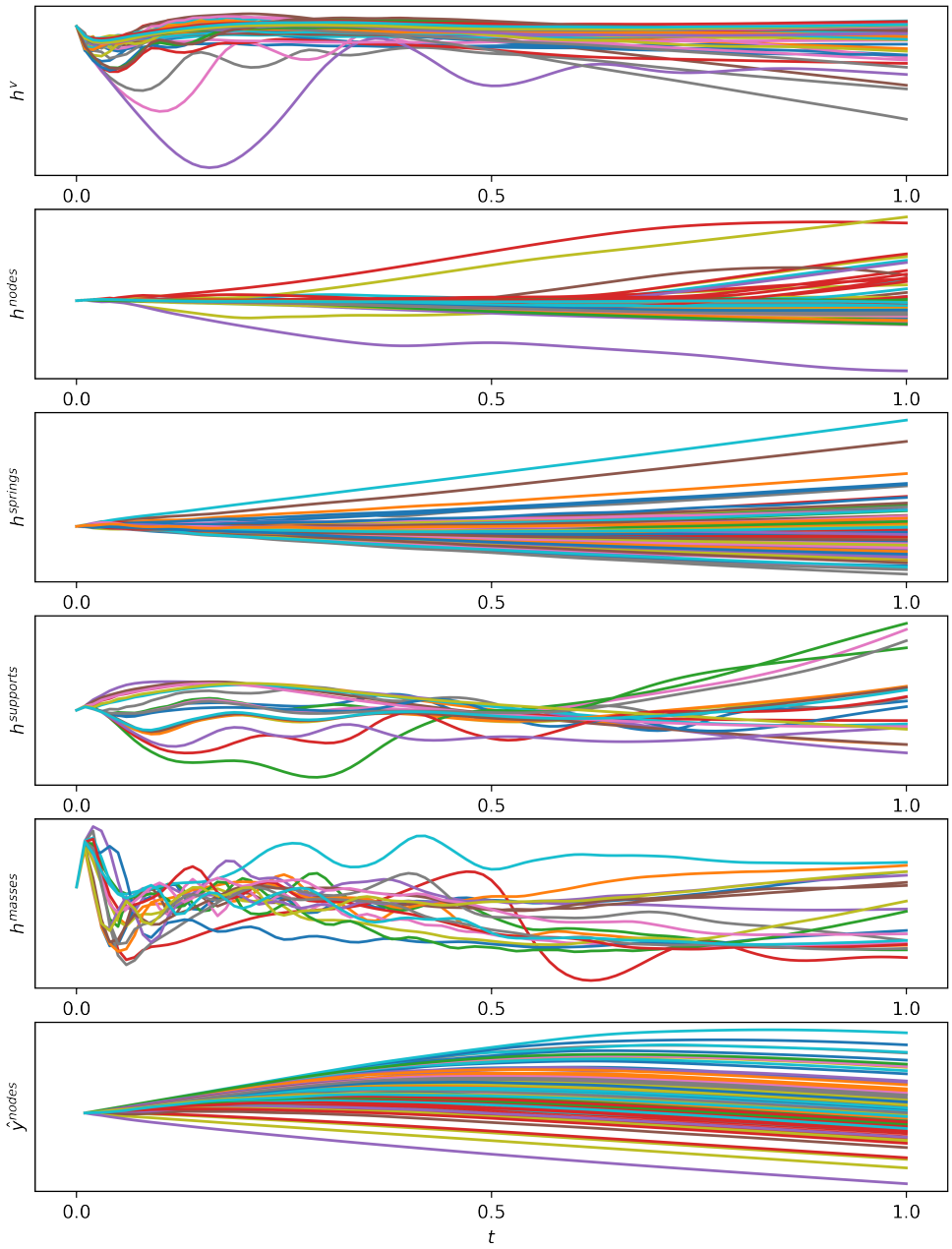


Figure 7.4: Evolution of latent variables h^v, h^{nodes}, $h^{springs}$, $h^{supports}$, $h^{weights}$ and intermediate predictions $\hat{y}^{nodes}$ for a single instance of input grid x, at inference time. All latent variables are initialized at 0, and then proceed to evolve with t, by interacting with each other and with the input data, as described in equations (4.13-4.15) and Algorithm 2. The non-trivial behavior and asynchronous oscillations of latent variables show that the H2MGNN has learned to have the different variables interact with each others.

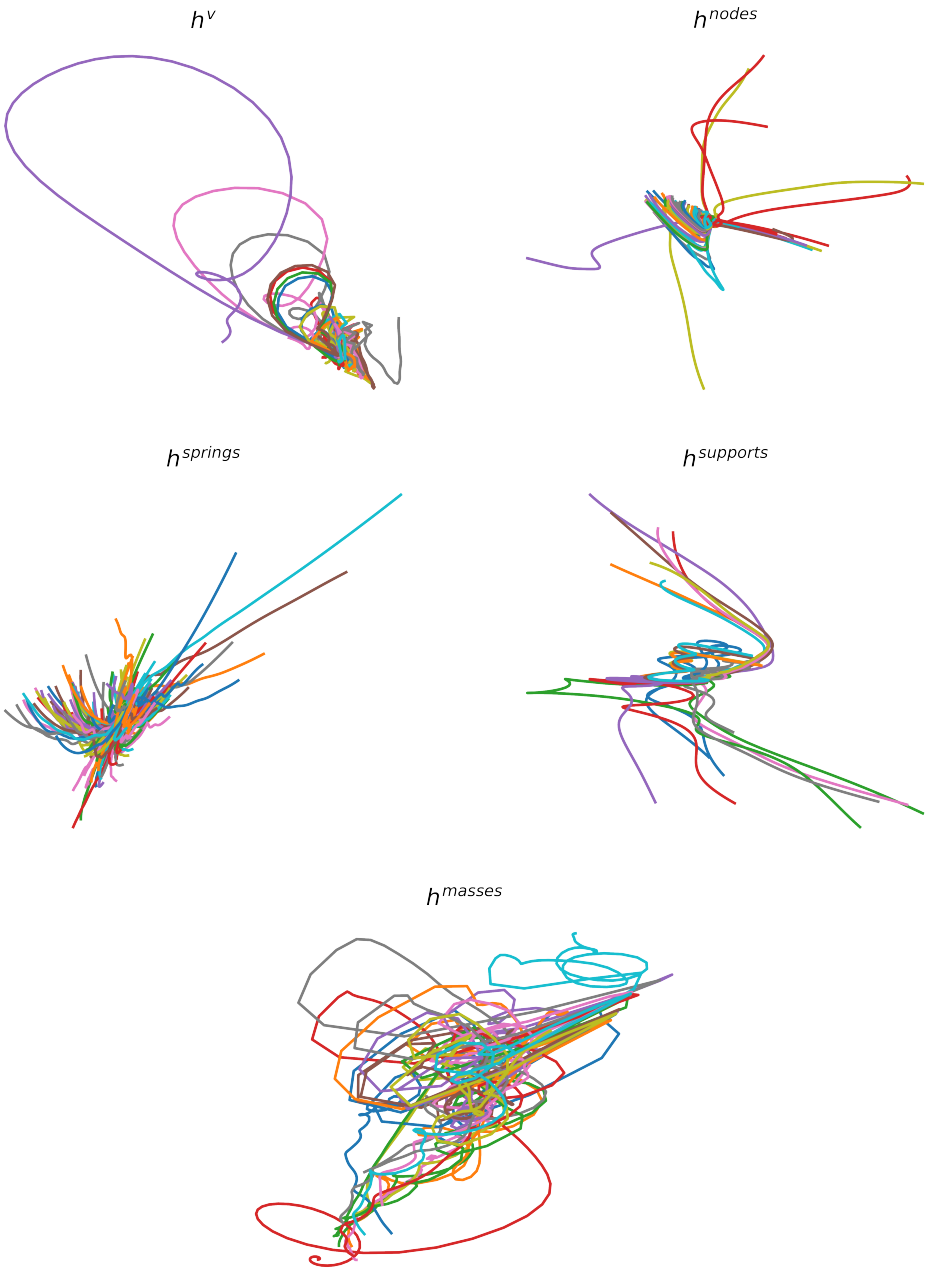


Figure 7.5: Trajectories of latent variables according to their first two dimensions for a single instance of input grid x, at inference time. All latent variables are initialized at 0, and then proceed to evolve with t, by interacting with each other and with the input data, as described in equations (4.13-4.15) and Algorithm 2. The non-trivial behavior of latent variables show that the H2MGNN has learned to have the different variables interact with each others.

7.2 Discretized Poisson equation

The second toy example considered in this chapter comes from the Finite Element Method applied to solve the 2D Poisson equation, one of the simplest and most studied partial differential equation in applied mathematics, and that appears in fluid dynamics, electrostatics, Newtonian gravity, etc. It is commonly solved by first discretizing the spatial domain of definition of the equation into an unstructured mesh, and then converting the differential system into a linear system of equations defined at every node of the mesh. Just like in the previous experiment, it results in solving a linear system of equations. The major distinction lies in the data distribution.

This experiment has been published at NeurIPS 2020 in the paper *Deep Statistical Solvers* [3]. In that paper, the data formalism, the model and the training processes were marginally different from the previous experiment, as detailed in Section 4.A. Despite these changes, those experiments are still largely relevant in the context of this document.

7.2.1 Problem and Data generation

Let us first introduce Poisson's equation and the process of solving it by finite element method.

Formulation of the problem The Poisson equation with Dirichlet boundary condition is defined over a 2D domain Ω with boundary $\partial\Omega$ by:

$$-\Delta u = f \text{ in } \Omega \tag{7.4}$$

$$u = g \text{ in } \partial\Omega \tag{7.5}$$

Generating random geometries Random 2D domains Ω are generated from 10 points, randomly sampled in the unit square. A Bézier curve that passes through these points without any loop is created, and is further subsampled to obtain approximately 100 points in the unit square. These points define a polygon, that is used as the boundary $\partial\Omega$. See the left part of Figure 7.6 to see four instances of domains Ω. The random 2D geometries are discretized using Fenics'[3] standard mesh generation method, as illustrated by the right part of Figure 7.6.

[3] https://fenicsproject.org/

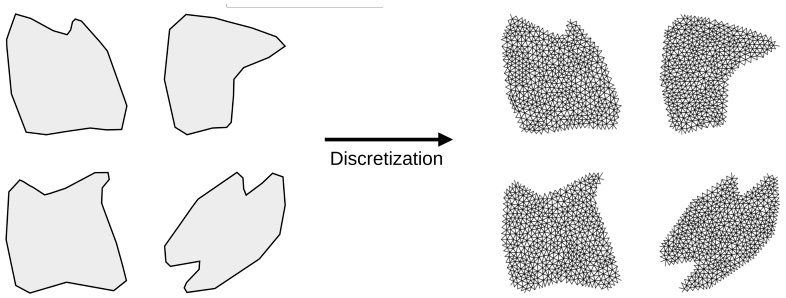


Figure 7.6: Discretization of randomly generated domains

Random functions f and g Functions f and g are chosen to be defined by the following equations:

$$f(v_1, v_2) = r_1(v_1 - 1)^2 + r_2 v_2^2 + r_3 \tag{7.6}$$
$$g(v_1, v_2) = r_4 v_1^2 + r_5 v_2^2 + r_6 v_1 v_2 + r_7 v_1 + r_8 v_2 + r_9 \tag{7.7}$$

where v_1 and v_2 denote the 2D coordinates, and parameters r_i are uniformly sampled between -10 and 10.

Assembling The assembling step [153] consists in building a linear system from the partial differential equation and the discretized domain. The unknowns are the values of the solution at the nodes of the mesh, and the equations are obtained by using the variational formulation of the partial differential equation on basis functions with support in the neighbors of each node. This step is also automatically performed using the Fenics package. The result of the assembling step is a square matrix A and a vector b, and the approximate solution is the vector u such that $Au = b$.

Two distinct types of nodes emerge from this process: those who belong to the boundary $\partial\Omega$ are set to a constant value (analogous to supports from the spring system), while the other nodes are "free" but impose a constant force (analogous to weights). The matrix A is a stiffness matrix and encodes stiffness coefficients that are analogous to springs. We aim at finding a scalar quantity defined at all nodes, which is analogous to the height of nodes in Section 7.1. Even the loss function is the same.

Still, the data formalism in the original paper and implementation only considered two types of objects: nodes and edges, as detailed in Section 4.A.

7.2.2 Experiments

Dataset The dataset consists of 96180/32060/32060 training/validation/test examples from the distribution generated from the discretization of the Poisson equation. Randomly generated 2D geometries and random values for the second-hand function f and boundary condition g are used to compute matrices A and vectors b. Their number of vertices n are around[4] 500 (max 599).

Baselines Two baseline methods are considered, the direct LU decomposition, that could be considered giving the "exact" solution for these sizes of matrices, and the iterative Biconjugate Gradient Stabilized methods (BGS), with stopping tolerances of 10^{-3} . These algorithms are run on an Intel Xeon Silver 4108 CPU (1.80GHz) (GPU implementations were not available, they could decrease LU computational cost by a factor 6 [33]). In addition to the DSS model that learns in an unsupervised fashion, a similar GNN model with the same hyperparameters was trained by imitation of the LU solution, which we refer to as a "proxy". We compare the results of our method with the LU by measuring the Pearson correlation (Corr) and the normalised root mean squared error (nRMSE). The normalization is done by dividing the RMSE by the difference between the largest and smallest values (dividing by the mean for values centered around zeros does not make sense). The 10^{th}, 50^{th} and 90^{th} percentiles of the cost function are displayed.

Model hyperparameters We used our own implementation[5] of the GNN architecture using TensorFlow [151] (no GNN framework were used). The model used is introduced in Section 4.A. In this experiment, there are 30 propagation steps, the hidden dimension is set to 10, each neural network block has a single hidden layer with 10 neurons and a leaky-ReLU activation, and the factor α was set to 10^{-3}. The complete architecture has $49,830$ weights.

