LEAP Nets for System Identification and Application to Power Systems

Balthazar Donon, Benjamin Donnot, Isabelle Guyon, Zhengying Liu, Antoine Marot, Patrick Panciatici, Marc Schoenauer

LEAP Nets for System Identification and Application to Power Systems

B. Donon[†‡1], B. Donnot[†‡], I. Guyon[•‡], Z. Liu[‡],
A. Marot[†], P. Panciatici[†], M. Schoenauer[‡]
• ChaLearn, USA. ‡ UPSud/Inria, U. Paris-Saclay, France. † RTE France

Abstract

Using neural network modeling, we address the problem *system identification* for continuous multivariate systems, whose structures vary around an operating point. Structural changes in the system are of combinatorial nature, and some of them may be very rare; they may be *actionable* for the purpose of controlling the system. Although our ultimate goal is both *system identification and control*, we only address the problem of identification in this paper. We propose and study a novel neural network architecture called LEAP net, for Latent Encoding of Atypical Perturbation. Our method maps system structure changes to neural net structure changes, using structural actionable variables. We demonstrate empirically that LEAP nets can be trained with a natural observational distribution, very concentrated around a "reference" operating point of the system, and yet generalize to rare (or unseen) structural changes. We validate the generalization properties of LEAP nets theoretically in particular cases. We apply our technique to power transmission grids, in which high voltage lines are disconnected and re-connected with one-another from time to time, either accidentally or willfully. We discuss extensions of our approach to actionable variables, which are continuous (instead of discrete, in the case of our application) and make connections between our problem setting, transfer learning, causal inference, and reinforcement learning.

Keywords: System Identification, Latent Space, Residual Networks, LEAP Net, Power Systems

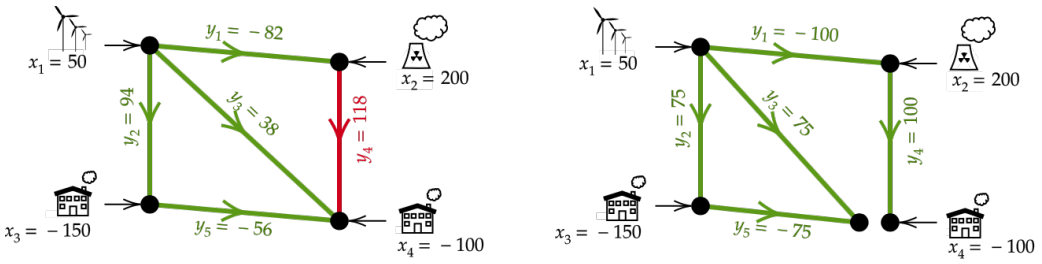

Figure 1: Electricity is transported from production nodes (top) to consumption nodes (bottom), through lines (green and red edges) connected at substations (black circles), forming a transmission *grid* of a given topology $\boldsymbol{\tau}$. Injections $\boldsymbol{x} = (x_1, x_2, x_3, x_4)$ (production or consumption) add up to zero. Grid operators (a.k.a. *dispatchers*) should maintain current flows $\boldsymbol{y} = S(\boldsymbol{x}; \boldsymbol{\tau})$ below thermal limits. Left: Line $\boldsymbol{y}_4$ goes over its thermal limit 100. Right: A change in topology (splitting of10.1007/3-540-45356-3 bottom right node) brings $\boldsymbol{y}_4$ back to its thermal limit.

1. Introduction

Neural networks have been used for identification and control since the 1990s[16]. They present the distinctive advantage that gradients can be back-propagated both through the model of the system and through the controller, if both are neural networks. In this paper, we tackle a problem of **power system identification**, with the long term goal of improving the automatic control of such systems (although the control part is not addressed in this paper). Our intermediate goal, tackled in this paper, is to develop a neural network model of the power grid, which allows us to replace *computationally costly physical simulators* in the computation of power flows, following other pioneering work [18, 13, 7, 6]. Keeping our "control objective" in mind, we adopt an approach allowing us to simulate the effect of planned coordinated actions on the grid topology (as opposed to accidental suffered changes). In particular, we can simulate the effect of changes that rarely or never occur in observational data.

Power systems have the objective of transporting electricity over long distances from production sites (*e.g.* power plants, windmills, or solar cells) to consumers, using a power transmission grid. The grid is made of power lines that are interconnected at substations. At all times, the grid must be maintained in "security", namely no lines should operate over its "thermal limit". Indeed, if a line overheats, it may be damaged, melt and/or cause fire or break. To prevent this to happen, lines are automatically put out of service if their thermal limit is exceeded for some period of time. Unfortunately,

when a line goes out of service, this results in a surcharge of lines remaining in service, increasing their risk of overheating. This may result in a cascading failure (also called black-out). **Our main charter is thus to prevent black-outs.**

Although rare, situations of crisis can occur when several line suddenly break *e.g.* due to a thunderstorm, or if fluctuations of power production or consumption are unpredictable. Highly skilled engineers (dispatchers) constantly monitor the grid to make sure it operates in security. However, operating the grid is becoming increasingly complex because of the advent of less predictable renewable energies, the globalization of energy markets, growth in consumption and concurrent limitations on new line construction. Therefore, it is becoming urgent to optimize more tightly the grid operation. This will require computationally efficient simulators of power flow to evaluate the effect of candidate actions. Even in the absence of fully automatic control, an effective neural network emulator of a power system would be very useful in routine operations: this would allow dispatchers to quickly run many simulations to evaluate the outcome of various preventive or curative actions. Although many types of actions can be considered, in this paper, we limit ourselves to one particular type of action: **reconfiguration of the grid by splitting or merging nodes at substations.**

Figure 1 illustrates the problem setting on a small example. A line going over its thermal limit will be put out of service. Hence, the grid must be reconfigured quickly to re-balance current flows before that happens. Should this happen, more lines might go over their thermal limit, which could result in a cascading failure (black-out). The figure shows a solution consisting of a node splitting in a substation. We call that a grid topology change. The space of possible grid topologies grows exponentially with the number of substations (nodes in the grid). For example, the French high-voltage transmission grid includes $N \approx 6200$ substations, with more than a dozen possible configurations per substation and thus $\gtrapprox 10^N$ possible grid topologies. Even if only a small number of those are achievable, the search space is still humongous. Our challenge is to devise a neural architecture and a training method such that, using a training dataset illustrating only a few grid tolopogies, **power flows can still be accurately predicted for topologies never seen before.**

In this paper, the approach that we present uses a neural network to emulate the power grid $\boldsymbol{y} = S(\boldsymbol{x}; \boldsymbol{\tau})$ in which the inputs $\boldsymbol{x}$ are so-called "injections" (productions and consumptions) and the outputs $\boldsymbol{y}$ are the power

flows on all the connecting lines of the grid. The system is parameterized by structural parameter vector $\boldsymbol{\tau}$ encoding various grid topologies, which includes our **actionable parameters**. We use $\boldsymbol{\tau}$ to encode changes in our neural network architecture.

The rest of the paper is organized as follows. In Section 2 we motivate our neural network architectural choices. In Section 3 we develop a mathematical formulation of our proposed approach and perform a theoretical analysis. In Section 4 we experimentally validate our method on both synthetic and real world data in our application domain. Finally, in Section 5 we propose and discuss various extensions of LEAP nets and make connections with other settings, including transfer learning, causal modeling, and reinforcement learning.

This paper is an extended version of our work [5] presented at ESANN 2019 in Bruges, Belgium. The additional contributions of this version include: a new framework in terms of system identification, a toy example to better illustrate the problem, the corresponding toy experiment and a comparison with 3 baseline methods, an enhanced mathematical formalism and two theorems, new figures illustrating of the behavior of LEAP nets, new experiments on a standard benchmark power grid, and a new discussion section.

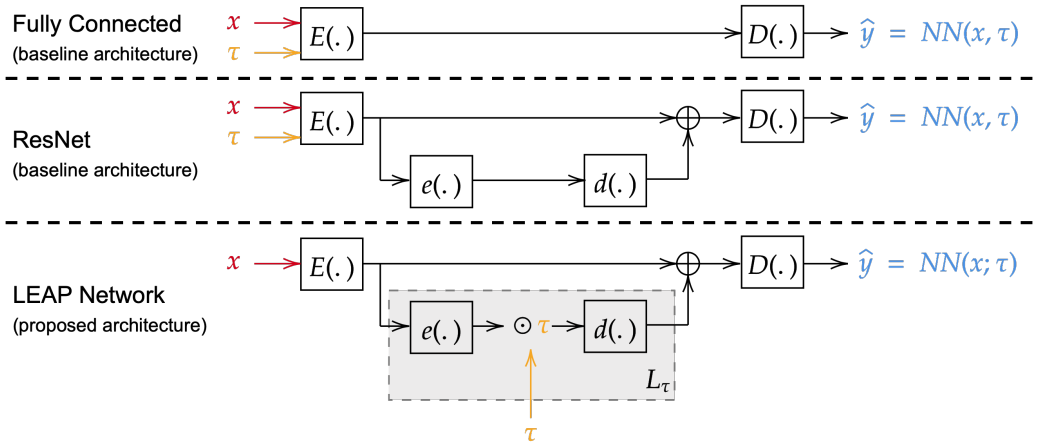

Figure 2: **Baselines and LEAP architectures:** Top: Fully Connected architecture with both $\boldsymbol{x}$ and $\boldsymbol{\tau}$ as inputs. Middle: ResNet [12] architecture, with both $\boldsymbol{x}$ and $\boldsymbol{\tau}$ as input. Bottom: Proposed LEAP net: $\boldsymbol{\tau}$ intervenes in the latent embedding space. The effect is to make a "leap" in latent space. See Section 3 for details.

