Learning Human-Understandable Description of Dynamical Systems from Feed-Forward Neural Networks

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

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

Abstract. Learning the dynamics of systems, the task of interest in this paper, is a problem to which artificial neural networks (NN) are naturally suited. However, for a non-expert, a NN is not a convenient tool. There are two reasons for this. First, the creation of an accurate NN requires fine-tuning its architecture and training parameters. Second, even the most accurate NN prediction gives no insight on the rules governing the system. These two issues are addressed in this paper, that presents a method to automatically fine-tune a NN to accurately predict the evolution of a dynamical system and to extract human-understandable rules from it. Experimental results on Boolean systems are presented. They show the relevance of this approach and open the way to many extensions naturally supported by NNs, such as the handling of noisy data, continuous variables or time delayed systems.

1 Introduction

Artificial neural networks (NNs) have been successfully applied to solve a large variety of predictive learning and function approximation problems [1]. Often, the motivation behind their use is their inherent ability to generalize observations and to handle noisy data [2]. As such, it is no wonder that the NN community has been actively researching means of understanding what happens inside NNs since nearly as long as NNs have existed [3]. To do so, the usual method is to extract a symbolic reasoning system from the NN, which can be made of, e.g., logic rules [4,5,6,7,8] or decision trees [9]. To render this extraction possible a method to build a NN with a specific architecture is usually devised first [4,5,8] but standalone extraction methods from trained NN have also been studied [10,7]. Such techniques are not only profitable to NN researchers seeking to understand what is captured by their NNs, but also for people in the field of Inductive Logic Programming (ILP) [11], aiming at constructing logic programs generalizing the observed behavior of systems given in a background theory.

This paper presents a method named NN-LFIT that uses NNs in an ILP learning context. It differs from the neural-symbolic approaches previously mentioned in that it is applied not to a standard classification problem but to the modeling of the relational dynamics of a system, i.e., of logic rules that describe the evolution of the system through time, and in that it builds NNs and rules using only the measures of the system. Examples of application include cellular automata studied by physicians and several AI sub-domains such as planning (e.g., discovering action rules), multi-agent systems (e.g., studying social networks evolutions) and systems biology [12,13] (e.g., understanding gene-protein interactions, a key component in the design of better drugs). This work is of interest to the NN community because on the one hand it enhances the methods of automatic generation and tuning of feed-forward NNs for classification tasks from [14,6] in order to deal with dynamical systems in the case of Boolean inputs and on the other hand it gives an explicit method relying on a state-of-the-art symbolic reasoning tool for the extraction of easy-to-understand rules from NNs. Moreover, the experimental results show the relevance of the neural approach which, thanks to the generalization power of NNs, is more accurate than its purely symbolic counterpart LFIT [15]. This suggests extensions such as the handling of continuous data and delayed effects that are very costly for symbolic systems like LFIT but naturally suited to NNs.

In Section 2, we present a formal description of the problem. In Section 3, the NN-LFIT algorithm is detailed. Section 4 contains the experimental results and their analysis and Section 5 concludes this paper. A short version of this article was presented at *ILP 2016* [16] but not included in the formal proceedings.

2 Problem Description

We adopt the representation of dynamical systems used in [15]. The standard terminology and notations of propositional logic (PL) are used[1], e.g., 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. In this framework a dynamical 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 a 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 [15]. 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 DNF. 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 $\mathbf{x}(t)$.

Example 1. Figure 1 is an example of application of NN-LFIT. On the left-hand side is the input problem, made of a set of observed transitions of the system.

[1] An introduction to logic is available in, e.g., [17].