Training Training is performed using Adam [117] with a learning rate of 10^{-2} and standard Adam hyperparameters ($\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-7}$), for $280,000$ iterations (48h) with batch size 100. The loss discount factor γ (see Section 4.A) is set to 0.9. In the following, all experiments were repeated three times, with the same datasets and different ran-

[4] Fenics automatic mesh generator does not allow a precise control of n

[5] https://github.com/bdonon/DeepStatisticalSolvers

Method	DSS	Proxy	LU	BGS (tol=1e-3)
Corr. w/ LU	> 0.9999	> 0.9999	-	-
nRMSE w/ LU	1.6e-3	**1.1e-3**	-	-
$\ell\, 10^{th}$ p.	**3.9e-4**	7.0e-3	4.5e-27	1.3e-3
$\ell\, 50^{th}$ p.	**1.2e-3**	1.6e-2	6.1e-26	1.7e-2
$\ell\, 90^{th}$ p.	**4.1e-3**	4.0e-2	6.3e-25	1.1e-1

Table 7.2: Discretized Poisson equation experiment results – Both the unsupervised DSS and the supervised "proxy" achieve good correlation with the LU method. However, the DSS has never seen the output of the LU during training.

dom seeds. We only report the results of the worst of the three trained models.

Results Table 7.2 displays comparisons between a trained DSS and the baselines. These results validate the approach, demonstrating that DSS can learn to solve 500 dimensional problems rather accurately, and in line with the "exact" solutions as provided by the direct method LU (99.99% correlation). While both methods achieve similar results, the DSS was trained in a purely unsupervised way.

Computational times of the LU and BGS methods were estimated on an Intel Xeon Silver 4108 CPU (1.80GHz), while the DSS and "proxy" methods were run on a Nvidia GeForce RTX 2080 Ti. Since a major benefit of deep learning is that it can perform multiple inferences in parallel on GPUs, we consider the inference time divided by the batch size. For all methods, we obtain similar computational time of $2ms$ per instance. Still, comparing computational times of two methods that rely on two different sorts of hardware (cpu vs. gpu) really depends on the actual use case.

Figure 7.7 illustrates, on a hand-made test example (the mesh is on the upper left corner), how the trained DSS updates its predictions, at inference time, along the 30 updates. The flow of information from the boundary to the center of the geometry is clearly visible.

7.2.3 Out-of-distribution Generalization

Varying graph sizes We now experimentally analyze how well a trained model is able to generalize to a distribution that is different from the training distribution. The same data generation process that was used to generate the training dataset is now used with meshes of very different sizes, everything else being equal. Whereas

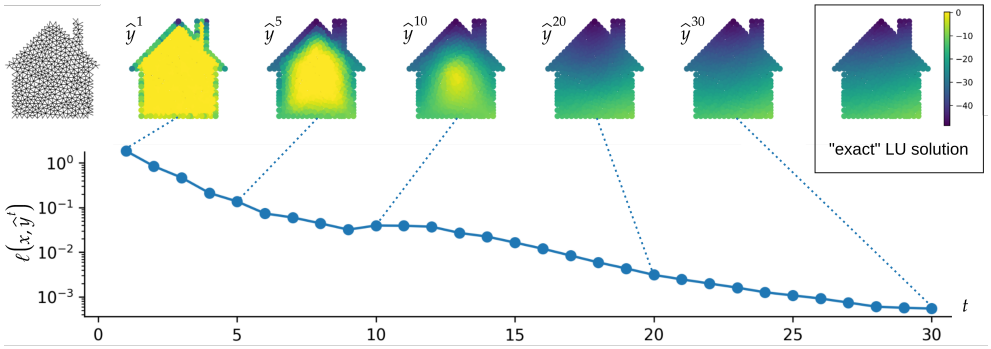


Figure 7.7: Intermediate losses and predictions. Top left: the graph structure; Top right: the LU solution; Bottom: evolution of the loss along the 30 updates for a trained DSS, at inference time. The intermediate predictions $\hat{y}^t$ are displayed for several values of t.

the training distribution only contains graphs of sizes around 500, out-of-distribution test examples have sizes from 100 and 250 (left of Figure 7.8) up to 750 and 1000 (right of Figure 7.9). In all cases, the trained model is able to achieve a correlation with the "true" LU solution as high as 99.9%. Interestingly, the trained DSS achieves a higher correlation with the LU solutions for graphs with fewer nodes, while the correlation of the "proxy" model decreases when n both increases and decreases. Nevertheless, thanks to the specific structure dictated to the linear system by the Poisson equation, DSS was able to perform zero-shot learning [152] for problems of very different sizes.

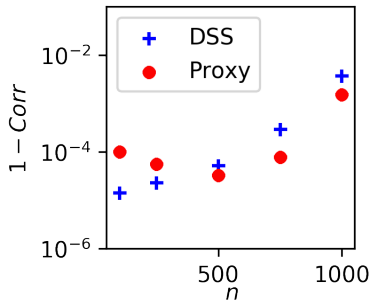


Figure 7.8: Varying problem size n: Correlation (DSS, LU)

Varying features distributions We may now observe the effect of changing the continuous features distribution. In order to change the feature distribution, we alter both the stiffness matrix A and the vector

b, and compute the solution by using the LU method on these altered A and b. Figure 7.9 displays the results of the DSS model, learned on the initial dataset, when increasing alteration is added to the test examples, more and more diverging from the distribution of the training set (the graph size remains unchanged). The alteration is applied by means of a random noise with variance parameterized by τ. Log-normal noise is applied to A ($A_{ij}exp(\mathcal{N}(0,\tau))$, and normal noise to b ($b_i\mathcal{N}(1,)$). Multiple values of noise variance τ were tested, as shown in 7.9. Although DSS results remain highly correlated with the ground truth for small values of τ, they become totally uncorrelated for large values of (correlation close to 0). DSS has learned something specific to the distribution $p(x)$ of linear systems coming from the discretized Poisson EDP.

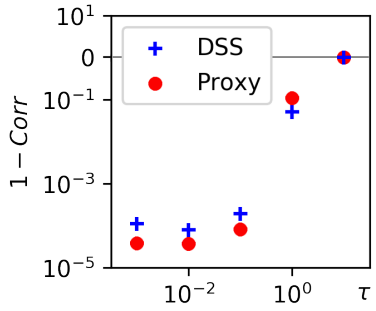


Figure 7.9: Changing feature distribution by varying τ: Correlation (DSS, LU)

This chapter shows results obtained using the DSS approach on linear systems stemming from two distinct domains. While the trained neural networks do not exploit the linearity of the problem, they manage to get extremely accurate results as evidenced by the very good correlation compared to the LU method. Moreover, we prove the viability of the approach for graphs as large as $1,089$ vertices. We also explore the capacity of DSSs to generalize to out-of-distribution samples by distinguishing between the generalization to different graph sizes and different feature distributions. We observe that trained DSSs generalize very well to both larger and smaller graph, underlining a strong robustness with regards to topology changes, which is of primary importance for our power grid problems. Moreover, we observe that altering the feature distribution causes the trained DSSs to drastically decrease in accuracy. One may argue that it is not surprising and that it is not really an issue for power grid

related problems, as the physical quantities tend to always lie in the same distribution (for instance a generator will always produce power between its minimal and maximal possible values).

Chapter 8

AC Power Flow

The AC Power Flow (AC-PF) problem [106, 103] consists in computing the steady-state electrical flows in a power grid knowing the amount of power that is being produced and consumed throughout the grid, the way power lines are interconnected, as well as their physical properties. This non-linear problem is at the heart of real-time power systems operation, and is solved daily for a wide variety of power grid instances using the Newton-Raphson method. As underlined by the GARPUR consortium [12], replacing traditional optimization methods with fast neural networks could be key in developing a probabilistic reliability management approach. Prior to our contributions, most investigated neural network architectures did not take into account topology variations, and were trained by imitation of the output of the Newton-Raphson.

This chapter presents two experiments conducted on this problem using DSSs. First, Section 8.1 experimentally demonstrates the ability of our method to learn in an unsupervised manner to solve the AC-PF problem on two standard benchmarks from the PS literature. Second, Section 8.2 presents ongoing work on scaling up the approach to real-life data from the French power grid. Although the results obtained are not yet satisfactory, this experiment provides valuable lessons while trying to apply the DSS approach to real-life power grids. We thus share our experience and what we believe are opportunities for improvement, in the hope that it will benefit the future research.

8.1 Synthetic data experiments

The first experiment focuses on trying to solve the AC-PF problem on synthetic data. This experiment was published at NeurIPS 2020 in the paper *Deep Statistical Solvers* [3]. We refer readers to the Section 4.A

for more details about the data formalism and neural network architecture used in this experiment.

8.1.1 Problem & Data generation

Experiments are conducted on two standard power grids from the PS literature, namely the *IEEE case14* (n = 14), and the *IEEE case118* (n = 118), as displayed in Figure 8.1. Power injections (production and consumption) are sampled from the time series developed for the *Learning to Run a Power Network competition* [34]. Moreover, in order to increase the diversity in terms of grid topology, for each example there is a 25% chance that a randomly chosen line is disconnected, and 25% chance that two randomly chosen lines are disconnected.

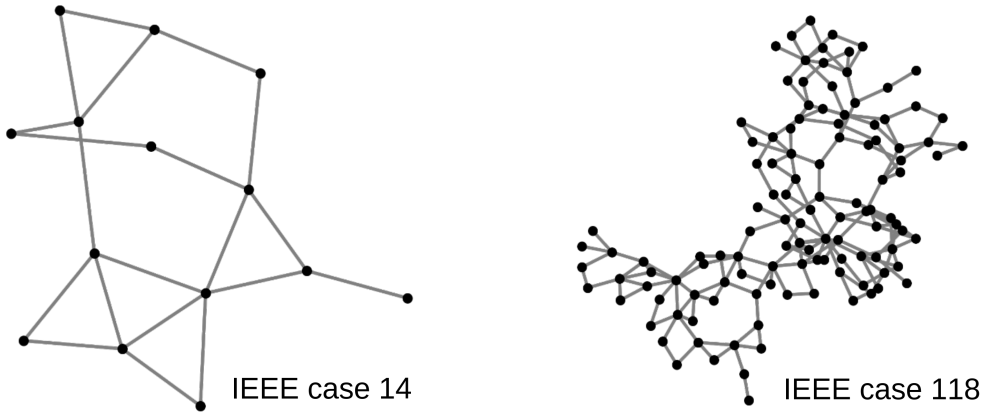


Figure 8.1: Power grid instances used in the synthetic data experiments

In the following, we briefly summarize the input and output features of the considered systems, and refer readers to Chapter 1 for additional information. Six distinct classes of objects compose the considered power grids: $\mathcal{C} = \{buses, loads, generators, shunts, lines, transformers\}$. Only buses have output features.

Buses Buses lie at the interface between the various dipoles and quadrupoles that generate, transport and consume electrical power. Each bus $e \in \mathcal{E}^{buses}$ bears input features $x_e^{buses} = [\mathring{v}_e, \mathbb{1}_e^{pv}, \mathbb{1}_e^{slack}]$ which define its voltage setpoint and if it is a "PV" and/or a "slack" bus. It also bears an output feature $y_e^{buses} = [\hat{v}_e, \hat{\vartheta}_e]$ that define its voltage magnitude and phase angle.

Loads They withdraw power from buses. Each load $e \in \mathcal{E}^{loads}$ bears input features $x_e^{loads} = [\mathring{p}_e, \mathring{q}_e]$ that define their active and reactive power.

Generators They inject power into buses, although their actual behavior is far more complex and detailed in Section 1.3. Each generator $e \in \mathcal{E}^{generators}$ bears input features $x_e^{generators} = [\mathring{p}_e, \mathring{q}_e]$ that define their active and reactive power.

Shunts They have a fixed impedance and are usually used to modulate the reactive power. Each shunt $e \in \mathcal{E}^{shunts}$ bears input features $x_e^{shunts} = [g_e, b_e]$ that define their conductance and susceptance.

Lines They transport electrical power through the coupling of electrical oscillations of both their ends. Each transmission line $e \in \mathcal{E}^{lines}$ bears input features $x_e^{lines} = [r_e, x_e, b_e^c]$ that define their resistance, reactance and total line charging susceptance.

Transformers They transport electrical power through the coupling of electrical oscillations of both their ends, and can interconnect different voltage levels. Each transmission line $e \in \mathcal{E}^{lines}$ bears input features $x_e^{lines} = [r_e, x_e, b_e^c, \tau_e, \vartheta_e^{shift}]$ that define their resistance, reactance, total line charging susceptance, ratio and phase shift.

Cost function All devices inject electrical power into the bus they are connected to. Their respective behaviors are detailed in Section 1.3. For each bus $e \in \mathcal{E}^{buses}$, the complex power mismatch[1] is given by Δs_e. To satisfy Kirchhoff's laws, it should be zero at every bus simultaneously. Thus, we use as a cost function the following:

$$\ell(x, y) = \sum_{e \in \mathcal{E}^{buses}} |\Delta s_e|^2 \tag{8.1}$$

8.1.2 Experiments

Dataset For case14 (resp. case118), the dataset is split into 16064/2008/2008 (resp. 18432/2304/2304) samples.

[1] discarding active power mismatch for the slack bus, and reactive power mismatches for voltage controlled buses

Baselines State-of-the-art AC power flow computation rely on the Newton-Raphson method, used as baseline here (using the pandapower[154] implementation, on an Intel i5 dual-core (2.3GHz)). To the best of our knowledge, no GPU implementation was available, although recent work [155, 156] investigates such an avenue. We compare the results of our method with the Newton-Raphson by measuring the Pearson correlation (Corr) and the normalised root mean squared error (nRMSE). Those metrics are computed by comparing the results at every vertex (or line) of every sample in the test set. The normalization is done by dividing the RMSE by the difference between the largest and smallest values (dividing by the mean for values centered around zeros does not make sense). The 10^{th}, 50^{th} and 90^{th} percentiles of the cost function are displayed. We also compare the DSS to the "proxy" approach: the architecture is strictly the same, but the loss function used during training is the distance to the "ground truth" (provided by the Newton-Raphson method).

Model hyperparameters We used our own implementation[2] of the GNN architecture using TensorFlow [151] (no GNN framework were used). The model used is detailed in Section 4.A. For the 14 nodes case (resp. 118 nodes), there are 10 (resp. 30) propagation steps, the hidden dimension is set to 10, each neural network block has a single hidden layer with 10 neurons and a leaky-ReLU activation, and the factor α is set to 10^{-3}. The complete architecture has $17,220$ (resp. $51,660$) weights.

Training Training is performed using Adam [117] with a learning rate of 10^{-2} and standard hyperparameters, for $883,000$ (resp. $253,000$) iterations (48h) with batch size $1,000$ (resp. 500), on an Nvidia GeForce RTX 2080 Ti. The discount factor γ is set to 0.9.