2. Architectural design

In this section, we motivate the design of the LEAP net architecture (our proposed approach) with considerations of training data imbalance, and illustrate its benefits on a toy example.

2.1. Problem setting: system identification

We recall that **system identification** is a problem quite distinct to that of predictive modeling. Much of *machine learning* addresses the problem of predictive modeling in which both training and test data samples are drawn from the **same** (unknown) distribution, usually independently and identically (i.i.d. data). Under these conditions, if a predictive model makes poor predictions in depleted regions of input space, it affects only weakly its generalization performance. In contrast, we are here concerned with a system identification task, in which we want to be able to **make predictions uniformly well for all possible values of actionable parameters** (or at least all possible permitted values), to minimize the risk of making ill-informed decision, which may have catastrophic effects. The problem is particularly acute when the natural observational distribution of training data, used to fit our model (our neural network) is very concentrated around an operating point. This is the case in our power system application: most of the historical data are recorded for a so-called "reference topology" $\boldsymbol{\tau}^{\emptyset}$ (or in its close neighborhood), for a large variety of injections $\boldsymbol{x}$. In contrast, very few examples of other topologies (each for few injection values) are available for training. This calls for an architecture, which learns principally the mapping $\boldsymbol{y} = S(\boldsymbol{x}; \boldsymbol{\tau}^{\emptyset})$, and then learns corrections accounting for deviations from $\boldsymbol{\tau}^{\emptyset}$. This is why we have been inspired by the ResNet [11] architecture, see Figure 2. The main branch, consisting of encoder $E(.)$ and decoder $D(.)$, principally learns $\boldsymbol{y} = S(\boldsymbol{x}; \boldsymbol{\tau}^{\emptyset})$; meanwhile, the insert, consisting of encoder $e(.)$ and decoder $d(.)$, learns deviations from $\boldsymbol{\tau}^{\emptyset}$. We will come back to this in more details in the Section 3.

In this paper, **actionable parameters**, encoded by the discrete structural parameters $\boldsymbol{\tau}$, and other *non actionable* variables, encoded as continuous variables $\boldsymbol{x}$, play different roles. We are mostly interested in keeping the same distribution of $\boldsymbol{x}$ between training and test data[2], but re-balancing

[2]although in some cases of practical interest, the distribution of $\boldsymbol{x}$ may also drift.

the distribution of $\boldsymbol{\tau}$ in test data, as previously motivated. Changes in distribution between training and test data may call for an analysis in terms of *transfer learning* (*e.g.* [19]) or even *causal modeling* (*e.g.* [21]), since we are talking of "actions" deliberately performed by external operators at test time, hence of a "manipulated distribution". However, for simplicity, we prefer casting our problem in the framework of machine learning from *i.i.d.* data. We postpone making connections with transfer learning and causal modeling to the discussion section (Section 5). For the purpose of this paper, as stated before, in order to be deployed into production as part of a computer-assisted human decision system, our neural network model must **make predictions uniformly well on a designated distribution support of $\boldsymbol{\tau}$**. Assuming that this support is included in the support of the observational distribution, we can re-cast the problem as a regular machine learning problem, with *i.i.d.* training and test data drawn for the same observational distribution. To that end, at test time we use an objective function that re-weighs the samples to attain the desired importance for each actionable parameter $\boldsymbol{\tau}$. Equivalently, we can re-balance the test distribution by importance sampling (*e.g.* [17]) and use an un-weighted objective function. In what follows, we will interchangeably adopt either viewpoint, depending on which one is conceptually or practically more convenient.

There are other obvious departures from the *i.i.d.* framework in the setting of power systems. Triplets $\{\boldsymbol{x}, \boldsymbol{\tau}, \boldsymbol{y}\}$ are not independent of one another, they are organized in a time series. We can assume that $\boldsymbol{y}$ is independent of the rest of the data samples given $\boldsymbol{x}$ and $\boldsymbol{\tau}$, reflecting the fact that the system, *i.e.* the power grid and its physical laws, remains unchanged over time. What changes over time though is the distribution of $\boldsymbol{x}$ (and that of $\boldsymbol{\tau}$), due to daily, weekly, or seasonal variations, various weather imponderables, changes in technology, and other disruptions. Thus the *i.i.d.* assumption poorly characterizes the way in which power networks are operated. But, even though, in many ways, there is a time component to our problem, we chose here to ignore it, and treat the problem more simply. We assume a double random process in which a month is first drawn at random (*i.i.d.* from a distribution of months that does not change from year to year) and within that month a "load flow" $\{\boldsymbol{x}, \boldsymbol{\tau}, \boldsymbol{y}\}$ is drawn at random (*i.i.d.* from a distribution of load flows that occur within that month). So, everything considered, we bring ourselves back to an *i.i.d.* setting.

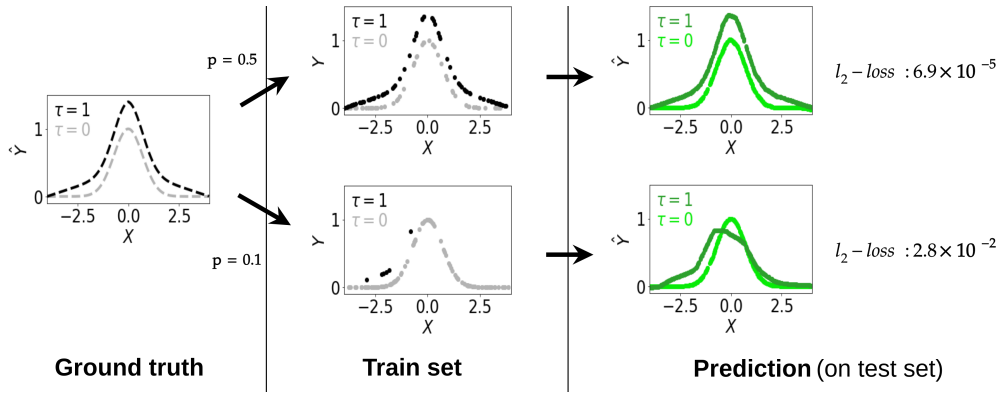

Figure 3: **Impact of data distribution imbalance on prediction accuracy (toy example)**. Left: system to be identified, see Equation 1. Top: Balanced training data *w.r.t* τ ($p = 0.5$). A standard fully connected neural network has no problem learning, reaching l_2-loss $= 6.9 \times 10^{-5}$. Bottom: Imbalanced training data, $p = 0.1$. A fully connected neural net cannot learn well the impact of $\boldsymbol{\tau}$: l_2-loss $= 2.8 \times 10^{-2}$.

2.2. A toy example

In this section, we give a first motivational toy example to show how data imbalance may hamper system identification. In this example, the system to be identified, instead of being a complex power grid, is just defined by:

$$y = S(x; \tau) = e^{-x^2} - 0.1 \times \tau \times (|x| - 4) \tag{1}$$

Here, x, τ, and y are just scalar values, but they play the same role as described before. Input values x are continuous. They are drawn for a uniform distribution $x \sim U(-4, 4)$. Structural parameters τ are discrete and have binary values 0 or 1 drawn from a Bernoulli distribution $\tau \sim B(p)$ with probability p of drawing a 1 and $(1 - p)$ of drawing a 0.

It is equivalent for our analysis to assume either that we draw training and test data *i.i.d.* from the same distribution, then re-weigh the distribution of τ at test time in our objective function to make it uniform, or to assume that test data are re-balanced, *i.e* that $\tau \sim B(0.5)$ in test data. We adopt the latter angle in Figure 3 to illustrate how a distributional imbalance due to a small value of p affects the quality of predictions for a vanilla fully connected neural network. Here $\tau = \tau^{\emptyset} = 0$ can be thought of as the indicator of the "reference" situation most frequently encountered, while $\tau = 1$ is a less typical situation, hence getting very few training samples, if p is small.

As a preview of how our LEAP net solution addresses this problem, we compare four neural networks with associated training strategies:

- *LEAP Net*, our proposed architecture as detailed in section 3 with $\boldsymbol{\tau}$ injected in the latent space.[3]
- *Fully Connected* Multi-Layer Perceptron (MLP) with $\boldsymbol{\tau}$ as an input. [4]
- *ResNet* where the leap block is replaced by a standard residual block, closer in morphology to LEAP Net, but with $\boldsymbol{\tau}$ still as an input.[5]
- *ResNet + Importance Sampling*. Same ResNet architecture, but during training the data points are weighted to compensate for the imbalance with regards to τ [3]. The idea is to give samples with $\tau = 0$ a weight of $\frac{1}{1-p}$, and samples with $\tau = 1$ a weight of $\frac{1}{p}$, so that the same emphasis is put on the two cases.[6]

For each of those four modeling strategies, and for each selected value of p, we trained 100 model instances using the Adam optimizer, with default parameters except for the learning rate that has been set to 1.10^{-3} and minibatches of 32 points. All instances are trained for 2000 epochs, ensuring that all architectures have fully converged. The training of one model takes approximately 5 minutes on a single Nvidia Titan Xp GPU. In all cases, the training set counts $N = 128$ data points drawn according to:

- $x \sim \mathcal{U}(-4, 4)$: the uniform distribution between -4 and 4
- $\tau \sim \mathcal{B}(p)$: the Bernoulli law with parameter p: $\mathbb{P}(\tau = 1) = p$

In Figure 4, we observe that the more imbalanced the distribution (smaller values of p), the larger the loss function. A too large dissimilarity between

[3]The encoder $E(\cdot)$ has 5 hidden layers of 10 neurons, $e(\cdot)$ and $d(\cdot)$ are single layer neural nets, and the decoder $D(\cdot)$ is composed of 1 hidden layer of 10 neurons. All layers in E and D use "ReLu" non-linearity. Hyperparameters tuning made us chose different values for the depths of E and D, which in our interpretation reflects the fact that the modification caused by τ is "closer to the output y than to the input x". This interpretation is based on our current understanding of what the LEAP net does, but should be further substantiated in future work.