Input
transitions:
$(p(t), q(t), r(t)) \rightarrow (p(t+1), q(t+1), r(t+1))$

$(1,1,1) \rightarrow (1,1,1) \mid (1,0,1) \rightarrow (0,0,1)$
$(0,1,1) \rightarrow (1,0,0) \mid (0,0,1) \rightarrow (0,0,0)$
$(1,1,0) \rightarrow (1,1,1) \mid (0,1,0) \rightarrow (1,0,0)$
$(1,0,0) \rightarrow (0,1,1) \mid (0,0,0) \rightarrow (0,0,0)$

$\xrightarrow{NN-LFIT}$

Output
logic program:

$p(t+1) \leftarrow q(t)$
$q(t+1) \leftarrow$
$(p(t) \wedge \neg r(t))$
$\vee (p(t) \wedge q(t))$
$r(t+1) \leftarrow p(t)$

Fig. 1: An application of NN-LFIT

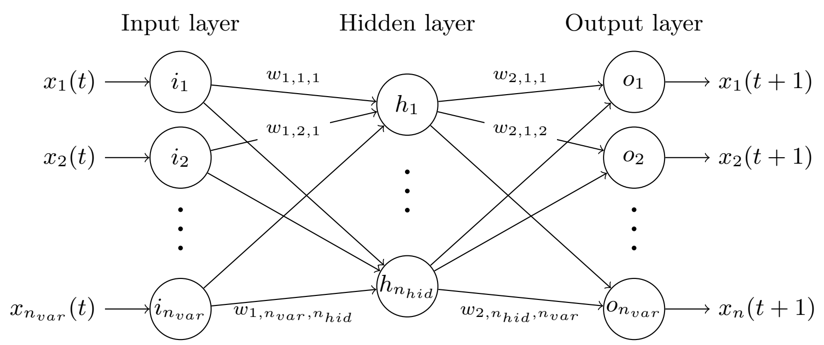


Fig. 2: NN architecture and notations used in NN-LFIT

For example, the transition $(1,0,1) \rightarrow (0,0,1)$ indicates that if at time t, $p = 1$, $q = 0$ and $r = 1$, then at time $t+1$, $p = 0$, $q = 0$ and $r = 1$. On the right-hand side is the logic program outputed by NN-LFIT. For instance, the first rule of this program mean that p is true at time $t+1$ iff q is true at time t and the second rule means that q is true at time $t+1$ iff either p is true and r is false at time t or p and q are true at time t.

The type of NN used in NN-LFIT reflects the simplicity of the systems considered. We use feed-forward NNs [2] and 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 [18]. The user is assumed to be familiar with the notion of feed-forward NN, and the notations used in this paper are introduced in Fig. 2. The state vector $\mathbf{x}(t)$ describing the dynamical system is directly fed to the input layer and the output layer predicts the values of the next state $\mathbf{x}(t+1)$. This fixes the the number of neurons on the input and output layer to the number of variables in the system. The activation function of the neurons is a sigmoid and the training method used is standard: backward propagation with an adaptive rule on the gradient step and L2 regularization to avoid over-fitting the training data. The errors made by the trained NN on the training, validation[2] and test sets are written respectively

[2] Note that, as is usual, the validation set is made of 20% of the training set

by E_{train}, E_{val}, and E_{test} and denote as usual the ratio of incorrect predictions made by each output neuron averaged on all output neurons. The only parameter remaining to choose is the number of neurons on the hidden layer n_{hid}, which is automatically tuned by NN-LFIT to suit each problem as described in the following section.

3 The NN-LFIT Algorithm

This section introduces the details of the NN-LFIT algorithm. This algorithm automatically constructs 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 by querying the NN.
2. Simplify the logical rules into DNF with an external tool.

Step 1 is based on a dynamic node creation algorithm, which was originally proposed in [14] and has been used in the REANN algorithm [6] for classification tasks with a small number of output classes. Major differences between this work and REANN are explicitly indicated in the following description. Step 2 is an original contribution.

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. In REANN, this step is skipped. The constructive algorithm is directly used on a randomly initialized NN with only one neuron in the hidden layer. For real problems, one or two hidden neurons are unlikely to be enough. Thus the initialization algorithm speeds up the training process by identifying roughly the number of neurons needed before the constructive algorithm, of which the training converges more slowly, fixes this number.