Results In both cases, correlations between power flows output by the trained DSSs and the Newton-Raphson method are above 99.99% (both active p_{ij} and reactive q_{ij}). The same can be said for the "proxy" models. However, one can observe a less satisfactory correlation in terms of v_i and ϑ_i for the DSSs while the proxies maintain a correlation higher than 99.99%. This can be explained by the fact that the DSSs minimizes power mismatches while the proxies minimize the distance to the Newton-Raphson output in terms of v_i and ϑ_i. However, this does not impact the quality of the power flow prediction. Our DSS model is

[2] https://github.com/bdonon/DeepStatisticalSolvers

Dataset		IEEE 14 nodes			IEEE 118 nodes		
Method		DSS	Proxy	NR	DSS	Proxy	NR
Corr. w/ NR	v_i	.9993	$>$ .9999	-	.9979	$>$ .9999	-
	ϑ_i	.9986	$>$ .9999	-	.8131	$>$ .9999	-
	p_{ij}	$>$ .9999	$>$ .9999	-	$>$ .9999	$>$ .9999	-
	q_{ij}	$>$ .9999	$>$ .9999	-	$>$ .9999	$>$ .9999	-
nRMSE w/ NR	v_i	2.0e-3	4.9e-4	-	1.4e-3	1.2e-3	-
	ϑ_i	7.1e-3	1.7e-3	-	5.7e-2	4.5e-3	-
	p_{ij}	6.2e-4	2.6e-4	-	1.0e-3	3.9e-4	-
	q_{ij}	4.2e-4	2.0e-4	-	1.1e-4	1.7e-4	-
Loss 10^{th} p.		4.2e-6	2.3e-5	1e-12	1.3e-6	6.2e-6	3e-14
Loss 50^{th} p.		1.0e-5	4.0e-5	2e-12	1.7e-6	8.3e-6	4e-14
Loss 90^{th} p.		4.4e-5	1.2e-4	3e-12	2.5e-6	1.3e-5	6e-14

Table 8.1: Our trained DSS models are highly correlated with the Newton-Raphson solutions.

able to learn accurate predictions of v_i and ϑ_i, without having observed the output of the Newton-Raphson during training. As a result, our approach is a completely independent optimization method that does not rely on the imitation of potentially expensive optimization techniques.

Computational time of the Newton-Raphson method was estimated on an Intel i5 dual-core (2.3GHz), while the DSS and "proxy" methods were run on a Nvidia GeForce RTX 2080 Ti. As evoked in Section 7.2, deep neural networks can perform multiple inferences in parallel on GPUs, so we consider the inference time divided by the batch size (the batch size being chosen to be as large as possible, while not saturating the GPU). For the 14 nodes (resp. 118 nodes) dataset, we obtain $10^{-2}ms$ (resp. $2 \times 10^{-1}ms$) per instance for the DSS and "proxy" method, and $20ms$ (resp. $20ms$) for the Newton-Raphson, which provides a speed-up of 3 (resp. 2) orders of magnitude. However, we object that comparing computational times of two methods that rely on two different sorts of hardware (cpu vs. gpu) really depends on the use case. Nevertheless, these results highlight the fact that our neural network based method is competitive in regard to computational times.

8.2 Real data experiments

After having experimentally validated the approach over relatively small artificial networks, this work has been focusing on applying the DSS approach to real-life data from the French power grid owned and operated by RTE. This has highlighted several difficulties, that will be detailed hereafter. After having detailed the major obstacles to the scaling up of the approach, we provide and interpret preliminary results, which should be refined in future work (see Section 9.4). Due to time constraints, this is still an ongoing work, and the objective of this section is to report the current state of progress.

8.2.1 Major difficulties

First of all, let us dwelve into the aspects that were the most detrimental to this line of work.

Interfacing Real data from the French power grid include much more information than included in the "simplistic" model introduced in Chapter 1. Building features that compose the input data x requires to search through multiple data frames and to combine several of them, simply to compute the actual value of some coefficients. Features are not easily accessible and require an in-depth knowledge of power grids in general, of the specific French power grid, and of the format used to store power grid snapshots in particular. Nevertheless, the recent open-source suite of tools pypowsybl[3] developed by RTE results in a much easier and faster interfacing with real-life data, although some features are still missing.

Hidden modelling Computational methods that use real-life data sometimes resort to heuristics that are not well documented. For instance, some feature values may be considered as abnormal by the algorithm, and are replaced by a default value. In other words, the actual cost function ℓ used by traditional methods is somewhat altered by these heuristics. These hidden computations should thus also be included in the training phase, so as to allow for a fair comparison between methods.

For all the above reasons it is clear that if one aims at working on real-world data, a series of tests and data cleaning processes should be implemented. Systematic data assessment should check data validity

[3] https://github.com/powsybl/pypowsybl

and realism. The DSS method is sensitive to errors on either the input data x or the cost function ℓ. It is essential that data samples are in adequacy with the model underlying x, y and ℓ.

Ill-conditioned cost function Beyond numerical issues caused by the interfacing with real-life data, the optimization problem is intrinsically complex. The gradient used to train the parameters θ of our model may be decomposed into two terms:

$$\nabla_\theta \ell(x, f_\theta(x)) = \nabla_\theta f_\theta(x).\nabla_y \ell(x, f_\theta(x)) \tag{8.2}$$

where $\nabla_y \ell(x, f_\theta(x))$ denotes the gradient of ℓ with regards to its second argument, and estimated at $(x, f_\theta(x))$. The first term expresses the sensitivity of the model's output with regards to its parameters θ, and the second expresses the sensitivity of the cost function ℓ with regards to its second argument. However, it is possible that the the problem is ill-conditionned: gradient $\nabla_y \ell(x, f_\theta(x))$ provides a poor estimation of the best direction to follow to decrease ℓ.

- In a well-conditioned problem, the first order gradient $\nabla_y \ell(x, f_\theta(x))$ estimated at any location of the space $\mathcal{Y}$ points approximately in the direction of the actual solution. Thus, back-propagating this first order derivative into the model f_θ should quickly have the model converge towards a decent mapping.
- In an ill-conditioned problem, the optimization landscape is skewed, and the first order gradient $\nabla_y \ell(x, f_\theta(x))$ is very likely to point far from the right direction.

Unfortunately, ill-conditioned cost functions arise when considering real-life data: some transmission lines can be as long as several kilometers, while others are as short as a few meters. This creates extremely loose couplings between some variables, and extremely stiff couplings between others. As a consequence, the first order derivative of the cost function is very likely to give erroneous information regarding the actual direction of the solution.

Still, a possible solution to address this issue could be some preconditionning, so as to provide a better estimation of the best direction to follow. For instance, in the case where $\ell(x, y)$ is a quadratic function of y, correcting the gradient with the inverse Hessian matrix ($H_{ij}(\ell) = \frac{\partial^2 \ell}{\partial y_i \partial y_j}$) provides the exact direction of the actual solution in the space $\mathcal{Y}$. Instead of using the simple first order gradient of equation (8.2), one could instead use the following corrected gradient:

$$\nabla_\theta f_\theta(x).H(\ell)^{-1}.\nabla_y \ell(x, f_\theta(x)) \tag{8.3}$$

This requires to invert the potentially large matrix $H(\ell)$ during training, which can be computationally expensive. Still, one may argue that this computational burden is still limited to the training phase, and has no impact whatsoever on the inference speed of a trained model. Pre-conditionning the back-propagation algorithm is yet to be experimented with.

Message passing bottleneck At some point, it has been thought that the poor results obtained on real data could be caused by the fact that propagating information using only local message passing was not suitable for large graphs (the French grid is made of approximately $6,000$ buses, and has a diameter around 80). However, the following two observations seem to indicate that local message passing is not incompatible with an application to power grids that have large diameters:

- Learning on real data is as difficult on small portions extracted from the French grid as it is on the full system (which supports the hypothesis of the ill-conditioning of the cost function ℓ for real-life instances). For instance, the exact same issues appear if we only take into account the $400kV$ part of the network in the Lyon region (approx 50 buses).

- Experiments shown in Section 7 provide good results on large graphs (up to $1,089$ nodes).

8.2.2 Early experiments

Despite the various setbacks encountered trying to have the approach work on real-life data, the present subsection details and interprets preliminary results. Although there are currently no results for the actual unsupervised DSS approach (*i.e.* learning by minimizing the violation of Kirchhoff's law), we obtained decent performance by training a model in a supervised way (*i.e.* by imitation of the Newton-Raphson method). While previous experiments on the IEEE 14 nodes and 118 nodes included both an unsupervised H2MGNN approach and a supervised "proxy", we only managed to obtain relevant results for the supervised H2MGNN "proxy" model on real-life data. Experiments using the unsupervised DSS approach on real data failed up to now probably because of the ill-conditioning issue previously described. This "proxy" uses the latest version of the model, as described in Algorithm 2 in Section 4.2.

Dataset The training set consists of $2,500$ snapshots of the French power grid randomly picked from the year 2018, while the test set consists of $2,500$ snapshots from 2019. For now, only the extra high voltage network of the Lyon region (approx. 200 buses) is considered.

Metrics We compare a H2MGNN "proxy" (trained by imitation of the Newton-Raphson method) with the Newton-Raphson. We compute the Pearson correlation (Corr), the normalised MAE (nMAE) and RMSE (nRMSE) (normalized by dividing by the difference between max and min values), and estimate the 10^{th}, 50^{th} and 90^{th} percentiles of the cost function ℓ.

Hyperparameters They have not been subject to a thorough hyperparameter tuning, as this is still an open line of research. Currently, we use a time step $\Delta t = 1/50$, a latent dimension $d = 40$, and each fully connected neural network block has two hidden layers, with 80 hidden neurons each, and a hyperbolic tangent ($\tanh$) activation function.

Training We used our own implementation of the H2MGNN architecture using TensorFlow [151] (no GNN framework were used). Training is done for $10,000$ epochs with batch size 100, and lasted 27h on an NVIDIA TITAN Xp. The ADAMAX optimizer [117] with parameters $(lr = 10^{-1}, \beta_1 = 0.99, \beta_2 = 0.9999)$ is used.

Results As shown in Table 8.2, we managed to obtain quite decent results on the test set: a 92% correlation with NR method in terms of v_i, and a 96% correlation in terms of ϑ_i. Both the normalized RMSE and normalized MAE compared to the NR solution are around 2×10^{-2}, which is a decent score. Still, predictions provided by the H2MGNN model are far from being satisfying in regard to the industrial stakes. The two 2D histograms at the top of Figure 8.2 show that while predictions are mostly accurate (colors are in log scale), there is a non negligible amount of highly erroneous predictions.

All metrics for the active (p_{ij}) and reactive (q_{ij}) power flows are consistently not acceptable. The two 2d histograms at the bottom of Figure 8.2 allow for a more precise interpretation of those metrics. Predictions are made in terms of v_i and ϑ_i, and then power flows through transmission lines and transformers are computed using physical equations. While most lines have somewhat decent predictions, there is a large portion of lines that have erroneous predictions.

After investigation, it appears that lines that have a smaller reactance also have more erroneous predictions in terms of p_{ij} and q_{ij}.

Method		Proxy	NR
Corr. w/ NR	v_i	0.92	-
	ϑ_i	0.96	-
	p_{ij}	0.44	-
	q_{ij}	0.11	-
nRMSE w/ NR	v_i	2.8e-2	-
	ϑ_i	1.9e-2	-
	p_{ij}	1.7e-1	-
	q_{ij}	2.7e-1	-
nMAE w/ NR	v_i	1.5e-2	-
	ϑ_i	1.3e-2	-
	p_{ij}	3.8e-2	-
	q_{ij}	7.4e-2	-
ℓ 10^{th} percentile		34	1.2e-7
ℓ 50^{th} percentile		62	2.0e-7
ℓ 90^{th} percentile		124	1.5e-5

Table 8.2: Metrics of H2MGNN "proxy" predictions on real-life data.

Since both active and reactive flows are somewhat proportional to the inverse of the reactance, errors in terms of v_i and ϑ_i are amplified at transmission lines that have a low reactance. Since the reactance of transmission lines in real data can vary up to almost three orders of magnitude, we observe at the same time quite decent predictions on some lines, and extremely bad predictions on others.

Moreover, the vertical lines that appear at the bottom left of Figure 8.2 are caused by extremely small lines that are connected to single loads whose consumptions appear to be almost constant throughout the year.

This chapter considers the application of the DSS approach to the AC-PF problem. Excellent results are obtained on artificial data, even though the neural network has been trained in a completely unsupervised manner. However, experiments conducted on the real-life data underlined several major difficulties, which will be addressed in future work.

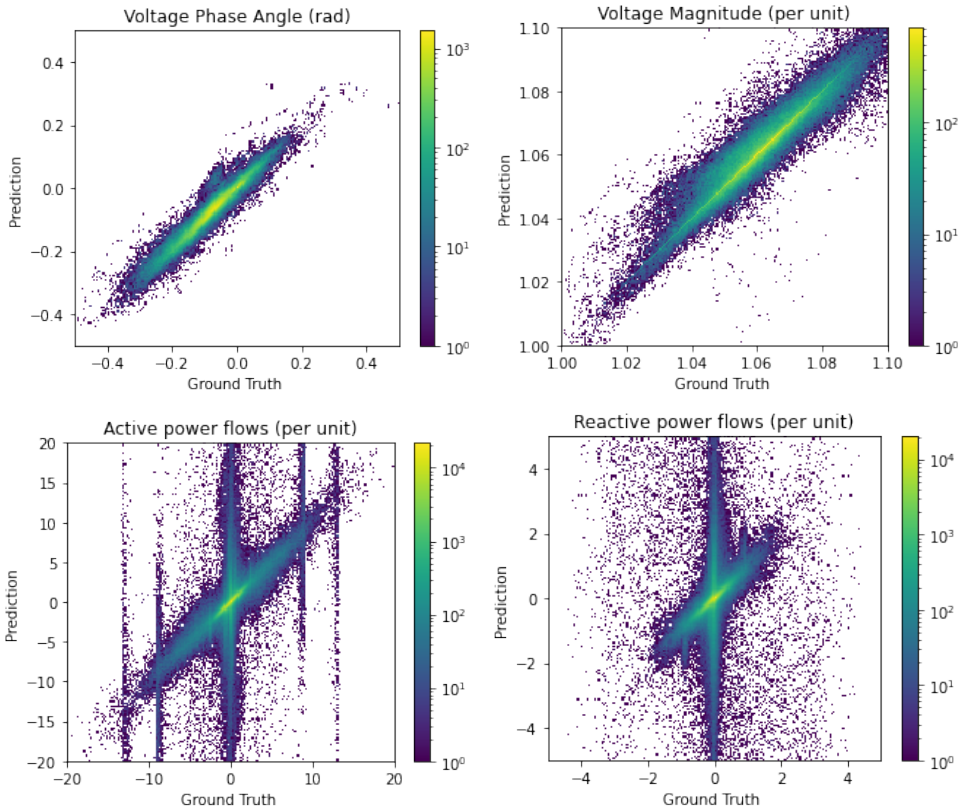


Figure 8.2: 2D histograms comparing H2MGNN "proxy" predictions to Newton-Raphson solutions. Bin colors are in log-scale. While predictions for v_i and ϑ_i are somewhat decent, both active and reactive power flows have poor predictions. Bins are squares of respective dimensions 5×10^{-3} rad for the voltage phase angles, 5×10^{-4} p.u. for the voltage magnitudes, 2×10^{-1} p.u. for the active power flows and 5×10^{-2} p.u. for the reactive power flows.

