Learning from Interpretation Transition using Feed-Forward Neural Networks

Enguerrand Gentet[1,2], Sophie Tourret[2], and Katsumi Inoue[2,3]

[1] ENS (Cachan)/Paris-Sud University (Orsay) France,
enguerrand.gentet@ens-cachan.fr
[2] National Institute of Informatics (Tokyo) Japan,
tourret@nii.ac.jp, inoue@nii.ac.jp
[3] Tokyo Institute of Technology (Tokyo) Japan.

Abstract. Understanding the evolution of dynamical systems is an ILP topic with many application domains, e.g., multi-agent systems, robotics and systems biology. In this paper, we present a method relying on an artificial neural network (NN) to learn rules describing the evolution of a dynamical system of Boolean variables. The experimental results show the potential of this approach, which opens the way to many extensions naturally supported by NNs, such as the handling of noisy data, continuous variables or time delayed systems.

1 Introduction

Learning the dynamics of systems is an important problem in Inductive Logic Programming (ILP) [14]. Applications range from multi-agent systems, where learning other agents behavior can be crucial for decision making [11,13], to systems biology [3,16], where knowing the interaction between genes can greatly help in the creation of drugs to treat sicknesses for example. This paper introduces an algorithm, called Artificial Neural Networks for Learning From Interpretation Transition (NN-LFIT), that automatically constructs an accurate model of the dynamics of a Boolean system from the observation of its state transitions and then extracts rules describing these dynamics. It offers an alternative to the LFIT algorithm introduced in [9], where rules describing the dynamics of a system are built directly from the observation of the system transitions in a purely logical framework. Artificial neural networks (NNs) have been successfully applied to solve a large variety of predictive learning and function approximation problems [2]. The motivation behind their use here is their inherent ability to generalize observations [17] that answers a major limitation of LFIT, namely its difficulty to handle noisy data and continuous variables in partially observed systems. Previous uses of NNs for ILP include [5] that initializes and refines NNs using logic programs (symbolic knowledge) and [6,12] that focus on rule extraction from NN. The results obtained during this preliminary work are encouraging and several extensions are considered.

In Section 2, we present a formal description of the problem. In Section 3, the NN-LFIT algorithm is detailed. Section 4 contains all the results and their discussion. Finally, Section 5 concludes this paper.

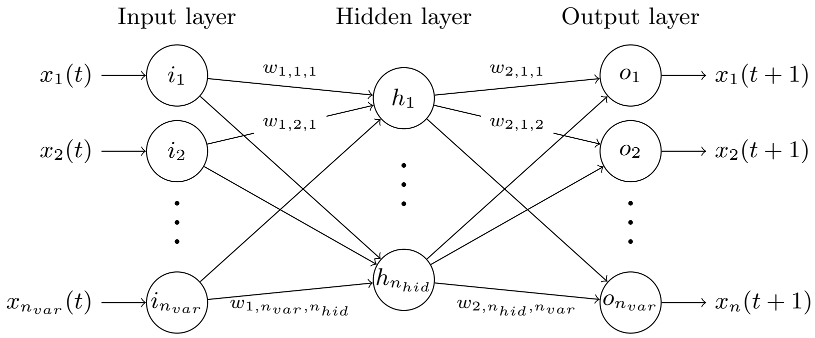


Fig. 1: NN architecture and notations used in NN-LFIT

2 Problem description

We adopt the representation of dynamical systems used in [9]. A system is a finite state vector evolving through time $\mathbf{x}(t) = (x_1(t), x_2(t), ..., x_{n_{var}}(t))$ where each $x_i(t)$ is Boolean variable. In systems biology these variables can represent, e.g., the presence or absence of some genes or proteins inside a cell. The aim of NN-LFIT is to output a normal logic program P that satisfies the condition $\mathbf{x}(t+1) = T_P(\mathbf{x}(t))$ for any t, where T_P is the immediate consequence operator for P [9]. The rules of P are of the form $\forall t, x_i(t+1) \leftarrow F(\mathbf{x}(t))$ for all i in $\{1 \ldots, n_{var}\}$ where F is a Boolean formula in propositional logic (PL). The standard terminology and notations of PL are used[1], i.e., when referring to literals (variables or negation of variables), terms (conjunctions of literals) and formulæ. We are especially concerned with formulæ in disjunctive normal form (DNF), i.e., disjunctions of terms. Note that this formalism allows us to describe only the simplest of dynamical systems, meaning those purely Boolean and without delays i.e. where $\mathbf{x}(t+1)$ depends only of $\mathbf{x}(t)$.