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 a hidden neuron is added, the trained weights attached to the other neurons are left unchanged.

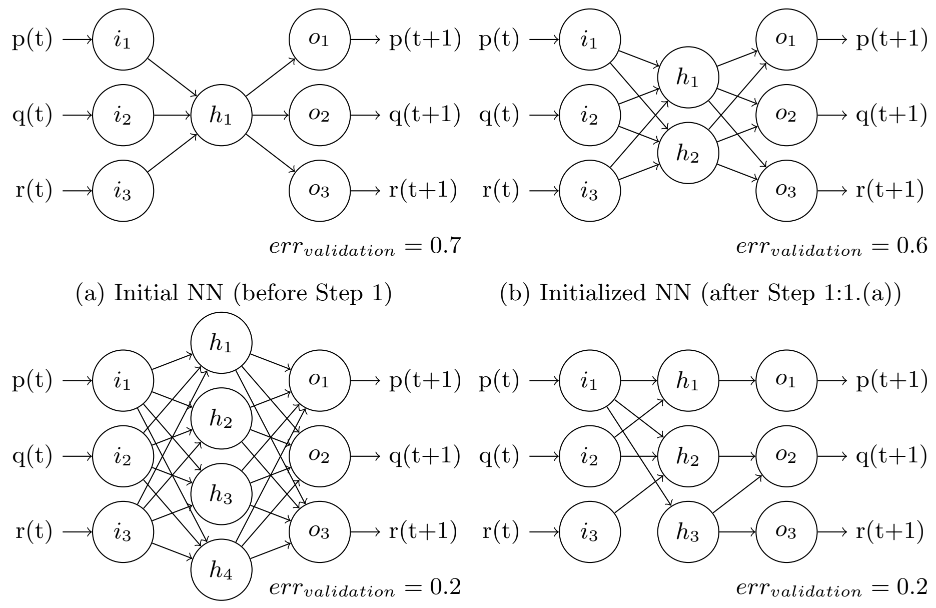
Pruning algorithm The purpose of this step is to remove useless links. To do so we introduce the notion of link efficiency. To compute the efficiency of a specific link, we multiply its weight by the weights of every other link starting from (or ending to) the same hidden neuron it ends to (or starts 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. Due to the presence of biases in the neurons activation functions, 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 neurons 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 do not influence the output of the NN.

The REANN algorithm, that handles non-Boolean inputs, includes a discretization step which is unneeded here.

Example 2. Figure 3 shows the NNs obtained after applying each sub-step of Step 1 on the system described in Ex. 1. The weights are omitted to improve the readability. The error rate on the validation set is given at each step.

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 black box 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 by computing a prime implicant cover of it using a tool called `primer` [19]. Formally, 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 $D'' \subseteq D$ and $D'' = D$ then $D'' \not\models F$. 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. When handling a formula in DNF, a formula syntactically simpler than but semantically equivalent is obtained by replacing each term of the formula by a prime implicant that subsumes it. To simplify F , we rely on `primer` to compute a prime implicate cover of the CNF formula $\tilde{F}$ that is called the dual of



(a) Initial NN (before Step 1)

(b) Initialized NN (after Step 1:1.(a))

(c) Constructed NN (after Step 1:1.(b))

(d) Pruned NN (after Step 1:2)

Fig. 3: Step 1 of NN-LFIT

F. It is obtained by swapping conjunctions and disjunctions in formulæ, hence transforming DNFs in CNFs and vice versa. This is done because primer only accepts CNF inputs. A prime implicant cover of F is then generated by duality from the prime implicate cover of $\tilde{F}$ generated by primer.