Part IV

Conclusion & Future Research

Chapter 9

Discussion and future work

Power grids cannot be reduced to the sole resolution of the AC-PF. In order to keep the system in security, dispatchers continuously make a multitude of decisions that involve various physical quantities, multiple time scales and numerous sources of uncertainty. Thus, the DSS approach should be able to address multiple classes of decision-making problems, and to consider the actual real-life power grid in its full complexity.

This chapter is an attempt at exploring the various ways the DSS approach should be extended. In Section 9.1, we present preliminary results on the unsupervised learning of a DSS aimed at controlling voltage, relying on the bilevel approach introduced in Section 5.2. In Sections 9.2 and 9.3, we address the issues of incorporating time and uncertainties in power grid management, and propose a way to seamlessly include both aspects in the DSS approach. In Section 9.4, we succinctly explore major axes for future improvements. Finally, in Section 9.5, we outline what could be an artificial assistant to dispatchers based on a DSS.

9.1 Voltage control

RTE has been facing an increase in the frequency of high voltage violations for the past decade, which may damage infrastructures. Several changes in the French power system contribute to an overall rise of the voltage magnitude across the grid:

- The development of diffuse renewable energies, which reduce the load as seen from the transmission network, thus reducing power flows across transmission lines, and therefore decreasing reactive losses in the lines,

- The burial of overhead lines, as the cables that replace them have a higher capacitive behavior,
- The replacement of domestic devices by new ones whose reactive energy consumption is different than in the past.

Managing these problems implies spending a significant amount of time in studies for the operators, time that they do not necessarily have. To date, there is no satisfactory decision support tool to tackle this problem of voltage management.

Thankfully, the bilevel DSS introduced in Section 5.2 could allow us to train a GNN to map power grid instances to generator voltage setpoints, so as to control (in open-loop) the voltage. As described in Figure 9.1, this method relies on the joint learning of two neural networks, one playing the role of a dispatcher (controller$_\theta$) and the other playing the role of a physics simulator (solver$_\omega$). In order to be consistent with the formalism introduced in Section 5.2, we denote by x the power grid instances, by y the voltage setpoints and by z the voltage magnitudes and phase angles. The controller parameterized by $\theta \in \Theta$ takes x as input and outputs y, and the solver parameterized by $\omega \in \Omega$ takes a pair (x, y) as input and outputs z. Both neural networks are trained jointly (by alternating between a solver and a controller update), as described in Algorithm 5.

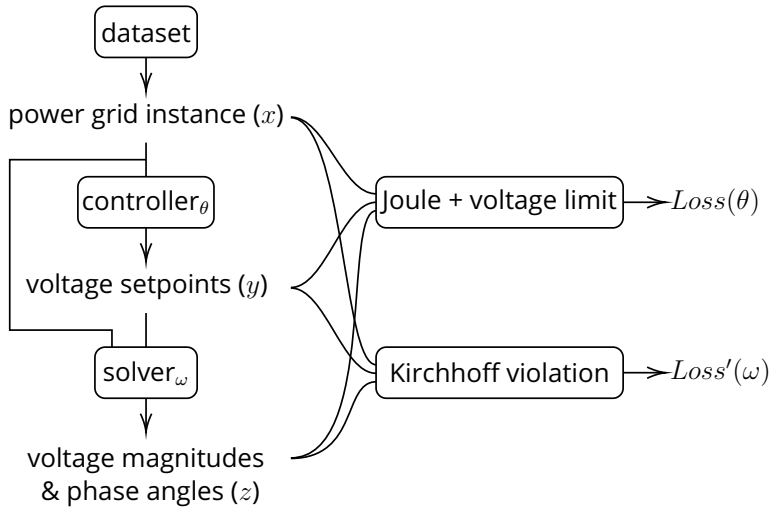


Figure 9.1: Voltage setpoint control. Two neural networks, namely controller$_\theta$ and solver$_\omega$, are trained jointly.

This work has been carried out in collaboration with Guillaume Houry and Maxime Sanchez during their respective internships at RTE

R&D. In the following, we present preliminary results which should be further refined in future work.

9.1.1 Problem & data generation

Experiments are conducted on the IEEE case14 ($n = 14$) power grid, as shown on the left part of Figure 8.1. Injections are sampled from the time series generated for the *Learning to Run a Power Network competition* [157]. For each sample, there is a 25% probability that one randomly chosen power line is disconnected, and a 25% probability that two distinct randomly chosen power lines are disconnected.

Input and output features are detailed thereafter, and additional information about power grid modelling is available in Chapter 1. Power grids are composed of six classes of objects: $\mathcal{C} = \{buses, loads, generators, shunts, lines, transformers\}$. While most input features are exactly the same as in Section 8.1, the main difference lies in buses. Only buses have a controller output $y = (y^{buses})$ and a solver output $z = (z^{buses})$.

Buses Each bus $e \in \mathcal{E}^{buses}$ bears an input feature $x_e^{buses} = [\overline{v}_e, \underline{v}_e, \mathbb{1}_e^{pv}, \mathbb{1}_e^{slack}]$ that define its maximal and minimal voltage magnitudes and if it is a "PV" or a "slack" bus. In this experiment, the acceptable range of voltage magnitudes is $[0.95p.u., 1.05p.u.]$. However, we choose to train the model using a smaller range as a security. For all buses, we thus consider $\underline{v} = 0.96p.u.$ and $\overline{v} = 1.04p.u.$. The controller output $y_e^{buses} = [\mathring{v}_e]$ defines the voltage setpoints that shoud be used if the bus is "PV". The solver output $z_e^{buses} = [\hat{v}_e, \hat{\vartheta}_e]$ defines the voltage magnitude and phase angle for each bus of the grid. Depending on whether a bus is "PV" or not, its voltage magnitude is set to $\mathring{v}_e$ (contained in y) or to $\hat{v}_e$ (contained in z).

Controller cost function The dispatcher aims at keeping the power grid in security. Regarding voltage control, its main requirement is to have the voltage magnitude of each bus $e \in \mathcal{E}^{buses}$ be in the acceptable range $[\underline{v}_e, \overline{v}_e]$, which is quantified by Δv_e in equation 1.16. Moreover, we consider as a secondary objective the minimization of the electrical losses caused by Joule's effect, which is defined at every line and every transformer by p^{Joule} in equations (1.23) and (1.27). Thus, the controller cost function ℓ is defined by:

$$\ell(x, y, z) = \mu \times \sum_{e \in \mathcal{E}^{buses}} |\Delta v_e| + \sum_{e \in \mathcal{E}^{lines} \cup \mathcal{E}^{transformers}} p_e^{Joule} \qquad (9.1)$$

The factor $\mu > 0$ is tuned so that the priority is given to ensuring that the voltage is in the acceptable range. It is set to 10^4 in this experiment.

Solver cost function All devices inject power in their respectives buses, as detailed in Section 1.3. The active power mismatch of bus $e \in \mathcal{E}^{buses}$ is given by Δs_e. To respect Kirchhoff's laws, all bus mismatches should be zero simultaneously. The solver cost function ℓ' is thus:

$$\ell'(x, y, z) = \sum_{e \in \mathcal{E}^{buses}} |\Delta s_e|^2 \tag{9.2}$$

9.1.2 Experiments

Dataset The training set is made of $10,000$ samples and the test set of 300 samples.

Baselines Our main focus is to learn a controller, while the training of a solver is only a means to that end. Thus, we only consider the quality of the controller part, which we assess by replacing the neural network solver that was used for training with an actual Newton-Raphson solver. We compare the obtained results with that of a particle swarm [158] algorithm: a black box optimization method that relies on a stochastic exploration of the set of voltage setpoints. Future work will include a comparison with a more rigorous (and slower) optimization method. Still, this allows us to have a fast and quite efficient baseline.

Model hyperparameters Both the controller and the solver use the model detailed in Section 4.A, with the same hyperparameters. There are 10 propagation steps, the hidden dimension is set to 20 and each neural network block has 4 hidden layers with 20 hidden neurons and a hyperbolic tangent ($\tanh$) activation function. The update scaling factor α is set to 10^{-3}.

Training Training is performed using Adam [117] with a learning rate of 10^{-5} for the controller and 10^{-4} for the solver. Other parameters of Adam are standard ($\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-7}$). Gradient clipping is used (over the norm) with a threshold of 10^2. Moreover, as explained in Section 5.2, when performing a step to learn the controller, we add a Gaussian noise with 0 mean and a variance of 2×10^{-2} over the output of the controller. This allows the solver to explore more different values for y, and have a better understanding of the actual dependency

	DSS	Baseline
Cumulated Joule losses (MW)	215.7	215.4
High voltage violations	0	0
Low voltage violations	1	1

Table 9.1: Cumulated performances of the bilevel DSS controller and the particle swarm baseline. Both obtain similar results.

of z with regards to y. The training is performed for $5,245$ epochs on a NVIDIA TITAN Xp, using minibatches of 100 samples. The discount factor γ is set to 0.9. .

Results The voltage setpoints output by the trained controller provide similar results as the patricle swarm baseline in terms of cumulated Joule losses over the whole test set. A single low voltage event occurs for both the baseline and the DSS, which seems to indicate that there is a single snapshot for which there is no solution. As highlighted by Figure 9.2, the results are similar for both methods in terms of Joule losses, regardless of whether lines have been disconnected or not. Still, the DSS method is a fast GNN-based heuristics trained in an unsupervised manner, while the particle swarm method is a black-box optimization method that requires to explore the space of possible voltage setpoints using numerous calls to the Newton-Raphson method.

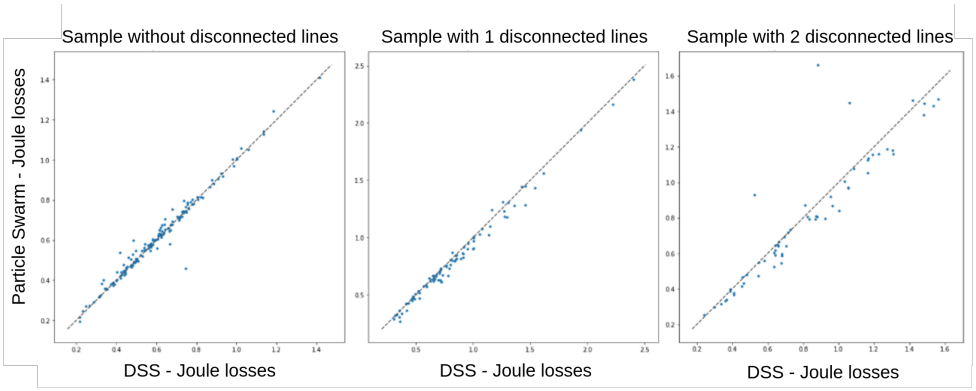


Figure 9.2: Correlation plots of the Joule losses caused by the output of the DSS vs. caused by the output of the particle swarm. Although some power grid instance depart from the diagonal, the two methods provide quite similar results.

Figure 9.3 shows the evolution of the cost function ℓ in the neighborhood of the output of the trained controller, by changing one voltage

setpoint at a time. For a single power grid instance x and for all 5 voltage setpoints, we observe that the prediction of our trained controller is very close to the minimum of the function, while always respecting the voltage limits.

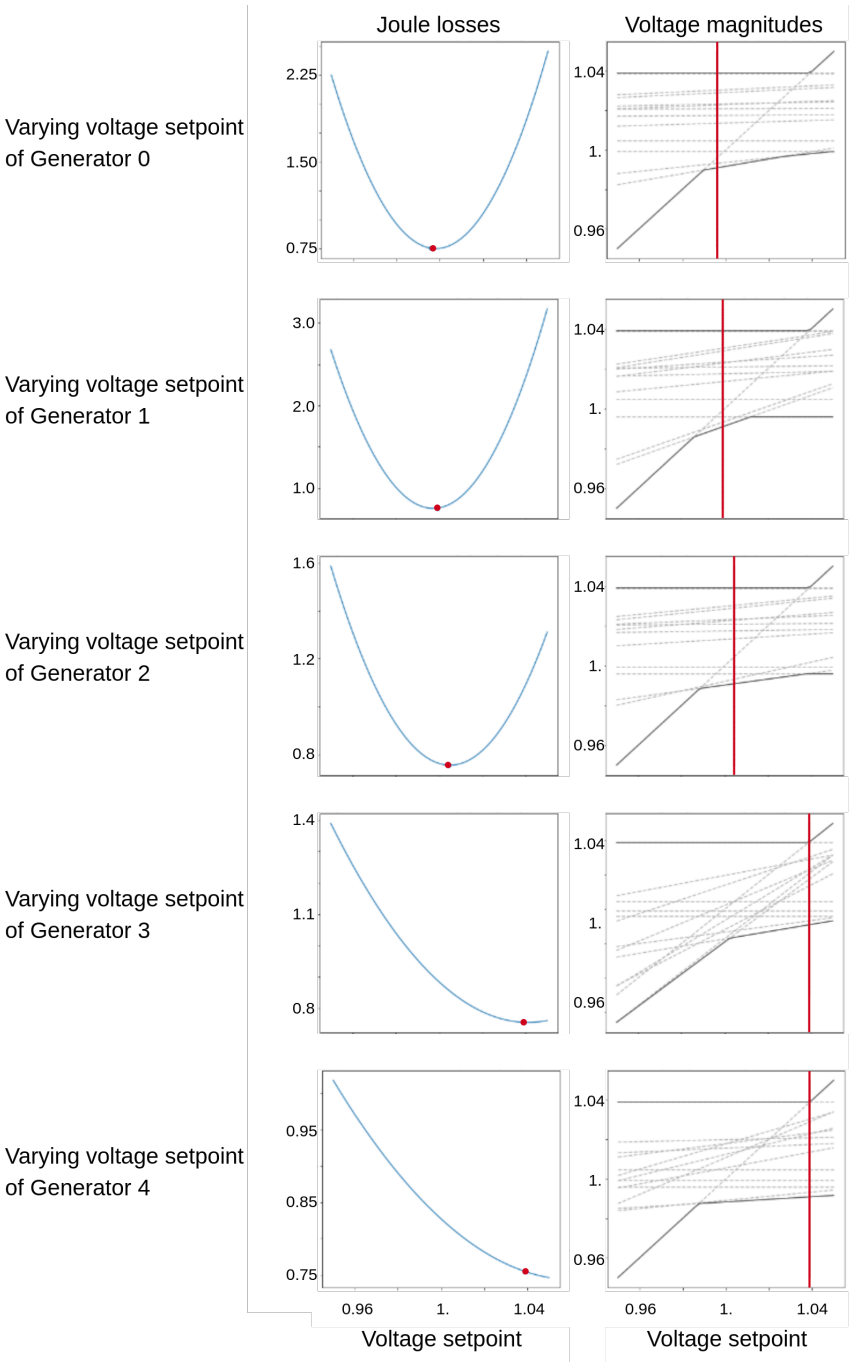


Figure 9.3: Evolution of Joule losses and voltage magnitudes in the neighborhood of the output of a trained bilevel DSS, for a single power grid instance. The output of the DSS is in red. For each of the four generators, the voltage setpoint is either close to the minimizer of Joule's effect, or to the maximum value (set to 1.04).