[4]The MLP architecture consists of 6 hidden layers each having 10 neurons and with a "ReLu" activation function. It has as many hidden layers as the LEAP architecture.

[5]We note that this architecture has slightly more trainable parameters than the LEAP Net architecture, which is to its advantage. The same parameters as for the LEAP are used.

[6]The same parameters as for the LEAP are used.

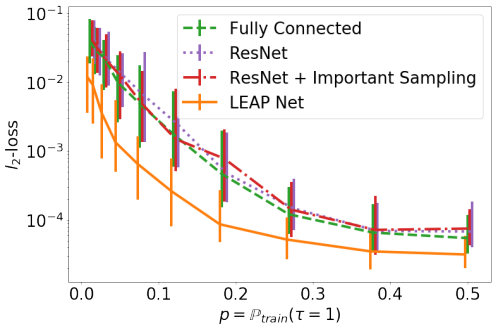

Figure 4: **Toy problem experiment. Evolution of the l_2-loss with respect to training data imbalance**. We vary parameter p, which is the probability of $\tau = 1$ in the training set (*i.e.* training data imbalance). For each of the 4 types of models and for each value of p, we trained 100 model instances. We show the median l_2-loss and the $20\% - 80\%$ percentile error bars (horizontally shifted to improve readability). Our proposed LEAP net is consistently more resilient to training data imbalance compared to the other models.

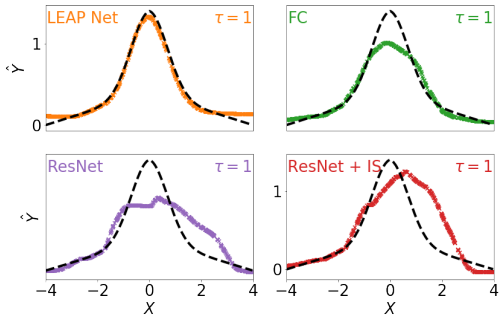

Figure 5: **Toy problem experiment. Predictions for $\tau = 1$ when $p = 0.1$.** For each of the four classes of models, we trained one model instance, in the setting of Figure 3, bottom. The ground truth for $\tau = 1$ is shown as a black dashed line and the corresponding prediction results on reweighted test data as colored dots. Our proposed LEAP net qualitatively outperforms other methods, preserving better the shape of the output.

training and test distributions is detrimental to the ability of our neural network model to learn the actual impact of parameter τ. Figure 5 shows typical predictions for one instance of each of the 4 types of models when $p = 0.1$. The LEAP makes better predictions compared to other models. Despite being trained with little training data, the LEAP net is able to generalize from the "reference" case $\tau = 0$ for which it has many values of x to the other case $\tau = 1$ for which it has only a few samples. The shape of the curves are not preserved in all the other tested models.

3. Mathematical formalism

In this section, we introduce a mathematical framework to analyze in more details the proposed LEAP architecture represented in Figure 2.

3.1. Notations and definitions

Our objective is to approximate a function $\boldsymbol{y} = S(\boldsymbol{x}, \boldsymbol{\tau})$ that maps input data $\boldsymbol{x}$ (*e.g.* power production and consumption) to output data $\boldsymbol{y}$ (*e.g.* power flows), parameterized by a discrete "grid topology vector" $\boldsymbol{\tau}$, taking values in

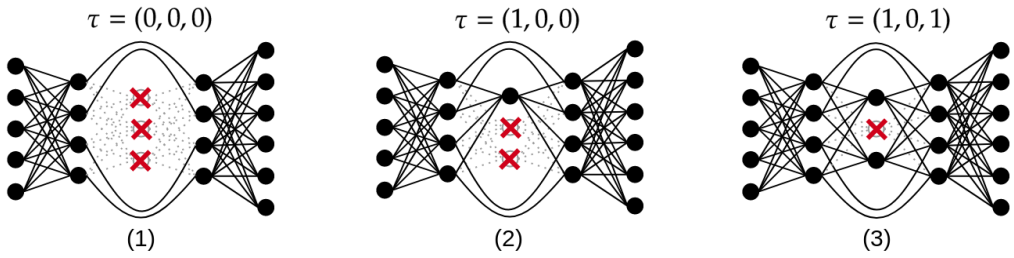

Figure 6: **Example of encoding of $\boldsymbol{\tau}$ in a LEAP net architecture.** This figure shows another view of the LEAP architecture of Figure 2. Black dots are neurons. The encoder and decoder $\boldsymbol{E}$ and $\boldsymbol{D}$ are represented by the densely connected layers on either side of the network. The direct connection between $\boldsymbol{E}$ and $\boldsymbol{D}$ is represented by curved lines bypassing the central hidden layer. The interventions in latent space (hidden layer), performed by $\boldsymbol{\tau}$, are represented by red crosses. We assume that *unary* topological changes in our system are one-hot encoded (other choices are possible). Crossed neurons are deactivated. **(1)** Architecture for $\boldsymbol{\tau} = \boldsymbol{\tau}^{\emptyset} = (0, 0, 0)$, the reference configuration. **(2)** Architecture for $\boldsymbol{\tau} = (1, 0, 0)$ creating a path through the central hidden layer. **(3)** Architecture for $\boldsymbol{\tau} = (1, 0, 1)$ combining two paths and therefore super-imposing effects.

a parameter space of potentially actionable variables (all possible power-grid topologies *e.g.* line interconnections). Triplets $\{\boldsymbol{x}, \boldsymbol{\tau}, \boldsymbol{y}\}$ are drawn *i.i.d.* according to an unknown probability distribution. In our application setting, $\boldsymbol{x}$ is drawn randomly, but $S(\cdot, \cdot)$ is a deterministic function implementing Kirchhoff's circuit laws, calculated by a physical solver that we wish to approximate.

We call **simple generalization** the capability of a neural net $\hat{\boldsymbol{y}} = NN(\boldsymbol{x}, \boldsymbol{\tau})$ to approximate $\boldsymbol{y} = S(\boldsymbol{x}, \boldsymbol{\tau})$ for test inputs $\boldsymbol{x}$ not pertaining to the training set, when $\boldsymbol{\tau}$ values are drawn *i.i.d.* from a distribution that remains the same in training and test data (this includes the case of a fixed $\boldsymbol{\tau}$). We call **super-generalization** the capability of a neural net $\hat{\boldsymbol{y}} = NN(\boldsymbol{x}, \boldsymbol{\tau})$ to approximate $\boldsymbol{y} = S(\boldsymbol{x}, \boldsymbol{\tau})$ when the natural observational distribution of $\boldsymbol{\tau}$ is very imbalanced, with a peak around a reference topology $\boldsymbol{\tau}^{\emptyset}$. We re-weigh samples in our objective function to re-balance test data to measure the correctness of system identification in regions of $\boldsymbol{\tau}$ space that are allowable actions but are under-represented in the natural distribution. Equivalently, the test set distribution is made uniform on the support of allowable actions.

The structural parameter vector $\boldsymbol{\tau}$ represents topological changes in the electricity grid, in our application setting. This prompted us to make particular choices for its encoding, although other choices are possible. One

particularity of our application related to power systems is that we have one *primary* "reference" configuration (corresponding to a reference grid topology $\boldsymbol{\tau}^{\emptyset} = (0, 0, 0, \dots)$, around which *small* variations are made. This is a generic scenario in industry for systems that operate around nominal conditions, thus we anticipate that our method could be extended to other similar situations. In our application scenario, we can easily get a lot of training data in the reference topology (corresponding to the typical way in which the grid is operated). We have comparably very little data available for training from other topologies corresponding to unary changes $\boldsymbol{\tau}^{i} = (0, 0, 1, \dots)$ (a single 1 at position i). Finally, we have extremely scarce data or no data at all available for training from double changes $\boldsymbol{\tau}^{ij}$, or higher order changes. This motivates our architectural design and how we encode $\boldsymbol{\tau}$ in the neural network architecture (see, Figure 6).

3.2. Analysis and interpretation of the architecture

Our proposed Latent Encoding of Atypical Perturbations network, or LEAP net (Figure 2), is composed of three parts: An Encoder $\boldsymbol{E}$, learning an embedding of the input data $\boldsymbol{x}$; a Decoder $\boldsymbol{D}$, learning how to perform the required task within this latent representation; and a Latent module $\boldsymbol{L_\tau}$, placed between the $\boldsymbol{E}$ and $\boldsymbol{D}$ where $\boldsymbol{\tau}$ intervenes. The overall architecture is given by:

$$\boldsymbol{L_\tau} \quad : \quad \boldsymbol{h} \quad \rightarrow \quad \boldsymbol{d}(\boldsymbol{e}(\boldsymbol{h}) \odot \boldsymbol{\tau}) \tag{2}$$

$$\hat{\boldsymbol{y}} \quad = \quad \boldsymbol{D} \circ (\boldsymbol{I} + \boldsymbol{L_\tau}) \circ \boldsymbol{E}(\boldsymbol{x}) \tag{3}$$

where $\boldsymbol{E}$ and $\boldsymbol{e}$ (encoders) and $\boldsymbol{D}$ and $\boldsymbol{d}$ (decoders) are all differentiable functions (typically implemented as artificial neural networks). The $\odot$ operation denotes the component-wise multiplication and $\circ$ the function composition. If the system is in the reference topology $\boldsymbol{\tau}^{\emptyset}$, predictions are made according to $\hat{\boldsymbol{y}} = \boldsymbol{D} \circ \boldsymbol{E}(\boldsymbol{x})$. A typical way in which we train LEAP nets is to use a lot of training data in the reference topology $\boldsymbol{\tau}^{\emptyset}$, very few examples for each of the unary changes $\boldsymbol{\tau}^{i}$, and we expect the network to generalize to target domains corresponding to double $\boldsymbol{\tau}^{ij}$ or higher level changes[7].