Example 3. Let us consider the neuron o_1 of the NN drawn in Fig. 3d that represents the system of Ex. 1. Due to the simplification of the network, o_1 only depends on i_1 and i_2. Then using $\mathcal{N}_1$ as a black box, we query all the different combinations of (i_1, i_2) inputs, keeping only the ones that activate o_2. In this example, o_1 is activated only in the following cases:

- i_1 is off and i_2 is on;
- i_1 is on and i_2 is on.

Then o_1 can be represented by the formula: $F_1 = (\neg i_1 \wedge i_2) \vee (i_1 \wedge i_2)$. Finally, the simplification of the formula F_1 is done by computing a prime implicant cover of F_1 as explained previously, resulting in the creation of the formula $F_1' = i_2$. Note how the term of F_1' subsumes the two terms of F_1 making F_1' equivalent to F_1. Going back to the original transition system, the rule describing the evolution of p extracted from the NN is thus: $p(t+1) \leftarrow q(t)$.

Now let us consider the neuron o_2 which, this time, depends on all the inputs i_1, i_2 and i_3. Then using $\mathcal{N}_2$ as a black box, we query all the different combinations of (i_1, i_2, i_3) inputs, keeping only the ones that activate o_2. In this example, o_2 is activated only in the following cases:

- i_1 is on, i_2 and i_3 are off;
- i_1 and i_2 are on and i_3 is off;
- i_1 ,i_2 and i_3 are on.

Then o_2 can be represented by the formula: $F_2 = (i_1 \land \neg i_2 \land \neg i_3) \lor (i_1 \land i_2 \land \neg i_3) \lor (i_1 \land i_2 \land i_3)$. Finally, the simplification of the formula F_2 is done by computing a prime implicant cover of F_2 as explained previously, resulting in the creation of the formula $F_2' = (i_1 \land \neg i_4) \lor (i_1 \land i_2)$. Note how the first term of F_2' subsumes the two first terms of F_2 and the second one subsumes the two last ones of F_2 , making F_2' equivalent to F_2 . Going back to the original transition system, the rule describing the evolution of q extracted from the NN is thus: $q(t+1) \leftarrow (p(t) \land \neg r(t)) \lor (p(t) \land q(t))$

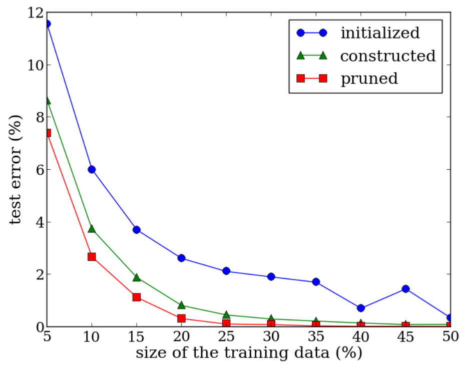
Finally, the neuron o_3 only depends on i_1. Then using $\mathcal{N}_3$ as a black box, we query the two different combinations of i_1. o_3 is activated only when i_1 is on. Then o_3 can be represented by the formula: $F_3 = i_1$. The only term is already a prime implicant of F_3 , the rule describing the evolution of q extracted from the NN is thus: $r(t+1) \leftarrow p(t)$.

Note that extracting rules from the fully connected NN right after the steps 1.(a) and 1.(b) using the exact same method is possible. However, as shown in the experimental results, the performances of the NN are better after all the steps. In addition, thanks to the pruning (step 1.2), the rule extraction process is less time consuming because the number of input variables to consider for each output can be significantly smaller than before the pruning.