9.2 Incorporating time

Time plays a key role in most problems related to power grid operation. Production and consumption evolve constantly and one-time events such as line disconnections or incidents may alter the grid topology at any moment. Decisions must be taken in accordance with the temporal variations and constraints of the considered system. As a consequence, it is of the utmost importance to be able to consider time series as inputs and/or as outputs. In the present section, we first discuss the case of graph time series where the topology (*i.e.* graph structure) is constant through time, and explain how such a case seamlessly fits into our formalism. We then underline challenges caused by variations of topology, and propose a possible solution.

In the following, $t \in [T]$ refers to time steps of a time series and should not be mistaken for the t used in Architecture 2.

9.2.1 Factorizing through time

Graph time series can easily fit into the DSS formalism as long as all snapshots share the same graph structure $(n, \mathcal{C}, \mathcal{E}, \mathcal{M})$. We refer to such graph time series as *factorizable*. In this case, snapshots can be written as follows: $x = (x^c_{e,m})_{(c,e,m)\in\mathcal{G}_x}$ where $x^c_{e,m} = (x^c_{e,m}(t = 1), \ldots, x^c_{e,m}(t = T))$, which gives rise to features in $T \times d^{c,x}$ dimensions. Figure 9.4 shows the distinction between factorizable and non factorizable graph time series.

Depending on the problem at hand, we may want to output either a single graph y, or another graph time series. Either way, y should respect the graph structure of the input. Moreover, cost functions need not be decomposable with regards to time, and in the general case the cost function still writes as $\ell(x, y)$. Basically our framework remains unchanged, at the exception of what is modelled by the pair (x, y) and what is hidden in the cost function ℓ.

Unfortunately, power grid time series are known to have a changing topology, making them non-factorizable. As a consequence, they do not directly fit into the DSS formalism. A possible work-around is to change the data representation so as to give rise to a factorizable representation.

9.2.2 Factorizable representations

In the general case, there exists no intrinsic way of representing a graph time series in a factorizable way. However, depending on the problem

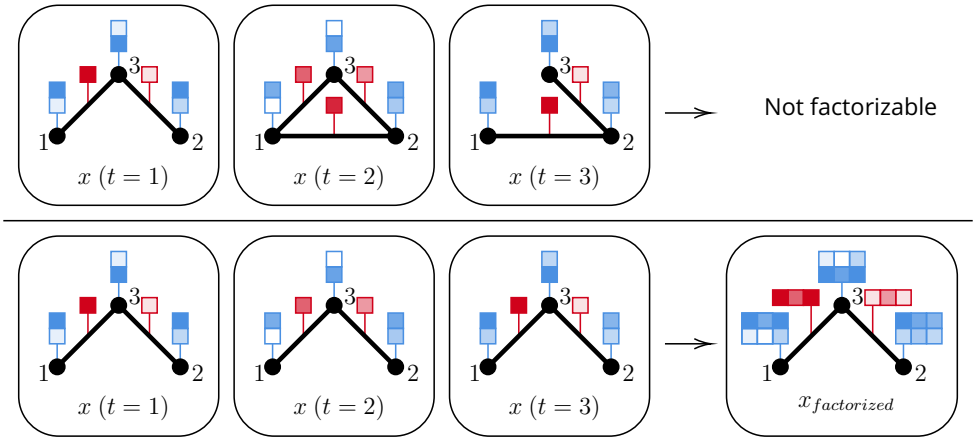


Figure 9.4: Non-factorizable graph time series (above) vs. Factorizable graph time series (below) – Having a constant graph structure over time allows to write the time series in a factorized form. This gives rise to a single graph that is compatible with our framework.

at hand, it may be possible to change the data representation so that topology remains constant through time. One should thus seek a data representation where only hyper-edge features vary with time.

In the case of power grids, we consider real-life infrastructures that do not change drastically over time. Interconnection patterns change on a daily basis but the space of possible topologies is quite restrained: transmission lines are heavy infrastructure that do not physically move, and only a limited set of topological actions are available to the dispatchers:

- Disconnect or reconnect transmission lines;
- Re-organize interconnection patterns at substations.

The first type can easily be modelled by a binary variable located at each power line, while the second requires more care.

Substation are places where objects such as transmission lines, transformers, generators, etc. can be connected to each other through buses. Each substation is made of two buses[1], and incoming objects can either be connected to the first or the second through a series of switches.

The left part of Figure 9.5 shows a small instance of power grid. There are two substations, each having two distinct buses. Switches located at buses can connect objects to each other.

[1]Actual substations can have more than two buses.

The right part of Figure 9.5 shows how such a system can be represented in a factorizable way. Dipoles are modelled as hyper-edges of order 2: they are linked to two distinct buses, while their actual electrical connection is encoded into a boolean variable. Similarly, quadrupoles are modelled as hyper-edges of order 4: both ends of quadrupoles can be connected to two distinct buses, and the choice between them is encoded into a boolean variable (one per end). Thus, actions over the grid topology only affect features, while the graph structure remains constant through time.

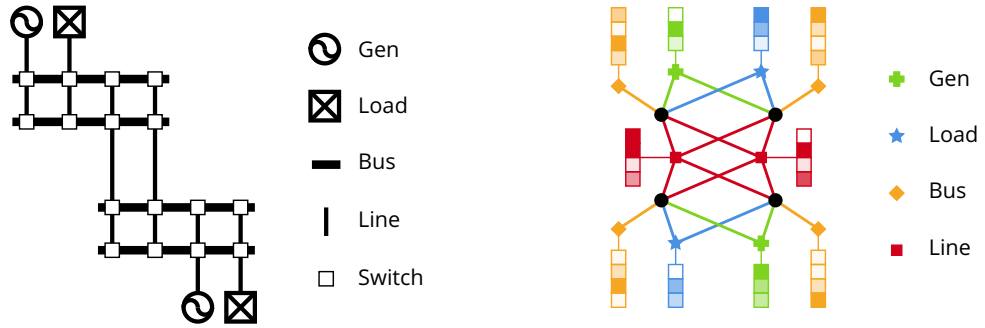


Figure 9.5: Factorizable representation of a power grid. Lines are modelled as hyper-edges of order 4. However, their actual electrical connectivity is encoded into their features. Thus topological modifications only affect binary variables located at features, and the graph structure remains constant through time.

9.3 Incorporating uncertainties

In addition to the time component, power grid operation should be robust to various forms of uncertainties. Unplanned incidents may disconnect transmission lines, production and consumption projections are not perfectly trustworthy, and sensors may perform erroneous measurements. Currently, the impact of certain classes of uncertainty is systematically estimated, while other types are neglected since they are too unlikely or too numerous. There are basically two distinct questions that one may want to address:

- How robust is a power grid situation in regard to a distribution of possible events?
- What is the best action in regard to a distribution of possible events?

The former consists in making sure that a security criterion is satisfied, while the latter consists in choosing a set of optimal actions. For the sake of simplicity, let us focus on the optimal action case.

Let us denote by $x \sim p(x)$ the initial situation, which may be a snapshot at $t-1$, a noisy prediction, or even a representation of the uncertainty (uncertainty bounds, gaussian parameters, quantiles, etc.). $y \sim q_\theta(y|x)$ is the action taken by our trainable model, based on the knowledge accessible from x. Then $z \sim r(z|x,y)$ is the actual occurrence of a random event. It depends on both x and y: the outcome of an event can be impacted by the action taken.

Depending on whether we want $q_\theta(y|x)$ to minimize an average or a maximal cost, the involved methodologies can be quite different. In power grid operation, there is currently no consensus, and both angles have their pros and cons.

9.3.1 Average cost

In the case where we are interested in average costs, we have to consider the actual distribution of random event $r(z|x,y)$, and optimize θ by considering the following SSP:

$$\Theta^* = \arg\min_{\theta\in\Theta} \mathbb{E}_{\substack{x\sim p(x)\\ y\sim q_\theta(y|x)\\ z\sim r(z|x,y)}} [\ell(x,y,z)] \tag{9.3}$$

This approach can be computationally heavy because it relies on a Monte-Carlo simulation to estimate the average cost.

9.3.2 Maximal cost

Another approach consists in minimizing the maximal possible cost for events $z \in supp(r)$. In the case where the set $supp(r)$ is countable and quite small, it may be possible to scan the full domain to find the maximal cost. However, an exhaustive search is not possible in the case of a continuous and/or large support.

A possible solution lies in the *worst case* approach, which consists in searching for the worst possible random event. Let $r^*(z|x,y)$ be the distribution that maps initial situations x and actions y to the worst possible random event. We try to imitate this distribution using a pa-

rameterized distribution $r_\omega(z|x,y)$. The related SSP is thus as follows:

$$\Theta^* = \arg\min_{\theta\in\Theta} \mathbb{E}_{\substack{x\sim p(x)\\ y\sim q_\theta(y|x)\\ z\sim r^*(z|x,y)}} [\ell(x,y,z)] \tag{9.4}$$

$$\Omega^* = \arg\min_{\theta\in\Theta} \mathbb{E}_{\substack{x\sim p(x)\\ y\sim q(y|x)\\ z\sim r_\omega(z|x,y)}} [-\ell(x,y,z)] \tag{9.5}$$

It is a typical instance of a bilevel optimization as introduced in Section 5.2, where $\ell' = -\ell$. Computations are much more focused as it is not required to explore the full support of distribution r.

9.4 Future research

Currently, several aspects of the applications of the DSS approach have not been addressed.

Choice of input data distribution The choice of the probability distribution of input data $p(x)$ is key. Unfortunately, properly defining such a distribution is in itself an extremely complex task. Moreover, associating events with probabilities and minimizing an expectation implies that rare events shall be disregarded by the learned model. In the case of critical industrial systems, we are more interested in rare but dangerous events than in common and harmless ones. Some work is required to choose a proper distribution for $p(x)$, and for assessing the validity of trained models over a series of critical test cases. However, one may object that estimating cost functions provides a surrogate of the reliability of the model.

Advanced power grid modelling Another limit of the current implementation concerns the modelling of some objects that appear in power grids. For instance, generators display a quite complex behavior. The active regulation (*i.e.* making sure that active production equates to active consumption plus Joule's effect) is not taken care by a single "slack" bus, but is rather a distributed mechanism. Generators contribute more or less depending on their respective capacities to provide additional active power. Likewise, generators that take part in the voltage control mechanism are bounded by their respective maximum and minimum reactive power. Other objects such as High Voltage Direct Current (HVDC) lines have simply been disregarded. Finally, the aforementioned voltage control problem has been drastically simplified compared to the actual issue, as dispatchers control voltage setpoints only at a predefined series of buses.

Topology control A major part of the work of dispatchers is to prevent transmission lines from overflowing. If the electrical current through a line is too high, the conductive material expands, which cause the line to get closer to the ground, thus endangering passers-by, housings, trees etc. Fortunately, automatic mechanisms are able to disconnect transmission lines that are overflowing. However, the power flow that was previously transported by the now disconnected line is pushed to neighboring lines, which may cause new overflows. This cascading failure phenomenon may end up in a complete black-out of the system. To avoid this, dispatchers can change interconnection patterns at substations by turning on or off switches (see Figure 9.5). The combinatorial aspect of this decision-making problem prevents the use of traditional optimization techniques. Employing a DSS to the problem of topology control could thus be a promising application domain.

Power grids as cyber-physical systems Real-life power grids tend to become cyber-physical systems: there is a growing number of automata on the French power grid that are able to make decisions on their own. In order to properly model the impact of their behavior over the system, it is required to simulate the whole dynamics of the electrical system. A traditional stationary simulator will not be able to properly model the dynamics involved in the automaton's decision making process. With a well designed cost function, a DSS should be able to include in its model the dynamics of automata.

Non differentiable cost functions The case where cost functions are not differentiable was completely disregarded in this work. In such a case, one could rely on gradient-free approaches typically used in Reinforcement Learning [159].

Initializing classical optimization methods A major drawback of DL-based methods is the lack of guarantee of convergence to the actual solution, and the lack of upper bound for the error. However, the solution provided by the DSS could be used as a starting point to a classical optimization method, similarly to [98].

Lagragian relaxation of the bilevel problem Regarding the bilevel SSP of Section 5.2, it would be interesting to try to simply consider Lagrangian relaxation of the bilevel optimization problem. Thus, a single neural network mapping would try to solve both the upper and the lower level problems.

Extension of the universal approximation theorem 2 The theorem presented and proved in the present document states that under hypotheses (H1-H7), there always exists a H2MGNN capable of approximating the solution of an SSP with an arbitrary precision (see Section 6.1). Still, we believe that hypothesis (H1) (no collocated objects of the same class) could be alleviated. However, this line of work is left to future research.

9.5 Long term vision

The long term objective is to develop an artificial intelligence algorithm to monitor and control real-life power grids. This goal remains distant, and one should be careful when applying DL to critical infrastructures. This PhD thesis is an attempt at properly laying the ground for a fully integrated approach in which everything would be phrased in terms of probabilities, cost functions and expectations. DL would not be simply limited to accelerating several computational bricks lost in between optimization algorithms and "for" loops.