The type of NN used in NN-LFIT reflects the simplicity of the systems considered. We use feed-forward NNs [17], where the information moves in only one direction, forward, starting from the input layer, going through the hidden layers and ending to the output layer. We furthermore restrict ourselves to using only one hidden layer, i.e. a total of three layers, because it simplifies a lot the architecture of the NN and its treatment. This does not limit the accuracy of the NN as long as there are enough neurons in the hidden layer [8]. The notations related to the NN are introduced in Fig. 1. Formally, each i^{th} neuron on the l^{th} layer is connected to each j^{th} neuron on the $(l+1)^{th}$ layer by a link characterized by its weight denoted $w_{l,i,j} \in \mathbb{R}$ and the output of each neuron is computed from the weighted sum of all its inputs[2], e.g., $output(o_k) = f(\sum_{j=1}^{n_{hid}} w_{2,j,k} output(h_j))$, where f is a sigmoid function. The state vector $\mathbf{x}(t)$ is directly fed to the input

[1] An introduction to logic is available in, e.g., [1].

[2] Due to the space limitations, details about inner mechanisms of NNs, e.g. biases, are omitted.

layer and the output layer predicts the values of $\mathbf{x}(t+1)$. The last parameter remaining to choose is the number of neurons on the hidden layer n_{hid}, which is tuned specifically by NN-LFIT to suit each problem. To determine the correct weights of an NN, it must be trained on the available data. The standard approach consists in splitting the available data in two sets: the training set, on which the training of the NN is performed, and the test set, on which the performance of the trained NN is measured. Usually 80% of the training set is used to tune the weights of the NN while the remaining 20%, called the validation set, is used to tune the NN parameters (meaning in our case, the n_{hid} value). The training method used in NN-LFIT is standard: backward propagation with an adaptive rule on the gradient step and L2 regularization to avoid over-fitting the training data. The error made by the trained NN on each data set (resp. written E_{train}, E_{val}, and E_{test}) is the ratio of incorrect predictions made by each output neuron averaged on all output neurons.

3 The NN-LFIT algorithm

The purpose of this section is to introduce the NN-LFIT algorithm. This algorithm constructs automatically a model of a system from the observation of its state transitions and generate transition rules which describe the dynamic of the system. The main steps of NN-LFIT are listed bellow:

Step 1: Create the model of the system.

1. Choose the number of hidden neurons n_{hid} and train the NN.
   (a) Initialize n_{hid} with a trial and error algorithm.
   (b) Refine n_{hid} with a basic constructive algorithm.
2. Simplify the NN by pruning useless links.

Step 2: Extract the rules

1. Extract logical rules in DNF using a blackbox algorithm.
2. Simplify logical rules into DNF form with an external tool.

The NN building steps are inspired by the work presented in [10].

Step 1 - Creation of the model. The first building step is to generate a fully connected NN with a well fitted architecture to learn the dynamics of the observed system. We first use an initialization algorithm and then we refine the architecture with a constructive algorithm.

Initialization algorithm The initial number of neurons on the hidden layer n_{hid} is chosen using a simple trial and error algorithm. It consists in training the NN using several architectures with an incremental initial number of hidden neurons starting from one and stopping when E_{val} no longer decreases after a few tries. Every time we try a new architecture, we randomly initialize all the weights.

Constructive algorithm The architecture is improved by using a basic constructive algorithm. It uses the same principle as the initialization algorithm except that every time we add a hidden neuron, we keep all the trained weights attached to the former neurons.