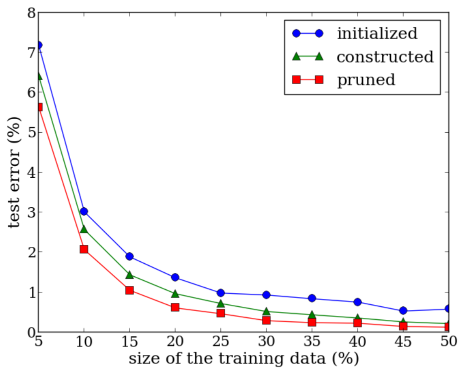
4 Experimental Results

The benchmarks used in the experiments are three Boolean networks from [20] also used for evaluating LFIT in [15]. They respectively describe the cell cycle regulation of budding yeast, fission yeast and mammalians. We randomly assign the $2^{n_{var}}$ transitions 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 data are often incomplete especially in biology, hence we start by analyzing the influence of the size of the training set on the accuracy of the NN (see Fig. 4)[3]. 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 [15] 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. Obviously the accuracy of the NN varies depending on the system it models but still these results show that the generalization power of NNs

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



(a) mammalian benchmark



(b) fission benchmark

Fig. 4: Influence of the train size on E_{test} for every step of NN-LFIT.

is a real advantage over a purely symbolic approach. The following experiments are conducted allocating 15% of the data to the training set and the results are also averaged over 30 random allocations.

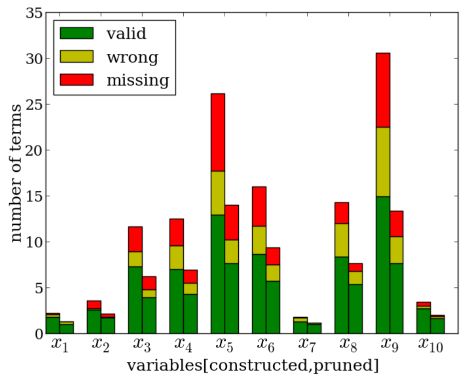
Table 1 shows the parameters of the NN architectures produced by NN-LFIT and their corresponding E_{test} as well as the error rate of LFIT on the test set, already mentioned in the previous experiment. The numbers of neurons and links decrease significantly during the pruning step (16% less hidden neurons 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. In addition, the accuracy of NN-LFIT clearly outperforms that of LFIT.

	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
LFIT	-	-	36	-	-	33	-	-	-

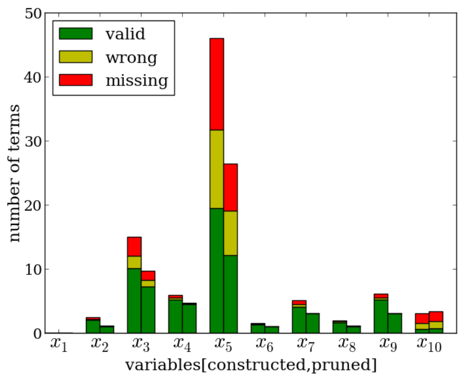
Table 1: Architecture and test error evolution during NN-LFIT steps.

Finally we evaluate the correctness and simplicity of the rules learned by NN-LFIT. For each variable x_i, we identify three categories: true positives, i.e. *valid* rules that output the same result as the original ones; false positives, i.e. *wrong* rules that contradict the original ones; and false negatives, i.e. *missing* rules that appear in the original model but are not present in the reconstructed one. Fig. 5 shows the distribution of these categories after the construction and pruning steps of NN-LFIT for each variable[4]. The pruning step reduces the number of terms (true and false positives) in almost all the rules which means they are

[4] Note that a rule of a logic program as defined in [15] is a term here, except for constant rules, e.g., x_1 in Fig. 5b which is always false and thus contains no term.



(a) mammalian benchmark



(b) fission benchmark

Fig. 5: Distributions of the categories of term on each variables.

simpler. Moreover the proportion of false positives and negatives diminishes after the pruning, reflecting the increase of the accuracy of the rules observed on Tab. 1.

5 Conclusion

In this paper, we present NN-LFIT, a method using feed-forward NNs to extract a logic program describing a dynamical system from the observation of its evolution. It includes a method to automatically tune a feed-forward NN to predict the evolution of the considered Boolean system and an original mechanism for the extraction of human-understandable rules from the NN. Experimental results indicate good overall performances in term of correctness and simplicity of the obtained rules, even when handling only as little as 15% of the data. Extensions of NN-LFIT exploiting more capacities of NNs are planned. One possibility is to extract the rules using a decompositional approach as in, e.g., [10] which details a sound but incomplete extraction algorithm improving the $complexity \times quality$ trade-off. Other extensions include the handling of noisy data and systems with continuous variables which can be naturally handled by feed-forward NNs. We are also considering how to use deep 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.