The paradigm developed and defended in this document is to consider a series of probabilistic mappings instantiated as H2MGNNs. They would be trained using various cost functions. Among the possible tasks, we could envision the following: resolving physical equations, modelling production and consumption patterns, predicting possible trajectories for the next 24 hours, deciding which infrastructures should be put out of service for maintenance, controlling voltage setpoints, managing grid topology, etc. Those modules would involve various time scales and physical quantities. Their training would be performed in a continuous and joint manner. We would thus have a fully integrated artificial intelligence algorithm that learns on its own, and with as many abilities as it has modules.

Chapter 10

Conclusion

The Power Systems community is increasingly looking towards the use of fast and expressive deep neural networks, be it to accelerate potentially heavy computations, or to address decision making problems for which there are currently no viable methods. Prior to this PhD thesis, almost all applications of deep neural networks assumed that the graph structure of power grids was constant over time. However, transmission lines are frequently cut open for maintenance purposes, and dispatchers rearrange interconnection patterns multiple times per day, and at various locations of the grid.

Thus, our primary focus was to be able to process grids of variable topology by elaborating a suitable neural network architecture. By framing power grids as graphs, we were the first to implement a Graph Neural Network architecture applied to Power Systems [1]. This type of neural network is especially designed to handle graph data. Moreover, our recent work on real data from the French power grid has uncovered that power grids cannot be properly modelled through standard graphs, and are better described by an extension thereof, that we called Hyper Heterogeneous Multi Graphs (H2MGs). We then proposed a full-fledged optimization method, which we refer to as Deep Statistical Solvers (DSSs). As a result, the unsupervised learning of a DSS is yet another global optimization approach, an alternative to Newton-Raphson.
We experimentally validated the DSS approach and demonstrated its ability to learn on large networks (up to 1,089 vertices). Moreover, we explored its ability to transfer its knowledge to out-of-distribution samples: it generalized very well to graphs that are both larger and smaller than the graphs it was trained on, as long as the important physical quantities remained in the same range as during training. Recent experiments focused on applying this framework to actual

data from the French power grid: we obtained honorable results using the supervised "proxy" learning approach on real data, but the unsupervised learning DSS approach did not yet provide satisfactory results. Applying the Deep Statistical Solver approach to real data from the French power grid is still an ongoing line of work.

As a second application, we addressed the problem of controlling voltage setpoints, a topic that has gained popularity over the past few years due to increasingly frequent high-voltage violation issues. In order to make a good decision, dispatchers have to rely on their expertise to elaborate a tentative decision, whose expected outcome they simulate thanks to their operating tools (in particular their power flow computation software). Thus dispatchers want to optimize a security objective, while anticipating the outcome of their decisions over the system, which shall be computed by a potentially complex simulator. Drawing inspiration from Generative Adversarial Networks, we proposed to jointly train two distinct Hyper Heterogeneous Multi Graph Neural Networks, one playing the role of the dispatcher (controller), and the other playing the role of the simulator (solver). Early experiments on a small 14 buses power grid showed promising results.

Further work include scaling up the full unsupervised Deep Statistical Solver approach to real data, improving the choice of power grid distributions used during training and including time and uncertainties in the framework.

Bibliography

[1] Balthazar Donon, Benjamin Donnot, Isabelle Guyon, and Antoine Marot. "Graph Neural Solver for Power Systems". In: *IJCNN 2019 - International Joint Conference on Neural Networks*. Budapest, Hungary, July 2019. url: https://hal.archives-ouvertes.fr/hal-02175989.

[2] Balthazar Donon, Rémy Clément, Benjamin Donnot, Antoine Marot, Isabelle Guyon, and Marc Schoenauer. "Neural networks for power flow: Graph neural solver". In: *Electric Power Systems Research* 189 (2020), p. 106547. issn: 0378-7796. doi: https://doi.org/10.1016/j.epsr.2020.106547.

[3] Balthazar Donon, Wenzhuo Liu, Antoine Marot, Zhengying Liu, Isabelle Guyon, and Marc Schoenauer. "Deep Statistical Solvers". In: *NeurIPS 2020 - 34th Conference on Neural Information Processing Systems*. Vancouver / Virtuel, Canada, Dec. 2020. url: https://hal.inria.fr/hal-02974541.

[4] John Cook, Dana Nuccitelli, Sarah A Green, Mark Richardson, Bärbel Winkler, Rob Painting, Robert Way, Peter Jacobs, and Andrew Skuce. "Quantifying the consensus on anthropogenic global warming in the scientific literature". en. In: *Environmental Research Letters* 8.2 (June 2013), p. 024024. issn: 1748-9326. doi: 10.1088/1748-9326/8/2/024024. url: https://iopscience.iop.org/article/10.1088/1748-9326/8/2/024024 (visited on 04/09/2021).

[5] Svante Arrhenius. "On the Influence of Carbonic Acid in the Air upon the Temperature of the Ground". In: *Philosophical Magazine and Journal of Science* 41.5 (Apr. 1896), pp. 237–276.

[6] G. S. Callendar. "The artificial production of carbon dioxide and its influence on temperature". In: *Quarterly Journal of the Royal Meteorological Society* 64.275 (1938). _eprint: https://rmets.onlinelibrary.wiley.com/doi/pdf/10.1002/qj.49706427503, pp. 223–240. doi: https://doi.org/10.1002/qj.49706427503.

url: https://rmets.onlinelibrary.wiley.com/doi/abs/10.1002/qj.49706427503.

[7] Gilbert N. Plass. "The Carbon Dioxide Theory of Climatic Change". en. In: *Tellus* 8.2 (May 1956), pp. 140–154. issn: 00402826, 21533490. doi: 10.1111/j.2153-3490.1956.tb01206.x. url: http://tellusa.net/index.php/tellusa/article/view/8969 (visited on 04/14/2021).

[8] D. Connolly, H. Lund, and B.V. Mathiesen. "Smart Energy Europe: The technical and economic impact of one potential 100% renewable energy scenario for the European Union". en. In: *Renewable and Sustainable Energy Reviews* 60 (July 2016), pp. 1634–1653. issn: 13640321. doi: 10.1016/j.rser.2016.02.025. url: https://linkinghub.elsevier.com/retrieve/pii/S1364032116002331 (visited on 04/18/2021).

[9] *Vers un mix électrique 100% renouvelable en 2050*. Tech. rep. Agence de l'Environnement et de la Maîtrise de l'Energie, 2020.

[10] Guillaume Denis, Thibault Prevost, Marie-Sophie Debry, Florent Xavier, Xavier Guillaud, and Andreas Menze. "The Migrate project: the challenges of operating a transmission grid with only inverter-based generation. A grid-forming control improvement with transient current-limiting control". English. In: *IET Renewable Power Generation* 12.5 (Apr. 2018). Publisher: Institution of Engineering and Technology, 523–529(6). issn: 1752-1416. url: https://digital-library.theiet.org/content/journals/10.1049/iet-rpg.2017.0369.

[11] RTE. *Integration of electric vehicles into the power system in France*. Tech. rep. RTE, May 2019. url: https://assets.rte-france.com/prod/public/2020-06/Rte%20electromobility%20report_-_eng.pdf.

[12] *The Garpur Project Results*. Tech. rep. Oct. 2017. url: https://www.sintef.no/projectweb/garpur/.

[13] D. Poole, A. Mackworth, and R. Goebel. *Computational Intelligence: A Logical Approach*. Oxford University Press, 1998.

[14] Tom M. Mitchell. *Machine Learning*. New York: McGraw-Hill, 1997. isbn: 978-0-07-042807-2.

[15] S. B. Kotsiantis. "Decision trees: a recent overview". In: *Artificial Intelligence Review* 39.4 (2013), pp. 261–283. issn: 1573-7462. doi: 10.1007/s10462-011-9272-4. url: https://doi.org/10.1007/s10462-011-9272-4.

[16] Kashvi Taunk, Sanjukta De, Srishti Verma, and Aleena Swetapadma. “A Brief Review of Nearest Neighbor Algorithm for Learning and Classification”. In: *2019 International Conference on Intelligent Computing and Control Systems (ICCS)*. 2019, pp. 1255–1260. doi: 10.1109/ICCS45141.2019.9065747.

[17] Chun Yu and Weixin Yao. “Robust linear regression: A review and comparison”. In: *Communications in Statistics-Simulation and Computation* 46.8 (2017). Publisher: Taylor & Francis, pp. 6261–6282.

[18] Gurneet Kaur and Er Neelam Oberai. “A review article on Naive Bayes classifier with various smoothing techniques”. In: *International Journal of Computer Science and Mobile Computing* 3.10 (2014), pp. 864–868.

[19] Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. “A Training Algorithm for Optimal Margin Classifiers”. In: *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*. COLT ’92. event-place: Pittsburgh, Pennsylvania, USA. New York, NY, USA: Association for Computing Machinery, 1992, pp. 144–152. isbn: 0-89791-497-X. doi: 10.1145/130385.130401. url: https://doi.org/10.1145/130385.130401.

[20] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[21] Chaochao Lu and Xiaoou Tang. “Surpassing Human-Level Face Verification Performance on LFW with GaussianFace”. In: *CoRR* abs/1404.3840 (2014). _eprint: 1404.3840. url: http://arxiv.org/abs/1404.3840.

[22] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. “Going Deeper with Convolutions”. In: *CoRR* abs/1409.4842 (2014). _eprint: 1409.4842. url: http://arxiv.org/abs/1409.4842.

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. “Deep Residual Learning for Image Recognition”. In: *CoRR* abs/1512.03385 (2015). _eprint: 1512.03385. url: http://arxiv.org/abs/1512.03385.

[24] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. *Generative Adversarial Networks*. _eprint: 1406.2661. 2014.

[25] Andrej Karpathy and Li Fei-Fei. “Deep Visual-Semantic Alignments for Generating Image Descriptions”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2015.

[26] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Ng, and Christopher Potts. “Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank”. In: *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*. Seattle, Washington, USA: Association for Computational Linguistics, Oct. 2013, pp. 1631–1642. url: https://aclanthology.org/D13-1170.

[27] Yelong Shen, Xiaodong He, Jianfeng Gao, Li Deng, and Gregoire Mesnil. “A Latent Semantic Model with Convolutional-Pooling Structure for Information Retrieval”. In: *CIKM*. Nov. 2014. url: https://www.microsoft.com/en-us/research/publication/a-latent-semantic-model-with-convolutional-pooling-structure-for-information-retrieval/.

[28] Grégoire Mesnil, Yann Dauphin, Kaisheng Yao, Yoshua Bengio, Li Deng, Dilek Hakkani-Tur, Xiaodong He, Larry Heck, Gokhan Tur, Dong Yu, and Geoffrey Zweig. “Using Recurrent Neural Networks for Slot Filling in Spoken Language Understanding”. In: *IEEE/ACM Trans. Audio, Speech and Lang. Proc.* 23.3 (Mar. 2015). Publisher: IEEE Press, pp. 530–539. issn: 2329-9290.

[29] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. “Sequence to Sequence Learning with Neural Networks”. In: *CoRR* abs/1409.3215 (2014). _eprint: 1409.3215. url: http://arxiv.org/abs/1409.3215.

[30] Marcelo Brocardo, Issa Traore, Isaac Woungang, and Mohammad Obaidat. “Authorship verification using deep belief network systems”. In: *International Journal of Communication Systems* 30 (2017), e3259. doi: 10.1002/dac.3259.

[31] Awni Y. Hannun, Carl Case, J. Casper, Bryan Catanzaro, G. Diamos, Erich Elsen, R. Prenger, S. Satheesh, Shubho Sengupta, A. Coates, and A. Ng. “Deep Speech: Scaling up end-to-end speech recognition”. In: *ArXiv* abs/1412.5567 (2014).

[32] Louis WEHENKEL. *Machine learning approaches to power system security assessment /*. Liège : Université de Liège, Faculté des sciences appliquées (ULg), Oct. 1994.

[33] Laurine Duchesne, Efthymios Karangelos, and Louis Wehenkel. "Recent Developments in Machine Learning for Energy Systems Reliability Management". en. In: *Proceedings of the IEEE* 108.9 (Sept. 2020), pp. 1656–1676. issn: 0018-9219, 1558-2256. doi: 10.1109/JPROC.2020.2988715. url: https://ieeexplore.ieee.org/document/9091534/ (visited on 04/07/2021).

[34] Antoine Marot, Benjamin Donnot, Camilo Romero, Balthazar Donon, Marvin Lerousseau, Luca Veyrin-Forrer, and Isabelle Guyon. "Learning to run a power network challenge for training topology controllers". In: *Electric Power Systems Research* 189 (2020), p. 106635. issn: 0378-7796. doi: https://doi.org/10.1016/j.epsr.2020.106635. url: https://www.sciencedirect.com/science/article/pii/S0378779620304387.

[35] Florian Schaefer, Jan-Hendrik Menke, and Martin Braun. *Evaluating Machine Learning Models for the Fast Identification of Contingency Cases*. _eprint: 2008.09384. 2020.

[36] Benjamin Donnot, Isabelle Guyon, Marc Schoenauer, Patrick Panciatici, and Antoine Marot. *Introducing machine learning for power system operation support*. _eprint: 1709.09527. 2017.

[37] A.P.S. Meliopoulos, G.J. Cokkinides, and X.Y. Chao. "A new probabilistic power flow analysis method". In: *IEEE Transactions on Power Systems* 5.1 (1990), pp. 182–190. doi: 10.1109/59.49104.

[38] T. T. Nguyen. "Neural network load-flow". English. In: *IEE Proceedings - Generation, Transmission and Distribution* 142.1 (Jan. 1995), 51–58(7). issn: 1350-2360. url: https://digital-library.theiet.org/content/journals/10.1049/ip-gtd_19951484.

[39] Yan Du, Fangxing Li, Jiang Li, and Tongxin Zheng. "Achieving 100x Acceleration for N-1 Contingency Screening With Uncertain Scenarios Using Deep Convolutional Neural Network". In: *IEEE Transactions on Power Systems* 34.4 (2019), pp. 3303–3305. doi: 10.1109/TPWRS.2019.2914860.

[40] Liangjie Chen and Joseph Euzebe Tate. *Hot-Starting the Ac Power Flow with Convolutional Neural Networks*. _eprint: 2004.09342. 2020.

[41] M. Krishna Paramathma, D. Devaraj, and B Subba Reddy. "Artificial neural network based static security assessment module using PMU measurements for smart grid application". In: *2016 International Conference on Emerging Trends in Engineering, Technology and Science (ICETETS)*. 2016, pp. 1–5. doi: 10.1109/ICETETS.2016.7603086.