Pruning algorithm The purpose of this step is to remove useless links. To do so we introduce the notion of link efficiency. To compute the efficiency of a specific link, we multiply its weight by the weights of every other link starting from (or ending to) the same hidden neuron it ends to (or starts from). In other words, the efficiency of a link quantifies the best contribution among all the paths going through this link. It is therefore logical to remove links with low efficiency because they have less effects on the predictions compared to others. We use a simple dichotomous search to remove as many links as possible without increasing E_{train}. After the pruning algorithm has been run, if some hidden neurons have lost all their links to the output layer or all their links from the input layer, they can be removed[3].

Extracting rules from the fully connected NN right after the steps 1.(a) and 1.(b) is possible. However, as shown in the experimental results, the rules extracted after the simplification step are both simpler and more accurate than those extracted before. In addition, thanks to the simplification (step 2.2), the rule extraction process is less time consuming.

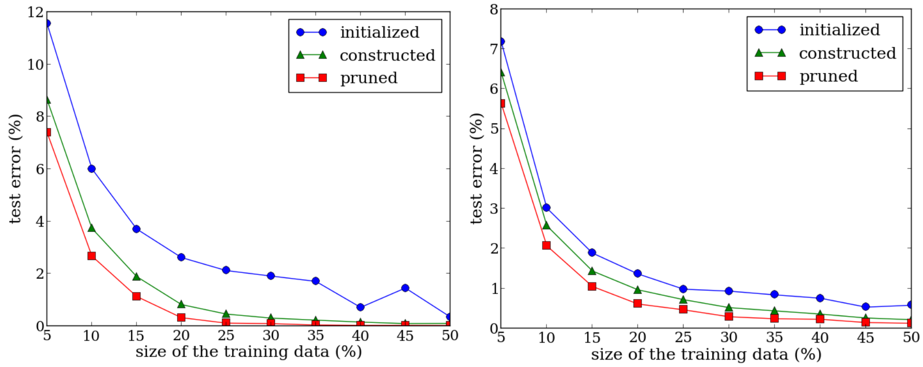
Step 2 - Extraction of the rules. To extract the rules underlying the transition system from the NN, each output neuron o_i is considered independently. First the sub-NN $\mathcal{N}_i$, made of o_i plus all the input and hidden neurons that can reach o_i and their connections to each other, is extracted from the main NN. Then, $\mathcal{N}_i$ is used as a blackbox to construct the rules. All possible input vectors are fed to $\mathcal{N}_i$ and only those that activate o_i are kept. The union of these vectors is converted into a DNF formula F that is then simplified using a tool called `primer` [15][4]. For example, let us consider a system $(x_1(t), \dots, x_4(t))$ and the NN $\mathcal{N}$ obtained by applying Step 1 of NN-LFIT on it. We focus on the neuron o_2 of $\mathcal{N}$ and consider that, due to the pruning algorithm, o_2 only depends on i_1, i_2 and i_4. We start by querying all the possible combinations of (i_1, i_2, i_4) inputs, keeping only the ones that activate o_2. Now consider that o_2 is activated only in the following cases: (1) i_1 is on, i_2 and i_4 are off;(2) i_1 and i_2 are on and i_4 is off; (3) i_1, i_2 and i_4 are on. Then o_2 is represented by the formula: $F = (i_1 \wedge \neg i_2 \wedge \neg i_4) \vee (i_1 \wedge i_2 \wedge \neg i_4) \vee (i_1 \wedge i_2 \wedge i_4)$. Finally, `primer` returns the simpler but equivalent formula $F' = (i_1 \wedge \neg i_4) \vee (i_1 \wedge i_2)$. Going back to the original transition system, the rule describing the evolution of $x_2(t)$ extracted from $\mathcal{N}$ is thus: $x_2(t+1) \leftarrow (x_1(t) \wedge \neg x_4(t)) \vee (x_1(t) \wedge x_2(t))$.