Our LEAP net architecture proceeds in the following way : we first embed $\boldsymbol{x}$ in a latent space by applying $\boldsymbol{E}$. Then, based on $\boldsymbol{\tau}$ and the location of $\boldsymbol{E}(\boldsymbol{x})$

[7] $\boldsymbol{\tau}^{ij}$ is defined as a vector that is 1 in both its i-th and j-th elements, and 0 everywhere else

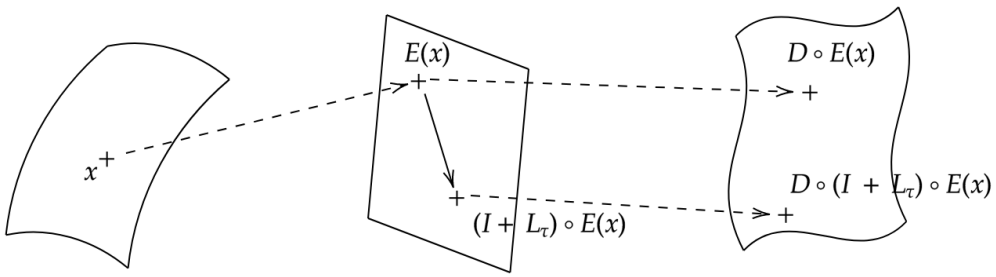

Figure 7: **Leap in latent space changes prediction $\hat{\boldsymbol{y}}$.** We schematically illustrate how input $\boldsymbol{x}$ is transformed by encoder $\boldsymbol{E}$ into $\boldsymbol{E}(\boldsymbol{x})$, then "moves" in latent space by application of vector $\boldsymbol{L}_\tau \circ \boldsymbol{E}(\boldsymbol{x})$, depending both on $\boldsymbol{E}(\boldsymbol{x})$ (*i.e.* the initial location in the latent space), and $\boldsymbol{\tau} \neq \boldsymbol{\tau}^\emptyset$. It is then decoded by $\boldsymbol{D}$ into $\hat{\boldsymbol{y}} = \boldsymbol{D} \circ (\boldsymbol{I} + \boldsymbol{L}_{\boldsymbol{\tau}}) \circ \boldsymbol{E}(\boldsymbol{x})$. We also represent at the top the "path" that $\boldsymbol{x}$ would have followed if $\boldsymbol{\tau} = \boldsymbol{\tau}^\emptyset$ (*i.e.* in the reference configuration).

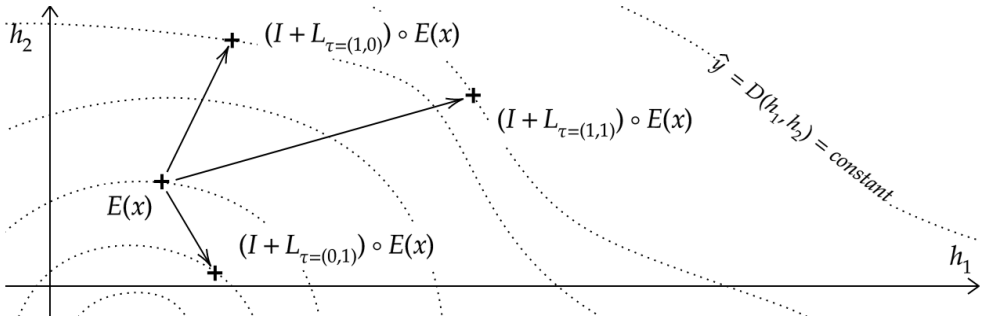

Figure 8: **Various leaps in latent space**. We schematically visualize latent space after having applied encoder $\boldsymbol{E}$. Here the output of $\boldsymbol{E}$ is only two-dimensional: $\boldsymbol{h} = (h_1, h_2)$, and $\boldsymbol{\tau}$ also has only 2 components (its dimension is unrelated to that of $\boldsymbol{h}$). The final output is one-dimensional. For a given $\boldsymbol{x}$ we can visualize the 2D location of $\boldsymbol{E}(\boldsymbol{x})$. Within this latent space we can see the results of the latent leaps for $\boldsymbol{\tau} = (1,0), (0,1)$, or $(1,1)$. One should keep in mind that the latent leap for $\boldsymbol{\tau} = (1,1)$ is not just the sum of the latent leap for $\boldsymbol{\tau} = (1,0)$ and $\boldsymbol{\tau} = (0,1)$: it can be a **combination of conditions with non-linear effects**. Moreover, the direction and norm of each of those leaps depends on the initial location within the latent space $\boldsymbol{E}(\boldsymbol{x})$. The dashed lines are contour lines of constant output $\hat{\boldsymbol{y}}$ defined as $\{\boldsymbol{h} | \boldsymbol{D}(\boldsymbol{h}) = cste\}$

within the latent space, we compute the corresponding leap $\boldsymbol{L}_{\boldsymbol{\tau}} \circ \boldsymbol{E}(\boldsymbol{x})$.[8] Then

[8]During our experiments, we observed that adding a bias in the Leap block was detrimental to its super generalisation ability. Our interpretation of this phenomenon is that by doing that, the LEAP block is never actually deactivated (*i.e.* multiplied by 0), which

we decode the signal by applying $\boldsymbol{D}$. Those latent leaps contain information about how much the system actually deviates from the reference state, and in which direction. Hence, in our architecture $\boldsymbol{L_\tau}$ only needs to learn to modulate the system response around its nominal value, disregarding of the system complexity, encoded in $\boldsymbol{E}(\cdot)$ and $\boldsymbol{D}(\cdot)$. This process is schematically illustrated in Figures 7 and 8.

While our architecture is inspired by both ResNet and Dropout, the underlying concepts are quite different:

- ResNet was developed to tackle problems of vanishing gradients in deep architectures, providing a shorter path between shallow and deep layers. In contrast, our use a single ResNet block aims at modeling a system changing around an operating point.

- Dropout is a regularization technique that randomly deactivates neurons during training in order to make the neural network more resilient. In contrast, our network alterations encode specific topological changes of the system being modeled.

In the experimental section, we compare our LEAP network to a ResNet architecture, in which topology changes are provided as an input rather than as a change in network structure, as a control experiment. We cannot compare with dropout because random disconnections would not implement specific changes in topology.

3.3. Theoretical analysis of super-generalization

In this section we formally prove the **super-generalization** capability of LEAP nets, when modeling systems with additive perturbation. It is important to note that LEAP nets are not limited to modeling additive perturbations. This simple theoretical analysis can be thought of as a "sanity check". It is easy to construct cases in which LEAP nets can fail to super-generalize. In our experimental section we will show various empirical cases of super-generalization in our application setting of power system, not limited to additive perturbations.

causes gradient to "flow through it" for every training sample. The LEAP block is thus never specialized in adjusting the latent space conditionally on some variables. We think that our proposed LEAP net is efficient only when gradient is backpropagated in its LEAP block only for selected samples, allowing for a specialization.

A system with **additive perturbations** is defined as a system $S(\boldsymbol{x}, \boldsymbol{\tau})$ that satisfies

$$\begin{cases} S(\boldsymbol{x}, \boldsymbol{\tau}^{\emptyset}) = F(\boldsymbol{x}) \\ S(\boldsymbol{x}, \boldsymbol{\tau}^{i}) = F(\boldsymbol{x}) + \epsilon_i(\boldsymbol{x}), \quad i = 1, \ldots, c \end{cases} \tag{4}$$

and

$$S(\boldsymbol{x}, \boldsymbol{\tau}^{\mathcal{I}}) = F(\boldsymbol{x}) + \sum_{i \in \mathcal{I}} \epsilon_i(\boldsymbol{x}), \quad |\mathcal{I}| \geq 2 \tag{5}$$

for some (unknown) deterministic functions $F(\boldsymbol{x}), \epsilon_1(\boldsymbol{x}), \ldots, \epsilon_c(\boldsymbol{x})$.

We begin with a theorem showing the super-generalization ability of LEAP nets in the case where a trained model makes perfect predictions on data coming from the same distribution as the training data. Then we generalize this result to the noisy case with imperfect predictions. All proofs are in appendix.

Theorem 1. (Super-generalization) *Let $S(\boldsymbol{x}, \boldsymbol{\tau})$ be a system satisfying Equations 4 and 5; and let $NN(\boldsymbol{x}, \boldsymbol{\tau})$ be a LEAP net with* linear *decoders $\boldsymbol{d}$ and $\boldsymbol{D}$. If $NN(\boldsymbol{x}, \boldsymbol{\tau})$ is trained to make perfect predictions on* **simple perturbations** *(data triplets $(\boldsymbol{x}, \boldsymbol{\tau}, \boldsymbol{y})$ coming from distribution defined by Equation 4), then it will make perfect predictions on* **combinations of perturbations** *(data coming from test distribution defined by Equation 5).*

The above theorem shows that, under some conditions, LEAP nets are indeed capable of performing *super-generalization*: making good predictions on complex structural parameters $\boldsymbol{\tau}$ while it was only trained with unary cases $\boldsymbol{\tau}^i$. The fact that $\boldsymbol{d}$ and $\boldsymbol{D}$ are linear is essential for capturing the additivity of perturbations. For other types of perturbations such as the multiplicative case defined as follows

$$S(\boldsymbol{x}, \boldsymbol{\tau}^{\mathcal{I}}) = F(\boldsymbol{x}) \prod_{i \in \mathcal{I}} (1 + \epsilon_i(x)),$$

LEAP nets can achieve similar results if $\boldsymbol{D}$ has, for example, an exponential-like behavior (*i.e.* it transforms additions into multiplications).