[42] Ishita Bhatt, Astik Dhandhia, and Vivek Pandya. “Static Security Assessment of Power System Using Radial Basis Function Neural Network Module”. In: *2017 IEEE International WIE Conference on Electrical and Computer Engineering (WIECON-ECE)*. 2017, pp. 274–278. doi: 10.1109/WIECON-ECE.2017.8468879.

[43] Ahmed Zamzam and Kyri Baker. *Learning Optimal Solutions for Extremely Fast AC Optimal Power Flow*. _eprint: 1910.01213. 2019.

[44] Minas Chatzos, Ferdinando Fioretto, Terrence W. K. Mak, and Pascal Van Hentenryck. *High-Fidelity Machine Learning Approximations of Large-Scale Optimal Power Flow*. _eprint: 2006.16356. 2020.

[45] Xiang Pan, Tianyu Zhao, and Minghua Chen. “DeepOPF: Deep Neural Network for DC Optimal Power Flow”. In: *2019 IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids (SmartGridComm)*. 2019, pp. 1–6. doi: 10.1109/SmartGridComm.2019.8909795.

[46] David Biagioni, Peter Graf, Xiangyu Zhang, Ahmed Zamzam, Kyri Baker, and Jennifer King. *Learning-Accelerated ADMM for Distributed Optimal Power Flow*. _eprint: 1911.03019. 2020.

[47] Wenqian Dong, Zhen Xie, Gokcen Kestor, and Dong Li. *Smart-PGSim: Using Neural Network to Accelerate AC-OPF Power Grid Simulation*. _eprint: 2008.11827. 2020.

[48] Kyri Baker. *Learning Warm-Start Points for AC Optimal Power Flow*. _eprint: 1905.08860. 2019.

[49] Raphael Canyasse, Gal Dalal, and Shie Mannor. *Supervised Learning for Optimal Power Flow as a Real-Time Proxy*. _eprint: 1612.06623. 2016.

[50] Laurine Duchesne, Efthymios Karangelos, and Louis Wehenkel. “Machine learning of real-time power systems reliability management response”. In: *2017 IEEE Manchester PowerTech*. 2017, pp. 1–6. doi: 10.1109/PTC.2017.7980927.

[51] Laurine Duchesne, Efthymios Karangelos, Antonio Sutera, and Louis Wehenkel. “Machine learning for ranking day-ahead decisions in the context of short-term operation planning”. In: *Electric Power Systems Research* 189 (2020), p. 106548. issn: 0378-7796. doi: https://doi.org/10.1016/j.epsr.2020.106548. url: https://www.sciencedirect.com/science/article/pii/S0378779620303527.

[52] Laurine Duchesne, Efthymios Karangelos, and Louis Wehenkel. "Using Machine Learning to Enable Probabilistic Reliability Assessment in Operation Planning". In: *2018 Power Systems Computation Conference (PSCC)*. 2018, pp. 1–8. doi: 10.23919/PSCC.2018.8442566.

[53] Yeesian Ng, Sidhant Misra, Line A. Roald, and Scott Backhaus. *Statistical Learning For DC Optimal Power Flow*. _eprint: 1801.07809. 2018.

[54] Sidhant Misra, Line Roald, and Yeesian Ng. *Learning for Constrained Optimization: Identifying Optimal Active Constraint Sets*. _eprint: 1802.09639. 2019.

[55] Thomas Falconer and Letif Mones. *Deep learning architectures for inference of AC-OPF solutions*. _eprint: 2011.03352. 2020.

[56] A. Robson, Mahdi Jamei, C. Ududec, and L. Mones. "Learning an Optimally Reduced Formulation of OPF through Meta-optimization". In: *ArXiv* abs/1911.06784 (2019).

[57] Benjamin Donnot, Balthazar Donon, Isabelle Guyon, Zhengying Liu, Antoine Marot, Patrick Panciatici, and Marc Schoenauer. *LEAP nets for power grid perturbations*. _eprint: 1908.08314. 2019.

[58] Peter W. Battaglia, Jessica B. Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Caglar Gulcehre, Francis Song, Andrew Ballard, Justin Gilmer, George Dahl, Ashish Vaswani, Kelsey Allen, Charles Nash, Victoria Langston, Chris Dyer, Nicolas Heess, Daan Wierstra, Pushmeet Kohli, Matt Botvinick, Oriol Vinyals, Yujia Li, and Razvan Pascanu. "Relational inductive biases, deep learning, and graph networks". en. In: *arXiv:1806.01261 [cs, stat]* (Oct. 2018). arXiv: 1806.01261. url: http://arxiv.org/abs/1806.01261 (visited on 04/09/2021).

[59] A. Sperduti and A. Starita. "Supervised neural networks for the classification of structures". en. In: *IEEE Transactions on Neural Networks* 8.3 (May 1997), pp. 714–735. issn: 1045-9227, 1941-0093. doi: 10.1109/72.572108. url: https://ieeexplore.ieee.org/document/572108/ (visited on 04/14/2021).

[60] M. Gori, G. Monfardini, and F. Scarselli. "A new model for learning in graph domains". en. In: *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.* Vol. 2. Montreal, Que., Canada: IEEE, 2005, pp. 729–734. isbn: 978-0-7803-9048-5. doi:

10.1109/IJCNN.2005.1555942. url: http://ieeexplore.ieee.org/document/1555942/ (visited on 04/14/2021).

[61] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. "The Graph Neural Network Model". en. In: *IEEE TRANSACTIONS ON NEURAL NETWORKS* 20.1 (2009), p. 20.

[62] Claudio Gallicchio. "Graph Echo State Networks". en. In: (), p. 8.

[63] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. *Spectral Networks and Locally Connected Networks on Graphs*. _eprint: 1312.6203. 2014.

[64] Mikael Henaff, Joan Bruna, and Yann LeCun. *Deep Convolutional Networks on Graph-Structured Data*. _eprint: 1506.05163. 2015.

[65] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. *Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering*. _eprint: 1606.09375. 2017.

[66] Thomas N. Kipf and Max Welling. *Semi-Supervised Classification with Graph Convolutional Networks*. _eprint: 1609.02907. 2017.

[67] Ron Levie, Federico Monti, Xavier Bresson, and Michael M. Bronstein. *CayleyNets: Graph Convolutional Neural Networks with Complex Rational Spectral Filters*. _eprint: 1705.07664. 2018.

[68] Alessio Micheli. "Neural Network for Graphs: A Contextual Constructive Approach". In: *IEEE Transactions on Neural Networks* 20.3 (2009), pp. 498–511. doi: 10.1109/TNN.2008.2010350.

[69] James Atwood and Don Towsley. *Diffusion-Convolutional Neural Networks*. _eprint: 1511.02136. 2016.

[70] Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. *Learning Convolutional Neural Networks for Graphs*. _eprint: 1605.05273. 2016.

[71] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. *Neural Message Passing for Quantum Chemistry*. _eprint: 1704.01212. 2017.

[72] Danfei Xu, Yuke Zhu, Christopher B. Choy, and Li Fei-Fei. *Scene Graph Generation by Iterative Message Passing*. _eprint: 1701.02426. 2017.

[73] Jianwei Yang, Jiasen Lu, Stefan Lee, Dhruv Batra, and Devi Parikh. *Graph R-CNN for Scene Graph Generation*. _eprint: 1808.00191. 2018.

[74] Yikang Li, Wanli Ouyang, Bolei Zhou, Jianping Shi, Chao Zhang, and Xiaogang Wang. *Factorizable Net: An Efficient Subgraph-based Framework for Scene Graph Generation*. _eprint: 1806.11538. 2018.

[75] Justin Johnson, Agrim Gupta, and Li Fei-Fei. *Image Generation from Scene Graphs*. _eprint: 1804.01622. 2018.

[76] Ashesh Jain, Amir R. Zamir, Silvio Savarese, and Ashutosh Saxena. *Structural-RNN: Deep Learning on Spatio-Temporal Graphs*. _eprint: 1511.05298. 2016.

[77] Sijie Yan, Yuanjun Xiong, and Dahua Lin. *Spatial Temporal Graph Convolutional Networks for Skeleton-Based Action Recognition*. _eprint: 1801.07455. 2018.

[78] Victor Garcia and Joan Bruna. *Few-Shot Learning with Graph Neural Networks*. _eprint: 1711.04043. 2018.

[79] Michelle Guo, Edward Chou, De-An Huang, Shuran Song, Serena Yeung, and Li Fei-Fei. "Neural Graph Matching Networks for Fewshot 3D Action Recognition". In: *ECCV*. 2018.

[80] Xinlei Chen, Li-Jia Li, Li Fei-Fei, and Abhinav Gupta. *Iterative Visual Reasoning Beyond Convolutions*. _eprint: 1803.11189. 2018.

[81] David Duvenaud, Dougal Maclaurin, Jorge Aguilera-Iparraguirre, Rafael Gómez-Bombarelli, Timothy Hirzel, Alán Aspuru-Guzik, and Ryan P. Adams. *Convolutional Networks on Graphs for Learning Molecular Fingerprints*. _eprint: 1509.09292. 2015.

[82] Steven Kearnes, Kevin McCloskey, Marc Berndl, Vijay Pande, and Patrick Riley. "Molecular graph convolutions: moving beyond fingerprints". In: *Journal of Computer-Aided Molecular Design* 30.8 (Aug. 2016). Publisher: Springer Science and Business Media LLC, pp. 595–608. issn: 1573-4951. doi: 10.1007/s10822-016-9938-8. url: http://dx.doi.org/10.1007/s10822-016-9938-8.

[83] Alex Fout, Jonathon Byrd, Basir Shariat, and Asa Ben-Hur. "Protein Interface Prediction Using Graph Convolutional Networks". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS'17. event-place: Long Beach, California, USA. Red Hook, NY, USA: Curran Associates Inc., 2017, pp. 6533–6542. isbn: 978-1-5108-6096-4.

[84] Yujia Li, Oriol Vinyals, Chris Dyer, Razvan Pascanu, and Peter Battaglia. *Learning Deep Generative Models of Graphs*. _eprint: 1803.03324. 2018.

[85] Nicola De Cao and Thomas Kipf. *MolGAN: An implicit generative model for small molecular graphs*. _eprint: 1805.11973. 2018.

[86] Jiaxuan You, Bowen Liu, Rex Ying, Vijay Pande, and Jure Leskovec. *Graph Convolutional Policy Network for Goal-Directed Molecular Graph Generation*. _eprint: 1806.02473. 2019.

[87] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. "A Comprehensive Survey on Graph Neural Networks". en. In: *IEEE Transactions on Neural Networks and Learning Systems* 32.1 (Jan. 2021). arXiv: 1901.00596, pp. 4–24. issn: 2162-237X, 2162-2388. doi: 10.1109/TNNLS.2020.2978386. url: http://arxiv.org/abs/1901.00596 (visited on 04/09/2021).

[88] William L. Hamilton. "Graph Representation Learning". In: *Synthesis Lectures on Artificial Intelligence and Machine Learning* 14.3 (). Publisher: Morgan and Claypool, pp. 1–159.

[89] Jonathan Tompson, Kristofer Schlachter, Pablo Sprechmann, and Ken Perlin. *Accelerating Eulerian Fluid Simulation With Convolutional Networks*. _eprint: 1607.03597. 2017.

[90] M. F. Kasim, D. Watson-Parris, L. Deaconu, S. Oliver, P. Hatfield, D. H. Froula, G. Gregori, M. Jarvis, S. Khatiwala, J. Korenaga, J. Topp-Mugglestone, E. Viezzer, and S. M. Vinko. *Building high accuracy emulators for scientific simulations with deep neural architecture search*. _eprint: 2001.08055. 2020.

[91] Stephan Rasp, Michael S. Pritchard, and Pierre Gentine. "Deep learning to represent subgrid processes in climate models". In: *Proceedings of the National Academy of Sciences* 115.39 (2018). Publisher: National Academy of Sciences _eprint: https://www.pnas.org/content/115/39/9684.full.pdf, pp. 9684–9689. issn: 0027-8424. doi: 10.1073/pnas.1810286115. url: https://www.pnas.org/content/115/39/9684.

[92] Kim Albertsson et al. *Machine Learning in High Energy Physics Community White Paper*. _eprint: 1807.02876. 2019.

[93] Thomas Kipf, Ethan Fetaya, Kuan-Chieh Wang, Max Welling, and Richard Zemel. "Neural Relational Inference for Interacting Systems". In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, July 2018, pp. 2688–2697. url: http://proceedings.mlr.press/v80/kipf18a.html.

[94] Xiangyang Ju, Steven Farrell, Paolo Calafiura, Daniel Murnane, Prabhat, Lindsey Gray, Thomas Klijnsma, Kevin Pedro, Giuseppe Cerati, Jim Kowalkowski, Gabriel Perdue, Panagiotis Spentzouris, Nhan Tran, Jean-Roch Vlimant, Alexander Zlokapa, Joosep Pata, Maria Spiropulu, Sitong An, Adam Aurisano, Jeremy Hewes, Aristeidis Tsaris, Kazuhiro Terao, and Tracy Usher. *Graph Neural Networks for Particle Reconstruction in High Energy Physics detectors*. _eprint: 2003.11603. 2020.

[95] M. Raissi, P. Perdikaris, and G. E. Karniadakis. "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations". In: *Journal of Computational Physics* 378 (2019), pp. 686–707. issn: 0021-9991. doi: https://doi.org/10.1016/j.jcp.2018.10.045. url: https://www.sciencedirect.com/science/article/pii/S0021999118307125.

[96] Shengze Cai, Zhiping Mao, Zhicheng Wang, Minglang Yin, and George Em Karniadakis. *Physics-informed neural networks (PINNs) for fluid mechanics: A review*. _eprint: 2105.09506. 2021.

[97] Damian Owerko, Fernando Gama, and Alejandro Ribeiro. "Optimal Power Flow Using Graph Neural Networks". In: *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2020, pp. 5930–5934. doi: 10.1109/ICASSP40776.2020.9053140.

[98] Frederik Diehl. "Warm-Starting AC Optimal Power Flow with Graph Neural Networks". In: 2019.

[99] Wenlong Liao, Birgitte Bak-Jensen, Jayakrishnan Radhakrishna Pillai, Yuelong Wang, and Yusen Wang. *A Review of Graph Neural Networks and Their Applications in Power Systems*. _eprint: 2101.10025. 2021.