References

1. Cherkassky, V., Friedman, J.H., Wechsler, H.: From statistics to neural networks: theory and pattern recognition applications. Volume 136. Springer Science & Business Media (2012)
2. Svozil, D., Kvasnicka, V., Pospichal, J.: Introduction to multi-layer feed-forward neural networks. Chemometrics and intelligent laboratory systems **39**(1) (1997) 43–62

3. Augasta, M.G., Kathirvalavakumar, T.: Rule extraction from neural networksa comparative study. In: Pattern Recognition, Informatics and Medical Engineering (PRIME), 2012 International Conference on, IEEE (2012) 404–408
4. Carpenter, G.A., Tan, A.H.: Rule extraction: From neural architecture to symbolic representation. Connection Science **7**(1) (1995) 3–27
5. Garcez, A.S.A., Zaverucha, G.: The connectionist inductive learning and logic programming system. Applied Intelligence **11**(1) (1999) 59–77
6. Kamruzzaman, S., Islam, M.M.: An algorithm to extract rules from artificial neural networks for medical diagnosis problems. International Journal of Information Technology **12**(8) (2006)
7. Lehmann, J., Bader, S., Hitzler, P.: Extracting reduced logic programs from artificial neural networks. Applied intelligence **32**(3) (2010) 249–266
8. Towell, G.G., Shavlik, J.W.: Extracting refined rules from knowledge-based neural networks. Machine learning **13**(1) (1993) 71–101
9. França, M.V.M., Garcez, A.S.d., Zaverucha, G.: Relational knowledge extraction from neural networks. (2015)
10. Garcez, A.d., Broda, K., Gabbay, D.M.: Symbolic knowledge extraction from trained neural networks: A sound approach. Artificial Intelligence **125**(1) (2001) 155–207
11. Muggleton, S., De Raedt, L., Poole, D., Bratko, I., Flach, P., Inoue, K., Srinivasan, A.: Ilp turns 20–biography and future challenges. Machine Learning **86**(1) (2012) 3–23
12. Comet, J.P., Fromentin, J., Bernot, G., Roux, O.: A formal model for gene regulatory networks with time delays. In: Computational Systems-Biology and Bioinformatics. Springer (2010) 1–13
13. Ribeiro, T., Magnin, M., Inoue, K., Sakama, C.: Learning delayed influences of biological systems. Frontiers in bioengineering and biotechnology **2** (2014)
14. Ash, T.: Dynamic node creation in backpropagation networks. Connection Science **1**(4) (1989) 365–375
15. Inoue, K., Ribeiro, T., Sakama, C.: Learning from interpretation transition. Machine Learning **94**(1) (2014) 51–79
16. Gentet, E., Tourret, S., Inoue, K.: Learning from interpretation transition using feed-forward neural networks. In: CEUR Workshop Proceedings of the 26th International Conference on Inductive Logic Programming (ILP 16 short papers). (2016)
17. Caferra, R.: Logic for computer science and artificial intelligence. John Wiley & Sons (2013)
18. Hornik, K., Stinchcombe, M., White, H.: Multilayer feedforward networks are universal approximators. Neural networks **2**(5) (1989) 359–366
19. Previti, A., Ignatiev, A., Morgado, A., Marques-Silva, J.: Prime compilation of non-clausal formulae. In: Proceedings of the 24th International Conference on Artificial Intelligence, AAAI Press (2015) 1980–1987
20. Dubrova, E., Teslenko, M.: A sat-based algorithm for finding attractors in synchronous boolean networks. IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB) **8**(5) (2011) 1393–1399