Despite its limitations, the fact that Theorem 1 works for any unknown functions $\epsilon_i(\boldsymbol{x})$ makes the results very general. We can consider for example linear perturbations with $\epsilon_i(\boldsymbol{x}) = \boldsymbol{W}_i \boldsymbol{x}$, constant additive perturbations with $\epsilon_i(\boldsymbol{x}) = \boldsymbol{\alpha}_i$ or heteroskedastic perturbations with $\epsilon_i(\boldsymbol{x}) = \boldsymbol{\alpha}_i G(\boldsymbol{z})$, etc. And all these types of perturbation are special cases of the above theorem.

Theorem 1 gives a strong theoretical guarantee, but one may wonder if the condition of making *perfect* predictions can be too restrictive. So we also investigated the slightly more general case of *imperfect* predictions, when the LEAP net is trained only on unary changes.

To do this, we first need to introduce a notion of distance between functions. As we used Mean Square Error (MSE) in our paper for the regression problem, the distance we use will be defined in the same flavor. Let $\mathcal{X} = \mathbb{R}^p$ be the support of $\boldsymbol{x}$, $\mathcal{Y} = \mathbb{R}^l$ be the support of $\boldsymbol{y}$. Let μ be a probability measure on $\mathcal{X}$, and f, g two functions from $\mathcal{X}$ to $\mathcal{Y}$. We define the distance $d_\mu(f, g)$ between f and g to be :

$$d_\mu^2(f, g) = \int_{\mathcal{X}} \|f(\boldsymbol{x}) - g(\boldsymbol{x})\|^2 d\mu(\boldsymbol{x}) \tag{6}$$

Here $\|\cdot\|$ represents the ℓ_2-norm on $\mathbb{R}^l$. Notice that this distance depends on the probability measure μ. If we put $\mu = D(\mathcal{X})$ (the ground truth distribution of $\boldsymbol{x}$) and write $\boldsymbol{y} = g(\boldsymbol{x})$, the right hand side becomes nothing but the generalization error[9] of f. If we instead select $\mu = \frac{1}{m}\sum_{i=1}^m \delta_{\boldsymbol{x}_i}$ (the empirical distribution), the right hand side becomes the MSE test error (or training error). We are now ready to formulate our theorem on the imperfect prediction case.

Theorem 2. *Let $S(\boldsymbol{x}, \boldsymbol{\tau})$ be a system satisfying Equations 4 and 5. Let $NN(\boldsymbol{x}, \boldsymbol{\tau})$ be a LEAP net with linear submodules $\boldsymbol{d}$ and $\boldsymbol{D}$ such that*

$$\begin{aligned} d_\mu\left(NN(\cdot, \boldsymbol{\tau}^\emptyset), S(\cdot, \boldsymbol{\tau}^\emptyset)\right) &\leq d_0 \\ d_\mu\left(NN(\cdot, \boldsymbol{\tau}^i), S(\cdot, \boldsymbol{\tau}^i)\right) &\leq d_i, \quad i = 1, \ldots, c. \end{aligned} \tag{7}$$

for some constant $d_0, d_1, \ldots, d_c \in \mathbb{R}$. Then, for any $\mathcal{I} \subset \{1, \ldots, c\}$, we have

$$d_\mu\left(NN(\cdot, \boldsymbol{\tau}^{\mathcal{I}}), S(\cdot, \boldsymbol{\tau}^{\mathcal{I}})\right) \leq (|\mathcal{I}| + 1)d_0 + \sum_{i \in \mathcal{I}} d_i. \tag{8}$$

Theorem 2 shows that as long as a LEAP neural network approximates well the ground truth functions 4 by fitting data from a given distribution of

[9]We remind that in our problem setting, the system $S(\cdot, \boldsymbol{\tau})$ we want to identify is deterministic (solutions of differential equations), the labels $\boldsymbol{y}$ are thus deterministic given $\boldsymbol{x}$ and $\boldsymbol{\tau}$. So we don't need to consider the joint distribution on the pair $(\boldsymbol{x}, \boldsymbol{y})$.

$(\boldsymbol{x}, \boldsymbol{\tau})$, its error on any points where $\boldsymbol{\tau}$ is altered and results in combination of the vector observed in the training set can be bounded too. This can indeed be considered as a super-generalization property. Notice that the weight of d_0 is larger than that of the others, which suggests we should make more efforts on improving the accuracy of the predictions on the reference topology case $\boldsymbol{\tau} = \boldsymbol{\tau}^{\emptyset}$. This was the case in all our experiments where more data came from this distribution.

Obviously, Theorem 1 is a special case of Theorem 2 with $d_0 = d_1 = \cdots = d_c = 0$.

At last, we emphasize that the distance d_μ is a very flexible notion as it can be defined as the generalization error or more importantly in practice, as the test error.

4. Empirical evaluation

In this section, we experimentally validate our proposed LEAP net architecture against baseline methods, using a toy problem and standard benchmarks.

4.1. Experimental setting, simulations, and DC approximation

In all experiments, the electricity transportation system is defined by a power grid of interconnected nodes (called buses) joined by power transmission lines. Most buses, in addition to being interconnected, are connected to productions (sources of electrical power, such as power plants) and consumptions (loads consuming power, such as home appliances), altogether referred to as "injections". We recall that we group in a vector $\boldsymbol{x}$ all injections and that we denote by vector $\boldsymbol{y}$ the resulting flows of electricity on all lines, which can be calculated with physical equations. We use Hades2[10] to compute the flows $\boldsymbol{y}$ in our experiments, the official simulator of the French power transmission operator RTE. Such calculations are called "load flow" in the jargon of power systems; they are performed with a Newton-Rhapson solver. Even in real data, flows are actually computed with Hades2 because RTE possesses very few sensors on power lines measuring actual flows.

It is worth pointing out that "load flow" equations are **non linear**, and losses caused by Joule's effect are included. Injected real and reactive power

[10]Freeware available at `http://www.rte.itesla-pst.org/`.

at a bus are functions of voltage magnitudes and voltage angles. The general equations governing a load flow are termed the "AC power flow equations". To simplify analysis, one often considers only real power via the "DC approximation," which allows one to write the vector of power injections as a linear function of the vector of voltage angles. Because the DC approximation yields somewhat faster calculations than solving the full AC power flow, we use the DC approximation as a baseline method against which we compare the LEAP nets.

One drawback of the DC approximation however, is that it requires the exact knowledge of the grid topology (line interconnections). In contrast, our LEAP net approach only requires topological changes to be known through and abstract numbering representation, which we encore with a binary vector $\boldsymbol{\tau}$ representing one-hot encoded unary changes to a reference topology $\boldsymbol{\tau}^{\emptyset}$. The LEAP net task is to learn to emulate the Newton Raphson solver S that computes power flows $\boldsymbol{y} = S(\boldsymbol{x}; \boldsymbol{\tau})$, without requiring the specific knowledge of the graph of the transmission grid. This feature is important, as we will see in real data experiments.

4.2. Illustrative experiments on a toy power grid

The objective of these first experiments is to conduct a series of comparisons of LEAP net against other neural network architectures (Fully Connected network and ResNet). We use the toy power grid illustrated in Figure 9.

In terms of system identification objective, our goal is to train a neural network emulator of a load flow simulator, which is capable of making good predictions for all possible disconnections of a single line (so-called (N-1) situation) and for all possible disconnections of two lines (so-called (N-2) situation). In the jargon of power systems, a "N-1" test (with capital N) means evaluating ALL possible single line disconnections, while a "N-2" test would mean evaluating all possible disconnections of pairs of lines.

In our experimental setting, for the distribution of $\boldsymbol{x}$, we simulate data using semi-realistic injections, inspired by real production and consumption time series. For the grid topology, in training data, we vary the topology around a reference topology $\boldsymbol{\tau} = \boldsymbol{\tau}^{\emptyset}$ by making unary changes. We give a high probability of selecting $\boldsymbol{\tau}^{\emptyset}$ and a small (identical) probability p to all unary changes (single line disconnections). Even though, in practice, higher order changes (multiple line disconnections) occur with a small probability, we do not include any higher order change in our training data, to make

the situation more extreme. In other words, the training distribution is very imbalanced with respect to $\boldsymbol{\tau}$ and by varying p we monitor the effect of this imbalance on system identification.

The training was performed on mini batches of 100 samples during 100 epochs. The input is a vector of 4 elements (2 productions, 2 consumptions), and the output is a vector of size 8. The encoder $\boldsymbol{E}$ has 4 hidden layers, with a latent dimension of 15 and ReLU activations. $\boldsymbol{e}$, $\boldsymbol{d}$ are single layer linear neural networks with respective output sizes of 8 and 15. The decoder $\boldsymbol{D}$ has 2 hidden layers, with a latent dimension of 15 and leaky ReLU activations. The training of one model takes approximately 5 minutes on a single Nvidia Titan Xp GPU.

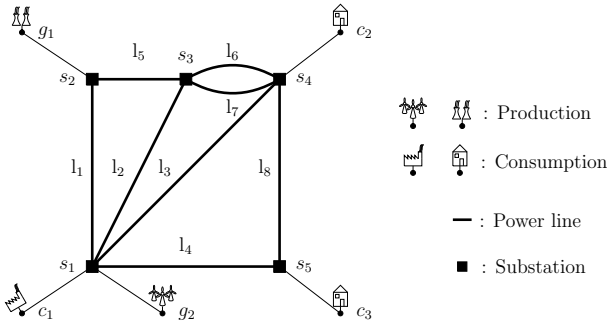

Figure 9: **Small power grid experiment**: Reference topology $\boldsymbol{\tau}^{\emptyset}$.