[100] Alvaro Sanchez-Gonzalez, Jonathan Godwin, Tobias Pfaff, Rex Ying, Jure Leskovec, and Peter Battaglia. "Learning to Simulate Complex Physics with Graph Networks". In: *Proceedings of the 37th International Conference on Machine Learning*. Ed. by Hal Daumé III and Aarti Singh. Vol. 119. Proceedings of Machine Learning Research. PMLR, July 2020, pp. 8459–8468. url: http://proceedings.mlr.press/v119/sanchez-gonzalez20a.html.

[101] Laurent Pagnier and Michael Chertkov. *Physics-Informed Graphical Neural Network for Parameter & State Estimations in Power Systems*. _eprint: 2102.06349. 2021.

[102] Wenting Li and Deepjyoti Deka. *Physics-Informed Graph Learning for Robust Fault Location in Distribution Systems*. _eprint: 2107.02275. 2021.

[103] Prabha Kundur. *Power System Stability and Control*. McGraw-Hill, 1994.

[104] Hugh G. J. Aitken and Thomas P. Hughes. "Networks of Power: Electrification in Western Society". In: *Journal of Interdisciplinary History* 15 (1984), p. 350.

[105] Massimo Guarnieri. "The Beginning of Electric Energy Transmission: Part One [Historical]". In: *IEEE Industrial Electronics Magazine* 7 (2013), pp. 50–52.

[106] Philippe JEANNIN and Jacques CARPENTIER. "Réseaux de puissance Méthodes de résolution des équations". In: *Techniques de l'ingénieur Conversion de l'énergie électrique* base documentaire : TIP301WEB.ref. article : d1120 (1994). Publisher: Editions T.I. Type: base documentaire _eprint: base documentaire : TIP301WEB. doi: 10.51257/a-v1-d1120. url: https://www.techniques-ingenieur.fr/base-documentaire/energies-th4/reseaux-electriques-lineaires-42258210/reseaux-de-puissance-d1120/.

[107] Vladimir N. Vapnik. *Statistical Learning Theory*. Wiley-Interscience, 1998.

[108] Xue Ying. "An Overview of Overfitting and its Solutions". In: *Journal of Physics: Conference Series* 1168 (Feb. 2019). Publisher: IOP Publishing, p. 022022. doi: 10.1088/1742-6596/1168/2/022022. url: https://doi.org/10.1088/1742-6596/1168/2/022022.

[109] G. Cybenko. "Approximation by superpositions of a sigmoidal function". In: *Mathematics of Control, Signals and Systems* 2.4 (1989), pp. 303–314. issn: 1435-568X. doi: 10.1007/BF02551274. url: https://doi.org/10.1007/BF02551274.

[110] K. Hornik, M. Stinchcombe, and H. White. "Multilayer Feedforward Networks Are Universal Approximators". In: *Neural Netw.* 2.5 (1989). Place: GBR Publisher: Elsevier Science Ltd., pp. 359–366. issn: 0893-6080.

[111] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323.6088 (Oct. 1986), pp. 533–536. issn: 1476-4687. doi: 10.1038/323533a0. url: https://doi.org/10.1038/323533a0.

[112] L. Bottou. “On-line learning and stochastic approximations”. In: 1999.

[113] B. T. Polyak. “Some methods of speeding up the convergence of iteration methods”. In: *USSR Computational Mathematics and Mathematical Physics* 4.5 (1964), pp. 1–17. issn: 0041-5553. doi: https://doi.org/10.1016/0041-5553(64)90137-5. url: https://www.sciencedirect.com/science/article/pii/0041555364901375.

[114] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. “On the importance of initialization and momentum in deep learning”. In: *Proceedings of the 30th International Conference on Machine Learning*. Ed. by Sanjoy Dasgupta and David McAllester. Vol. 28. Proceedings of Machine Learning Research. Issue: 3. Atlanta, Georgia, USA: PMLR, June 2013, pp. 1139–1147. url: https://proceedings.mlr.press/v28/sutskever13.html.

[115] John Duchi, Elad Hazan, and Yoram Singer. “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization”. en. In: (), p. 39.

[116] Geoffrey Hinton. *Neural Networks for Machine Learning*. url: http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[117] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. url: http://arxiv.org/abs/1412.6980.

[118] Timothy Dozat. “INCORPORATING NESTEROV MOMENTUM INTO ADAM”. en. In: (2016), p. 4.

[119] Xavier Glorot and Yoshua Bengio. “Understanding the difficulty of training deep feedforward neural networks”. en. In: (), p. 8.

[120] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*. _eprint: 1502.01852. 2015.

[121] C. A. R. de Sousa. “An overview on weight initialization methods for feedforward neural networks”. In: *2016 International Joint Conference on Neural Networks (IJCNN)*. 2016, pp. 52–59. doi: 10.1109/IJCNN.2016.7727180.

[122] James Bergstra and Yoshua Bengio. “Random Search for Hyper-Parameter Optimization.” In: *J. Mach. Learn. Res.* 13 (2012), pp. 281–305. url: http://dblp.uni-trier.de/db/journals/jmlr/jmlr13.html#BergstraB12.

[123] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. “Practical Bayesian Optimization of Machine Learning Algorithms”. In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger. Vol. 25. Curran Associates, Inc., 2012. url: https://proceedings.neurips.cc/paper/2012/file/05311655a15b75fab86956663e1819cd-Paper.pdf.

[124] Mauro Birattari, Thomas Stützle, Luis Paquete, and Klaus Varrentrapp. “A Racing Algorithm for Configuring Metaheuristics”. In: *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*. GECCO’02. event-place: New York City, New York. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002, pp. 11–18. isbn: 1-55860-878-8.

[125] Yann Lecun. “Generalization and network design strategies”. English (US). In: *Connectionism in perspective*. Ed. by R. Pfeifer, Z. Schreter, F. Fogelman, and L. Steels. Elsevier, 1989.

[126] Stéphane Mallat. “Understanding deep convolutional networks”. In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 374.2065 (Apr. 2016). Publisher: The Royal Society, p. 20150203. issn: 1471-2962. doi: 10.1098/rsta.2015.0203. url: http://dx.doi.org/10.1098/rsta.2015.0203.

[127] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. “ImageNet classification with deep convolutional neural networks”. en. In: *Communications of the ACM* 60.6 (May 2017), pp. 84–90. issn: 0001-0782, 1557-7317. doi: 10.1145/3065386. url: https://dl.acm.org/doi/10.1145/3065386 (visited on 04/18/2021).

[128] Moez Baccouche, Franck Mamalet, Christian Wolf, Christophe Garcia, and Atilla Baskurt. *A.: Sequential deep learning for human action recognition*. 2011.

[129] S. Ji, W. Xu, M. Yang, and K. Yu. “3D Convolutional Neural Networks for Human Action Recognition”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35.1 (2013), pp. 221–231. doi: 10.1109/TPAMI.2012.59.

[130] Edward Grefenstette, Phil Blunsom, Nando de Freitas, and Karl Moritz Hermann. *A Deep Architecture for Semantic Parsing*. _eprint: 1404.7296. 2014.

[131] Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. “A Convolutional Neural Network for Modelling Sentences”. en. In: *arXiv:1404.2188 [cs]* (Apr. 2014). arXiv: 1404.2188. url: http://arxiv.org/abs/1404.2188 (visited on 04/18/2021).

[132] Hansheng Ren, Bixiong Xu, Yujing Wang, Chao Yi, Congrui Huang, Xiaoyu Kou, Tony Xing, Mao Yang, Jie Tong, and Qi Zhang. “Time-Series Anomaly Detection Service at Microsoft”. en. In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (July 2019). arXiv: 1906.03821, pp. 3009–3017. doi: 10.1145/3292500.3330680. url: http://arxiv.org/abs/1906.03821 (visited on 04/18/2021).

[133] Izhar Wallach, Michael Dzamba, and Abraham Heifets. “AtomNet: A Deep Convolutional Neural Network for Bioactivity Prediction in Structure-based Drug Discovery”. In: *CoRR* abs/1510.02855 (2015). _eprint: 1510.02855. url: http://arxiv.org/abs/1510.02855.

[134] David Silver, Aja Huang, Christopher Maddison, Arthur Guez, Laurent Sifre, George Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. “Mastering the game of Go with deep neural networks and tree search”. In: *Nature* 529 (2016), pp. 484–489. doi: 10.1038/nature16961.

[135] Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. “An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling”. en. In: *arXiv:1803.01271 [cs]* (Apr. 2018). arXiv: 1803.01271. url: http://arxiv.org/abs/1803.01271 (visited on 04/18/2021).

[136] A. Tsantekidis, N. Passalis, A. Tefas, J. Kanniainen, M. Gabbouj, and A. Iosifidis. “Forecasting Stock Prices from the Limit Order Book Using Convolutional Neural Networks”. In: *2017 IEEE 19th Conference on Business Informatics (CBI)*. Vol. 01. 2017, pp. 7–12. doi: 10.1109/CBI.2017.23.

[137] M. E. J. Newman. *Networks: an introduction*. Oxford; New York: Oxford University Press, 2010. isbn: 978-0-19-920665-0 0-19-920665-1. url: http://www.amazon.com/Networks-An-Introduction-Mark-Newman/dp/0199206651/ref=sr_1_5?ie=UTF8&qid=1352896678&sr=8-5&keywords=complex+networks.

[138] Yifan Feng, Haoxuan You, Zizhao Zhang, Rongrong Ji, and Yue Gao. *Hypergraph Neural Networks*. _eprint: 1809.09401. 2019.

[139] Chuxu Zhang, Dongjin Song, Chao Huang, Ananthram Swami, and Nitesh V. Chawla. "Heterogeneous Graph Neural Network". In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. KDD '19. event-place: Anchorage, AK, USA. New York, NY, USA: Association for Computing Machinery, 2019, pp. 793–803. isbn: 978-1-4503-6201-6. doi: 10.1145/3292500.3330961. url: https://doi.org/10.1145/3292500.3330961.

[140] Tian Qi Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. "Neural Ordinary Differential Equations". In: *CoRR* abs/1806.07366 (2018). arXiv: 1806.07366. url: http://arxiv.org/abs/1806.07366.

[141] R Stoer J.\ and Bulirsch. *Introduction to numerical analysis*. Texts in applied mathematics. Springer, 2002. isbn: 978-0-387-95452-3.

[142] Grégoire Allaire. "A review of adjoint methods for sensitivity analysis, uncertainty quantification and optimization in numerical codes". In: *Ingénieurs de l'Automobile* 836 (July 2015). Publisher: SIA, pp. 33–36. url: https://hal.archives-ouvertes.fr/hal-01242950.

[143] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. _eprint: 1502.03167. 2015.

[144] Chii-Ruey Hwang. "Laplace's Method Revisited: Weak Convergence of Probability Measures". In: *The Annals of Probability* 8.6 (1980). Publisher: Institute of Mathematical Statistics, pp. 1177–1182. doi: 10.1214/aop/1176994579. url: https://doi.org/10.1214/aop/1176994579.

[145] Jerome Bracken and James T. McGill. "Mathematical Programs with Optimization Problems in the Constraints". In: *Oper. Res.* 21.1 (Feb. 1973). Place: Linthicum, MD, USA Publisher: INFORMS, pp. 37–44. issn: 0030-364X. doi: 10.1287/opre.21.1.37. url: https://doi.org/10.1287/opre.21.1.37.

[146] Ricard Durall, Avraam Chatzimichailidis, Peter Labus, and Janis Keuper. *Combating Mode Collapse in GAN training: An Empirical Analysis using Hessian Eigenvalues*. _eprint: 2012.09673. 2020.

[147] B. L. Douglas. *The Weisfeiler-Lehman Method and Graph Isomorphism Testing*. _eprint: 1101.5211. 2011.

[148] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. *How Powerful are Graph Neural Networks?* _eprint: 1810.00826. 2019.

[149] Nicolas Keriven and Gabriel Peyré. "Universal Invariant and Equivariant Graph Neural Networks". en. In: *arXiv:1905.04943 [cs, stat]* (Oct. 2019). arXiv: 1905.04943. url: http://arxiv.org/abs/1905.04943 (visited on 04/09/2021).

[150] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Third. The Johns Hopkins University Press, 1996.

[151] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015. url: https://www.tensorflow.org/.

[152] Yongqin Xian, Christoph H. Lampert, Bernt Schiele, and Zeynep Akata. *Zero-Shot Learning – A Comprehensive Evaluation of the Good, the Bad and the Ugly*. _eprint: 1707.00600. 2020.

[153] P.G. Ciarlet. *The Finite Element Method for Elliptic Problems*. ISSN. Elsevier Science, 1978. isbn: 978-0-08-087525-5. url: https://books.google.fr/books?id=TpHfoXnpKvAC.

[154] Leon Thurner, Alexander Scheidler, Florian Schafer, Jan-Hendrik Menke, Julian Dollichon, Friederike Meier, Steffen Meinecke, and Martin Braun. "Pandapower—An Open-Source Python Tool for Convenient Modeling, Analysis, and Optimization of Electric Power Systems". In: *IEEE Transactions on Power Systems* 33.6 (Nov. 2018). Publisher: Institute of Electrical and Electronics Engineers (IEEE), pp. 6510–6521. issn: 1558-0679. doi: 10.1109/tpwrs.2018.2829021. url: http://dx.doi.org/10.1109/TPWRS.2018.2829021.

[155] Xueneng Su, Chuan He, Tianqi Liu, and Lei Wu. "Full Parallel Power Flow Solution: A GPU-CPU-Based Vectorization Parallelization and Sparse Techniques for Newton–Raphson Implementation". In: *IEEE Transactions on Smart Grid* 11 (2020), pp. 1833–1844.

[156] Dong-Hee Yoon and Youngsun Han. “Parallel Power Flow Computation Trends and Applications: A Review Focusing on GPU”. In: *Energies* 13.9 (2020). issn: 1996-1073. doi: 10.3390/en13092147. url: https://www.mdpi.com/1996-1073/13/9/2147.

[157] A. Marot, B. Donnot, C. Romero, L. Veyrin-Forrer, M. Lerousseau, B. Donon, and I. Guyon. *Learning to run a power network challenge for training topology controllers. ArXiv: 1912.04211*. _eprint: 1912.04211. 2019.

[158] J. Kennedy and R. Eberhart. “Particle swarm optimization”. In: *Proceedings of ICNN'95 - International Conference on Neural Networks*. Vol. 4. 1995, 1942–1948 vol.4. doi: 10.1109/ICNN.1995.488968.

[159] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. url: http://incompleteideas.net/book/the-book-2nd.html.