4 Experimental results

The benchmarks used in this paper are three Boolean networks from [4] also used for evaluating LFIT in [9], describing the cell cycle regulation of budding yeast, fission yeast and mammalians. We randomly assign the $2^{n_{var}}$ transitions

[3] The method to remove unreachable neurons is detailed in Appendix A for the convenience of the reviewers.

[4] The usage of `primer` is detailed in Appendix B for the convenience of the reviewers.



(a) mammalian benchmark

(b) fission benchmark

Fig. 2: Influence of the train size on E_{test} for every step of NN-LFIT.

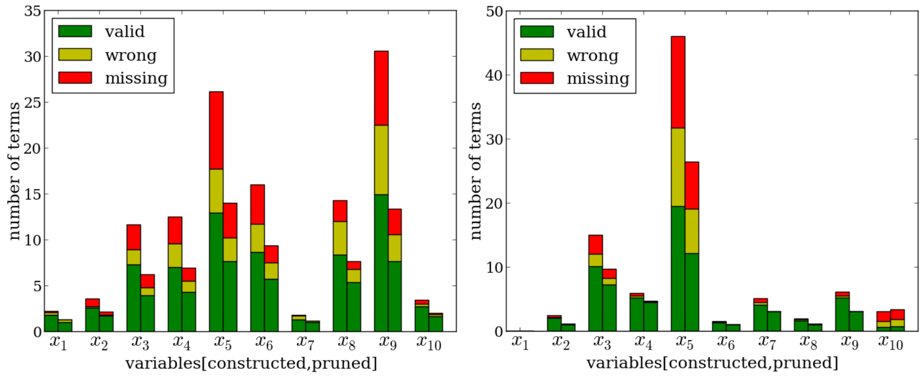
describing these networks into the test set and training set (that includes the validation set). Although it is standard to put around 80% of the available data in the training set, we want to simulate the fact that real world biological data are incomplete hence we start by analyzing the influence of the size of the training set on the accuracy of the NN (see Fig. 2)[5] It is measured by E_{test} and averaged over 30 random allocations of the data in the different sets. We observe that each successive sub-step of NN-LFIT improves the accuracy of the model and that, as expected, E_{test} decreases when the size of the training set increases. It reaches an error rate of only 1% while training only on 15% of the data and becomes negligible when the training covers 50% of the data. In comparison, LFIT [9] has a nearly constant error rate on the test set (resp. 36% and 33% on the mammalian and fission benchmarks) for all sizes of the training set. The following experiments are conducted allocating 15% of the data to the training set and the results are also averaged over 30 random allocations.

Table 1 shows the parameters of the NN architectures produced by NN-LFIT and their corresponding E_{test}. The numbers of neurons and links decrease significantly during the pruning step (16% less hidden neurons and 65% less links) along with E_{test} (29% reduction) showing that the simplification step not only reduces the complexity of the NN but also improves the model performances through an efficient generalization.

	Mammalian, $n_{var} = 10$			Fission, $n_{var} = 10$			Budding, $n_{var} = 12$		
Architecture	Neurons	Links	$E_{test}(\%)$	Neurons	Links	$E_{test}(\%)$	Neurons	Links	$E_{test}(\%)$
Initial	7.10	142	3.19	9.07	181	2.23	11.4	273	0.313
Constructed	13.5	270	1.92	13.73	275	1.61	14.4	346	0.237
Pruned	11.2	98.6	1.37	11.7	97.8	1.21	12.2	91	0.156

Table 1: Architecture and test error evolution during NN-LFIT steps.

[5] The results for the budding benchmark are omitted due to space limitations.



(a) mammalian benchmark

(b) fission benchmark

Fig. 3: Distributions of the categories of term on each variables.

Finally we evaluate the correctness and simplicity of the rules learned by NN-LFIT. For each variable x_i, we write R_i the corresponding inferred rule and R_i^* the original rule. Considering each term D in R_i and D^* in R_i^*, we identify three categories: *true positives* (valid) when $D \wedge R_i^*$ is true; *false positives* (wrong) when $D \wedge \neg R_i^*$ is true; and *false negatives* (missing) when $D^* \wedge \neg R_i$ is true. For each variable, the distribution of these categories after the construction and pruning steps of NN-LFIT are shown on Fig. 3[6]. The pruning step reduces the number of terms (true and false positives) in almost all the rules which means they are simpler. Moreover the proportion of false positives and negatives diminishes, reflecting the increase of the accuracy of the rules observed on Fig. 2.