Our evaluation method is to use the l_2-loss on test data sampled in a way that topologies corresponding to line disconnections are re-balanced. Two distinct test sets are used, namely the "N-1" and the "N-2" test sets.

In the "N-1" test single line, disconnections are uniformly distributed. Figure 10 shows the results on the "N-1" test after training on the imbalanced training set parameterized by p. For each value of p, we have repeated experiments for 11 instances of Fully Connected neural nets, ResNets, and LEAP nets. We report the l_2-loss on the test set at the end of the learning process. One can observe that for a fixed value of p LEAP nets performs significantly better than Fully Connected nets and ResNets. The two baselines show similar results. In particular, the LEAP net fares better for small values of p, which highlights that it deals better with training distribution imbalance.

In the "N-2" test, we measure the accuracy on a set of situations in which both lines 1 and 6 are disconnected, which has never been observed in the

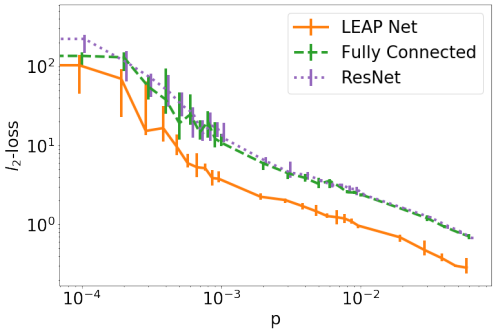

Figure 10: **Small power grid experiment "N-1" test**: p is the probability of every line to be disconnected in the training set. The "N-1" test set is composed of datapoints that were scarcely observed in the training set. This figure should not be mistaken for a learning curve.

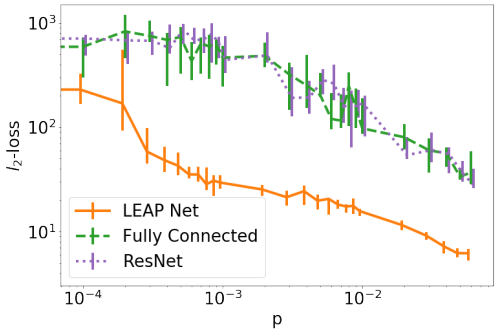

Figure 11: **Small power grid experiment "N-2" test**: p is still the probability of every line to be disconnected in the training set. The "N-2" test set is now only composed of datapoints where lines 1 and 6 are disconnected. This situation has never occurred in training data.

training set. We note that we are not doing a "full" "N-2" test in these experiments but consider a single pair. Figure 11 shows the results on the "N-2" test after being trained on the training set. For each value of p, we have fully trained multiple instances of a Fully Connected neural net, a ResNet and a LEAP net, and we plot here the resulting l_2-loss on the test set at the end of the learning process. The performances are lower than in the "N-1" test, which is normal since the situation in this test is not observed during training, but the gap between LEAP net and its contenders is larger.

4.3. Case 118 synthetic data experiment

To further assess the performance of our method, we conducted controlled experiments on a standard medium-size benchmark from "Matpower" [25], a library commonly used to test power system algorithms [1]: case118, a simplified version of the Californian power grid (dim $\boldsymbol{x} = 153$ injections and dim $\boldsymbol{y} = 186$ power lines). Power grid topology changes consist in reconfiguring line connection patterns in one or more substations (see Figure 1). Such changes are more complex than simple line disconnections considered in the previous section and in [7]. There are 11 558 possible unary actions (corresponding to single node splitting or merging, compared to the reference topology).

From this medium size power grid, we derived 3 distinct datasets:

- The *training set* is made of 150 000 independent data points: (a)

50 000 of them are sampled with injections $\boldsymbol{x}$ and the reference topology $\boldsymbol{\tau} = \boldsymbol{\tau}^{\emptyset}$ (all substations in the reference topology), (b) then 100 unary actions are sampled randomly among the 11 558 unary topologies, as well as 1 000 injections $\boldsymbol{x}$, which generates the other 100 000 samples. Compared to the smallest experiments presented in subsection 4.2, this is equivalent to having $p = 6.7 \times 10^{-3}$.

- The *simple generalization test set* consists in datapoints sampled from the same distribution as the training set. There are as many samples as in the training set.

- The *super generalization test set* is made of points drawn from a different distribution than the training one: we sampled 1 500 topologies among the 4 950 possible double actions $\boldsymbol{\tau}^{ij}$, (knowing that both $\boldsymbol{\tau}^{i}$ and $\boldsymbol{\tau}^{j}$ have been observed at training time). Then, for each of these 1 500 $\boldsymbol{\tau}^{ij}$, we sampled 100 inputs $\boldsymbol{x}$ (with the same distribution as the one used for the training and simple generalization test set). We used the same physical simulator to compute the $\boldsymbol{y}$ from the $\boldsymbol{x}$ and the $\boldsymbol{\tau}$. The super-generalization set counts then 150 000 different triplets $(\boldsymbol{x}, \boldsymbol{\tau}^{y}, \boldsymbol{y})$.

We compare the proposed LEAP net with two benchmarks: the DC approximation, a standard baseline in power systems, which is a linearization of the AC (Alternative Current) non-linear power flow equations, and the ResNet neural network architecture (Figure 2), the architecture that resembles the most to the LEAP net in terms of layer organization. Indeed, in this architecture the topology vector $\boldsymbol{\tau}$ is simply an input and the leap net is replaced with a residual layer with the same parameters. The mean-square error was optimized using the Tensorflow Adam optimizer with a learning rate of 1×10^{-5}. To make the comparison least favorable to LEAP net, all hyper-parameters (learning rates, number of units, number of layers, etc.) were optimized by cross-validation (using a separate validation set drawn iid with the same distribution as the training set) for the baseline ResNet architecture. This could explain why the LEAP net, despite learning faster at first, stops learning after some epoch while the ResNet baseline architecture keeps improving.

The training was performed on mini batches of 100 samples during 250 epochs. The input is a vector of 153 elements (54 productions, 99 consumptions), and the output is a vector of size 186. The encoder $\boldsymbol{E}$ has 6 hidden

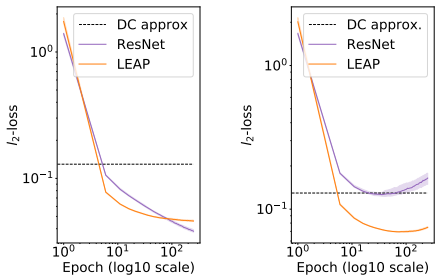

Figure 12: **Case 118 synthetic data experiment -** Neural nets trained with 15000 injections, for $\boldsymbol{\tau}^{\emptyset}$ and unary changes $\boldsymbol{\tau}^{i}$. (left) **Simple generalization.** Test injections for **unary changes** $\boldsymbol{\tau}^{i}$. (right) **Super generalization.** Test injections for **double changes** $\boldsymbol{\tau}^{i,j}$. Error bars are $20\% - 80\%$ intervals, computed over 30 repeat experiments.

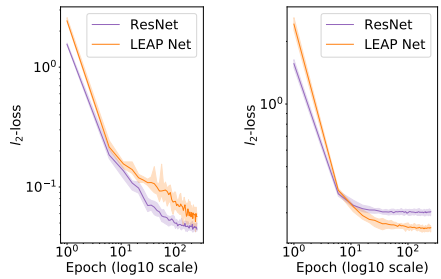

Figure 13: **Real French ultra-high voltage power grid data experiment -** The neural net in both cases is **trained from data until May 2017**. (left) **Simple generalization.** Test set made of randomly sampled data in same time period as training data. (right) **Super generalization.** Test set made of the months of June and July 2017.

layers, with a latent dimension of 600 and ReLU activations. $\boldsymbol{e}$, $\boldsymbol{d}$ and $\boldsymbol{D}$ are all single layer linear neural networks with respective output sizes of 200, 600 and 186. The training of one model takes approximately 2 hours on a single Nvidia Titan Xp GPU.

Figure 12 indicates that the LEAP net (orange curves) performs better than the DC approximation (black line) both for simple and super generalization. Figure 12b shows that the ResNet architecture (purple curve) is not viable: not only does it perform worse than the DC approximation, but its variance is quite high. While it is improving in regular generalization with the number of training epochs, its super-generalization performances get worse.

4.4. Real French ultra-high voltage power grid data experiment

We now present results on a part of the French ultra-high voltage power grid: the "Toulouse" area with 246 consumption units, 122 production units, 387 lines and 192 substations often split in a variable number of electrical nodes. The inputs $\boldsymbol{x}$ representing injections (production and consumption) are of dim $\boldsymbol{x} = 368 (= 246 + 122)$ and the outputs $\boldsymbol{y}$ (flows) of dim $\boldsymbol{y} = 387$. In this study, $\boldsymbol{x}$ and $\boldsymbol{y}$ come from real historical data from the company

RTE[11]. One important difference when using played-back data, compared to simulation, is that we cannot intervene (this is strictly observational data).

As for the previous experiment we used three distinct data sets in this setting:

- The *training set* is composed of data from January 2012 to May 2017 uniformely distributed between these two dates.
- The *simple generalization test set* is made of data coming from the same distribution as the one used for training.
- The *super generalization test set* is made of data coming from the same area of the French power grid but sampled between June and July 2017. The distribution of the data is different in this set compared to the training data. This phenomenon is in particular due to the maintenance operations: power lines are periodically disconnected from the grid for maintenance purposes. As this type of event happens only once or twice every 5-10 years for each line[12] there are some power lines out for maintenance operation in this test set that have never been disconnected in the training set.

The mean-square error was optimized using the Tensorflow Adam optimizer with a learning rate of 5×10^{-5}. Similarly to the previous experiment, all hyper-parameters (learning rates, number of units, number of layers, etc.) were optimized by cross-validation (using a separate validation set drawn iid with the same distribution as the training set) for the baseline ResNet architecture.

The training was performed on mini batches of 100 samples during 250 epochs. The input is a vector of 368 elements (122 productions, 246 consumptions), and the output is a vector of size 387. The encoder $\boldsymbol{E}$ has 16 hidden layers, with a latent dimension of 500 and ReLU activations. $\boldsymbol{e}$, $\boldsymbol{d}$ and $\boldsymbol{D}$ are all single layer linear neural networks with respective output sizes of 387, 500 and 387. The training of one model takes approximately 2 hours on a single Nvidia Titan Xp GPU.

[11] Even in real records, flows are estimated, not measured.

[12] The period of time for which a powerline is inspected can vary depending on its age, the past problems it has suffered, the corrosivity of the ambient atmosphere. In general it happens with a few years interval.

Another difference between this experiment and the previous one detailed in section 4.3, is the "action space". In real data, *actual* power grid topologies (i.e. specifying line interconnections) are not precisely recorded. Only information on *line outages* is available to us as surrogate information on topology. This makes the neural net task harder: it must learn the effects of hidden topological changes. This unfortunate loss of information on exact grid topology interventions makes it impossible for us to compare our method to the DC approximation: computing this approximation requires a full description of the topology. The results of Figure 13 yield the same conclusions as in the previous section: the LEAP model generalizes not only to data drawn from a similar distribution it was trained on (Figure 13a) but also to unseen grid states (Fig. 13b), better than the reference architecture, which is a critical property for our application.

5. Discussion and further work

In this section, we highlight differences between the approach taken in this paper (LEAP net) and another approach called "Graph Neural Solver", which we have introduced elsewhere[8]. We present various research directions, which could extend the capabilities of the LEAP net. We also make connections between our approach and related frameworks of transfer learning, causal modeling, and reinforcement learning.

5.1. Differences between LEAP net and Graph Neural Solver

Alternative approaches to the LEAP architecture to solve the power flow problem with neural networks include Graph Convolutional Networks (GCN) [14, 4]. GCNs generalize Convolutional Networks applicable to spatio-temporal data to any type of structured data whose structure can be represented by a graph. In [8], we introduced "Graph Neural Solvers" (GNS), a family of GCNs exploiting the graph structure of power grids. GNSs emulate Newton-Raphson physical solvers, assuming a **complete knowledge of the power grid structure and its injections.** The key difference between this approach and the present paper is that LEAP nets do NOT require an explicit description of the grid topology, the topology is coded in an arbitrary way via variable $\boldsymbol{\tau}$. Lack of explicit knowledge of the graph is a practical problem facing power grid operators [7]. This situation is exemplified in the real world experiments of the present paper. From the point of view of making empirical comparisons, the lack of a full description of the topology

makes it impossible to compare Graph Neural Solvers and LEAP nets in this setting. In that sense, LEAP nets are more versatile.

5.2. Extensions of LEAP net

In this paper, we have made several restrictive assumptions, which were guided by our application setting. However those could be easily lifted, although we do not have experiments to report:

- **Structural variable $\boldsymbol{\tau}$**: In our application setting, structural (actionable) variables $\boldsymbol{\tau}$ take only discrete values, but continuous values can also be considered. They would act as smooth gating factors modulating neurons' influence rather than disconnecting them. Our choice of encoding could also be changed from "one hot" encoding to *e.g.* a redundant or more compact encoding.
- **Stacked LEAP modules**: Similarly to deep ResNet networks [11], LEAP modules can be stacked with possible performance gains.

5.3. Connection with Transfer Learning

In this paper we have treated our problem from the angle of system identification and cast it as a machine learning problem from iid data, lumping distributional changes in our objective function, which re-balances the importance of various values of $\boldsymbol{\tau}$. Another angle would be to treat our problem as a true change in distribution. The issue of having to deal with changes in distributions has been extensively studied in the framework of Transfer Learning (TL) [19, 9]. To make the analogy with our problem setting, we can think of our reference case $\boldsymbol{\tau}^{\emptyset}$ as a **source domain**, and other $\boldsymbol{\tau}$ values as **target domains.**[13] Future work includes using LEAP nets to model changes in dritributions in which $\boldsymbol{\tau}$ is a latent variable rather than an actionable variable.

[13] Alternatively, the reference case and unary changes can be lumped in the source domain, then combinations of changes would make the target domain. In this context, the LEAP net architecture, as exemplified in our experimental section, can be interpreted as an architecture capable of "zero shot learning", i.e. learning combinations of perturbations from zero examples. Structural variable $\boldsymbol{\tau}$ should be interpreted as a latent variable (rather than an actionable variable) and may take continuous values.

5.4. Link to Causal Modeling

For researchers in causal modeling, we highlight a few connections, because we have borrowed from vocabulary familiar to this community: **Actions** (or actionable parameter), **Interventions**, **Manipulated distribution** [20]. The problem we addressed is not to recover the causal structure of our system under study (power grid). Such structure is assumed to be known, at least to the extent that the inputs to our system (*injections* $\boldsymbol{x}$ and *topologies* $\boldsymbol{\tau}$) are considered "causes" and the outputs (power line flows $\boldsymbol{y}$) are considered "consequences". We identify function $\boldsymbol{y} = S(\boldsymbol{x}, \boldsymbol{\tau})$ from observational data, then **intervene** on $\boldsymbol{\tau}$ and make good predictions. By intervening, we change the input distribution from the **observational distribution** to a **manipulated distribution**. Future research directions include using LEAP nets to identify causal direction, in the spirit of cause-effect pair studies [10], by comparing alternative models: $\boldsymbol{x} = S(\boldsymbol{y}, \boldsymbol{\tau})$ or $\boldsymbol{y} = S(\mathbf{x}, \boldsymbol{\tau})$.

5.5. Control problems and Reinforcement Learning

Our ultimate goal is to address identification and control problems. One of the advantages of using neural networks as system emulators is that gradient policy algorithms may be used to train a "policy network", *i.e.* a neural network controller (see *e.g.* [24, 2, 22]). This is one popular approach in reinforcement learning [23], an area of machine learning, which addresses the problem of designing "agents" (controllers). Such agents learn from examples which actions to perform in order to optimize some objective (in our case minimizing the losses while ensuring that the grid is operated in security). To that end, we have developed an open-source simulation environment to simulate small power grids[14], and we have started organizing a series of challenges in reinforcement learning, the first edition of which just ended (`l2rpn.chalearn.org`). The initial analysis of the results is promising, demonstrating that machine learning matches or exceeds conventional approaches used by professional dispatchers [15].

6. Conclusion

The LEAP net architecture has been evaluated on a number of real and artificial test cases, as well as on a synthetic example. Training was performed

[14] `https://pypownet.readthedocs.io/en/latest/`

on data triplets $(\boldsymbol{x}, \boldsymbol{\tau}, \boldsymbol{y})$, for which $\boldsymbol{\tau}$ is an actionable variable. The LEAP net generalizes not only by approximating well $\boldsymbol{y}$ for new values of $\boldsymbol{x}$ when $\boldsymbol{\tau}$ follows the observational distribution on which it was trained, but also when $\boldsymbol{\tau}$ is re-sampled uniformly on a domain included in the support of the original distribution. This property, which we called super-generalization, allows us to use LEAP nets for system identification in application, in particular, to power systems. We analyzed theoretically the properties of LEAP nets and demonstrated in some restricted case that super-generalization properties could be explained and guaranteed. We discussed the relationship between super-generalization and transfer learning and showed that LEAP nets could be used either for Domain Adaptation or for Multi-Task learning. In our experiments, we achieved a speed-up of ≈ 300 times using the LEAP net, compared to running the physical simulator, on the synthetic dataset (power grid of 118 nodes). With data stored in computer memory, our experiments on the Toulouse area attain a speed of $\simeq 2000$ times compared to running the physical simulator. These computational evaluations were carried out using a single high-end Graphical Processing Unit (GPU) Nvidia Titan X. Further work includes scaling up our method computationally to the entire French extra high voltage power grid. We also need to improve prediction accuracy before our system could be deployed to production. However, the fact that the regular generalization performance is already within an acceptable accuracy range shows great promises. We anticipate several developments of LEAP nets in other application domains involving various types of transfer learning or bias correction, and the incorporation of LEAP nets in control applications.

References

[1] O. Alsac and B. Stott. Optimal load flow with steady-state security. *IEEE transactions on power apparatus and systems*, PAS-93(3):745–751, 1974.

[2] J. Baxter and P. L. Bartlett. Infinite-horizon policy-gradient estimation. *Journal of Artificial Intelligence Research*, 15:319–350, 2001.

[3] Y. Bengio. Deep learning of representations: Looking forward. In *International Conference on Statistical Language and Speech Processing*, pages 1–37. Springer, 2013.

[4] K. Cheolmin, K. Kibaek, B. Prasanna, and A. Mihai. Graph convolutional neural networks for optimal load shedding under line contingency. In *IEEE Power Engineering Society General Meeting (to appear)*, 2019.

[5] B. Donnot, B. Donon, I. Guyon, Z. Liu, A. Marot, P. Panciatici, and M. Schoenauer. LEAP nets for power grid perturbations. In *ESANN*, Bruges, Belgium, Apr. 2019.