5 Conclusion

In this paper, we present NN-LFIT, a method using feed-forward NNs to extract logic programs, describing the dynamics of systems from state measurements. Experimental results indicate good overall performances in term of correctness and simplicity of the obtained rules, even when handling only a fraction of the data. Improvements and extensions of NN-LFIT exploiting more capacities of NNs are planned. One such improvement is to extract the rules using a decompositional approach as in, e.g., [6] which details a sound but incomplete extraction algorithm improving the *complexity* × *quality* trade-off. Considered extensions also include the handling of noisy data and systems with continuous variables which can be naturally handled by feed-forward NNs. It should also be possible to use recursive NNs to model systems with delays where $\mathbf{x}(t)$ depends not only on $\mathbf{x}(t-1)$ but also on some $\mathbf{x}(t-k)$ for k greater than one. Equipped with such extensions, the field of application of NN-LFIT would encompass problems such as those found in the Dream challenges [7], including real-life data.

[6] Note that a rule of a logic program as defined in [9] is a term here, except for constant rules, e.g., x_1 in 3b which is always false and thus contains no term.

References

1. Ricardo Caferra. *Logic for computer science and artificial intelligence*. John Wiley & Sons, 2013.
2. Vladimir Cherkassky, Jerome H Friedman, and Harry Wechsler. *From statistics to neural networks: theory and pattern recognition applications*, volume 136. Springer Science & Business Media, 2012.
3. Jean-Paul Comet, Jonathan Fromentin, Gilles Bernot, and Olivier Roux. A formal model for gene regulatory networks with time delays. In *Computational Systems-Biology and Bioinformatics*, pages 1–13. Springer, 2010.
4. Elena Dubrova and Maxim Teslenko. A sat-based algorithm for finding attractors in synchronous boolean networks. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 8(5):1393–1399, 2011.
5. Artur S Avila Garcez and Gerson Zaverucha. The connectionist inductive learning and logic programming system. *Applied Intelligence*, 11(1):59–77, 1999.
6. AS d'Avila Garcez, Krysia Broda, and Dov M Gabbay. Symbolic knowledge extraction from trained neural networks: A sound approach. *Artificial Intelligence*, 125(1):155–207, 2001.
7. Alex Greenfield, Aviv Madar, Harry Ostrer, and Richard Bonneau. Dream4: Combining genetic and dynamic information to identify biological networks and dynamical models. *PloS one*, 5(10):e13397, 2010.
8. Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.
9. Katsumi Inoue, Tony Ribeiro, and Chiaki Sakama. Learning from interpretation transition. *Machine Learning*, 94(1):51–79, 2014.
10. SM Kamruzzaman and Md Monirul Islam. An algorithm to extract rules from artificial neural networks for medical diagnosis problems. *International Journal of Information Technology*, 12(8), 2006.
11. Andreas D Lattner, Andrea Miene, Ubbo Visser, and Otthein Herzog. Sequential pattern mining for situation and behavior prediction in simulated robotic soccer. In *Robot Soccer World Cup*, pages 118–129. Springer, 2005.
12. Jens Lehmann, Sebastian Bader, and Pascal Hitzler. Extracting reduced logic programs from artificial neural networks. *Applied intelligence*, 32(3):249–266, 2010.
13. David Martínez, Guillem Alenya, Carme Torras, Tony Ribeiro, and Katsumi Inoue. Learning relational dynamics of stochastic domains for planning. *Proceedings of ICAPS 2016*, pages 235–243, 2016.
14. Stephen Muggleton, Luc De Raedt, David Poole, Ivan Bratko, Peter Flach, Katsumi Inoue, and Ashwin Srinivasan. Ilp turns 20–biography and future challenges. *Machine Learning*, 86(1):3–23, 2012.
15. Alessandro Previti, Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. Prime compilation of non-clausal formulae. In *Proceedings of the 24th International Conference on Artificial Intelligence*, pages 1980–1987. AAAI Press, 2015.
16. Tony Ribeiro, Morgan Magnin, Katsumi Inoue, and Chiaki Sakama. Learning delayed influences of biological systems. *Frontiers in bioengineering and biotechnology*, 2, 2014.
17. Daniel Svozil, Vladimir Kvasnicka, and Jiri Pospichal. Introduction to multi-layer feed-forward neural networks. *Chemometrics and intelligent laboratory systems*, 39(1):43–62, 1997.