[6] B. Donnot, I. Guyon, M. Schoenauer, A. Marot, and P. Panciatici. Anticipating contingengies in power grids using fast neural net screening. In *IEEE WCCI 2018*, Rio de Janeiro, Brazil, July 2018.

[7] B. Donnot, I. Guyon, M. Schoenauer, A. Marot, and P. Panciatici. Fast Power system security analysis with Guided Dropout. In *26th European Symposium on Artificial Neural Networks*, Electronic Proceedings ESANN 2018, Bruges, Belgium, Apr. 2018.

[8] B. Donon, B. Donnot, I. Guyon, and A. Marot. Graph Neural Solver for Power Systems. In *International Joint Conference on Neural Networks*, Budapest, Hungary, July 2019.

[9] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. The MIT Press, 2016.

[10] I. Guyon, A. Statnikov, and B. B. Batu, editors. *Cause Effect Pairs in Machine Learning*. The Springer Series on Challenges in Machine Learning. Springer International Publishing, 2019.

[11] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *CVPR*, pages 770–778, 2016.

[12] K. He, X. Zhang, S. Ren, and J. Sun. Identity mappings in deep residual networks. In *ECCV*, pages 630–645. Springer, 2016.

[13] T. Hossen, S. J. Plathottam, R. K. Angamuthu, P. Ranganathan, and H. Salehfar. Short-term load forecasting using deep neural networks (dnn). In *2017 North American Power Symposium (NAPS)*, pages 1–6, Sept 2017.

[14] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907, 2016.

[15] A. Marot, B. Donon, I. Guyon, and B. Donnot. Learning To Run a Power Network Competition. In *CiML Workshop, NeurIPS*, Montréal, Canada, Dec. 2018.

[16] K. S. Narendra and K. Parthasarathy. Identification and control of dynamical systems using neural networks. *IEEE Transactions on neural networks*, 1(1):4–27, 1990.

[17] R. M. Neal. Annealed importance sampling. *Statistics and computing*, 11(2):125–139, 2001.

[18] T. Nguyen. Neural network load-flow. *IEE Proceedings - Generation, Transmission and Distribution*, 142:51–58(7), January 1995.

[19] S. J. Pan and Q. Yang. A survey on transfer learning. *IEEE Transactions on Knoweledge and Data Engineering*, 22(10):1345–1359, October 2010.

[20] J. Pearl. *Causality: models, reasoning and inference*, volume 29. Springer, 2000.

[21] J. Pearl. *Causality*. Cambridge university press, 2009.

[22] J. Peters and S. Schaal. Reinforcement learning of motor skills with policy gradients. *Neural networks*, 21(4):682–697, 2008.

[23] R. S. Sutton and A. G. Barto. *Reinforcement learning: an introduction*. Adaptive computation and machine learning series. The MIT Press, Cambridge, Massachusetts, second edition edition, 2018.

[24] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[25] R. D. Zimmerman and et al. Matpower. *IEEE Trans. on Power Systems*, pages 12–19, 2011.

Appendix A. Proof of Theorem 1

Theorem 1 (Section 3.3) proves that a LEAP net trained on a system with linear perturbations exhibits super-generalization properties, if it makes perfect predictions on data triplets coming from the reference configuration.

Proof. We recall the LEAP Net architecture $NN(\boldsymbol{x}, \boldsymbol{\tau})$ which gives predictions

$$\hat{\boldsymbol{y}} = NN(\boldsymbol{x}, \boldsymbol{\tau}) = \boldsymbol{D}(\boldsymbol{E}(\boldsymbol{x}) + \boldsymbol{d}(\boldsymbol{e}(\boldsymbol{E}(\boldsymbol{x})) \odot \boldsymbol{\tau})) \in \mathbb{R}.$$

Since we assumed $l = 1$, $\boldsymbol{D}$ is actually a scalar linear function. By writing $\boldsymbol{\tau} = (\tau_1, \ldots, \tau_c) = \sum_{i=1}^{c} \tau_i \boldsymbol{\tau}^i$, we can use the linearity of $\boldsymbol{d}$ and $\boldsymbol{D}$ to write the output of LEAP Net as

$$\begin{aligned}
\hat{\boldsymbol{y}} &= \boldsymbol{D}\left(\boldsymbol{E}(\boldsymbol{x}) + \boldsymbol{d}\left(\boldsymbol{e}(\boldsymbol{E}(\boldsymbol{x})) \odot \sum_{i=1}^{c} \tau_i \boldsymbol{\tau}^i\right)\right) \\
&= \boldsymbol{D}\left(\boldsymbol{E}(\boldsymbol{x}) + \sum_{i=1}^{c} \tau_i \boldsymbol{d}\left(\boldsymbol{e}(\boldsymbol{E}(\boldsymbol{x})) \odot \boldsymbol{\tau}^i\right)\right) \\
&= f_0(\boldsymbol{x}) + \sum_{i=1}^{c} \tau_i f_i(\boldsymbol{x})
\end{aligned}$$

where

$$\begin{aligned}
f_0(\boldsymbol{x}) &= \boldsymbol{D}\boldsymbol{E}(\boldsymbol{x}) \\
f_i(\boldsymbol{x}) &= \boldsymbol{D}\boldsymbol{d}\left(\boldsymbol{e}(\boldsymbol{E}(\boldsymbol{x})) \odot \boldsymbol{\tau}^i\right), \quad i = 1, \ldots, c.
\end{aligned}$$

As $NN(\boldsymbol{x}, \boldsymbol{\tau})$ makes perfect predictions for any data point $(\boldsymbol{x}, \boldsymbol{\tau}, \boldsymbol{y})$ coming from training domainswe have:

$$S(\boldsymbol{x}; \boldsymbol{\tau}) = N(\boldsymbol{x}, \boldsymbol{\tau})$$

which means that the following equalities hold

$$\begin{aligned}
F(\boldsymbol{x}) &= f_0(\boldsymbol{x}) \\
F(\boldsymbol{x}) + \epsilon_i(\boldsymbol{x}) &= f_0(\boldsymbol{x}) + f_i(\boldsymbol{x}), \quad \forall i = 1, \ldots, c.
\end{aligned}$$

So we must have

$$\begin{aligned}
F(\boldsymbol{x}) &= f_0(\boldsymbol{x}) \\
\epsilon_i(\boldsymbol{x}) &= f_i(\boldsymbol{x}), \quad \forall i = 1, \ldots, c.
\end{aligned}$$

for all $\boldsymbol{x}$. As in our case the distribution $D(\mathcal{X})$ of $\boldsymbol{x}$ doesn't depend on $\boldsymbol{\tau}$,the above equality holds for all $\boldsymbol{x}$.

So when we use the trained $NN(\boldsymbol{x}, \boldsymbol{\tau})$ to make predictions for different $\boldsymbol{\tau}$,

$$NN(\boldsymbol{x}, \boldsymbol{\tau}^{\mathcal{I}}) = f_0(\boldsymbol{x}) + \sum_{i=1}^{c} \tau_i f_i(\boldsymbol{x}) = f_0(\boldsymbol{x}) + \sum_{i \in \mathcal{I}} f_i(\boldsymbol{x}) = F(\boldsymbol{x}) + \sum_{i \in \mathcal{I}} \epsilon_i(\boldsymbol{x}) = S(\boldsymbol{x}, \boldsymbol{\tau}^{\mathcal{I}})$$

holds for any $\boldsymbol{\tau}^{\mathcal{I}}$, which concludes the proof. □

Appendix B. Proof of Theorem 2

Theorem 2 (Section 3.3) generalizes Theorem 1 to the case of imperfect predictions made by LEAP net on data triplets coming from the reference configuration.

Proof. According to (4) and (5) in Theorem 1, we can write $S(\cdot, \boldsymbol{\tau}^{\emptyset}) = F$ and $S(\cdot, \boldsymbol{\tau}^{i}) = F + \epsilon_i$. And since $\boldsymbol{d}$ and $\boldsymbol{D}$ are linear, we can write $NN(\cdot, \boldsymbol{\tau}^{\emptyset}) = f_0$ and $NN(\cdot, \boldsymbol{\tau}^{i}) = f_0 + f_i$ according to the same argument in the proof of Theorem 1. Then we can rewrite (7) as

$$\begin{aligned} d_\mu\left(f_0, F\right) &\leq d_0 \\ d_\mu\left(f_0 + f_i, F + \epsilon_i\right) &\leq d_i, \quad i = 1, \ldots, c. \end{aligned}$$

Because the distance d_μ defined above satisfies triangle inequality (and is translation invariant), we have

$$\begin{aligned} d_\mu\left(NN(\cdot, \boldsymbol{\tau}^{\mathcal{I}}), S(\cdot, \boldsymbol{\tau}^{\mathcal{I}})\right) &= d_\mu\left(f_0 + \sum_{i\in\mathcal{I}} f_i, F + \sum_{i\in\mathcal{I}} \epsilon_i\right) \\ &= d_\mu\left(f_0 + \sum_{i\in\mathcal{I}} [(f_0 + f_i) - f_0], F + \sum_{i\in\mathcal{I}} [(F + \epsilon_i) - F]\right) \\ &\leq d_\mu(f_0, F) + \sum_{i\in\mathcal{I}} d_\mu(f_0 + f_i, F + \epsilon_i) + \sum_{i\in\mathcal{I}} d_\mu(f_0, F) \\ &\leq d_0 + \sum_{i\in\mathcal{I}} d_i + \sum_{i\in\mathcal{I}} d_0 \\ &= (|\mathcal{I}| + 1)d_0 + \sum_{i\in\mathcal{I}} d_i, \end{aligned}$$

which concludes the proof. □