Appendices

In case of acceptance, a technical report containing the following content and some other details will be made available.

A Removal of unreachable neurons

In this article, the NN model is simplified and in particular, the fact that each hidden neuron contains an intrinsic bias, learned along the weights of the links is omitted. Due to the presence of these biases, it is not possible to simply delete unreachable hidden neurons, because even without inputs, they can still influence the output neurons they are connected to. To remove an unreachable hidden neuron h with a bias b_h, it is thus necessary to update the bias of each of the output neuron under its influence by adding to it the product of its output value (computed from b_h alone) with the weight linking the two neurons before deleting the hidden neuron. On the contrary, hidden neurons with no connection to the output layer can be removed without care since they have no influence on the results of the NN.

B Usage of `primer`

To simplify a DNF formula F, we rely on `primer` [15] to compute a prime implicant cover of F, but in practice `primer` is used to solve the dual problem of prime *implicate* covering because it only accepts CNF formulæ (i.e., conjunctions of disjunctive clauses) as inputs.

We use the standard notion of entailment in the following definitions: F_1 entails F_2, written $F_1 \models F_2$, means that all the models of F_1 are models of F_2. We say that a term D_1 covers a term D_2 when $D_1 \subseteq D_2$, which is equivalent to $D_2 \models D_1$ in propositional logic. For formulæ we say that a formula F_1 in DNF covers another formula F_2 also in DNF when the terms in F_2 are all covered by terms in F_1, and F_1 and F_2 are equivalent. Note that the definition of "cover" usually used in ILP does not require the equivalence of F_1 and F_2. A prime implicant of F is a term D such that $D \models F$ and for any D' such that $D' \models F$, if $D \models D'$ then $D' \models D$. This means that if a term D'' is such that $D'' \subset D$ and $D'' \neq D$ then $D'' \not\models F$. When handling a formula F in DNF, a formula F' syntactically simpler than F but semantically equivalent to F is obtained by replacing each term of F by a prime implicant that covers it. The notion of a prime implicate is dual to that of a prime implicant. It is a clause C such that $F \models C$ and if there exists another clause C' such that $F \models C'$ and $C \models C'$ then $C' \models C$. Intuitively, prime implicants and prime implicates can be seen respectively as the most specific conditions and the most general consequences of a formula.

`primer` is thus fed the CNF formula $\tilde{F}$ that is called the *dual* of F. It is obtained by swapping conjunctions and disjunctions in formulæ, hence transforming DNFs in CNFs and vice versa. For example, if $F = (i_1 \wedge \neg i_2 \wedge \neg i_4) \vee$

$(i_1 \wedge i_2 \wedge \neg i_4) \vee (i_1 \wedge i_2 \wedge i_4)$ then $\tilde{F} = (i_1 \vee \neg i_2 \vee \neg i_4) \wedge (i_1 \vee i_2 \vee \neg i_4) \wedge (i_1 \vee i_2 \vee i_4)$. A prime implicant cover of F is then generated by duality from the prime implicate cover of $\tilde{F}$ generated by `primer`. To develop our example, in this case `primer` returns $\tilde{F}' = (i_1 \vee \neg i_4) \wedge (i_1 \vee i_2)$ thus the cover of F is $F' = (i_1 \wedge \neg i_4) \vee (i_1 \wedge i_2)$. Note how the first term of F' covers the two first terms of F and the second one covers the two last terms of F, but still F' remains equivalent to F.