HAL open science

Learning Logic Program Representation for Delayed Systems With Limited Training Data

Yin Jun Phua, Tony Ribeiro, Sophie Tourret, Katsumi Inoue

To cite this version:

Yin Jun Phua, Tony Ribeiro, Sophie Tourret, Katsumi Inoue. Learning Logic Program Representation for Delayed Systems With Limited Training Data. the 27th International Conference on Inductive Logic Programming, Sep 2017, Orléans, France. hal-01766236

HAL Id: hal-01766236

https://hal.science/hal-01766236v1

Submitted on 13 Apr 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons CC0 - Public Domain Dedication 4.0 International License

Learning Logic Program Representation for Delayed Systems With Limited Training Data

Yin Jun Phua[1] Tony Ribeiro[3] Sophie Tourret[2] Katsumi Inoue[2,1]

[1]Tokyo Institute of Technology, Japan
{phua,inoue}@il.c.titech.ac.jp

[2]National Institute of Informatics, Japan
{tourret,inoue}@nii.ac.jp

[3]Laboratoire des Sciences du Numérique de Nantes, France
tony.ribeiro@ls2n.fr

Abstract.

Understanding the influences between components of dynamical systems such as biological networks, cellular automata or social networks provides insights to their dynamics. Influences of such dynamical systems can be represented by logic programs with delays. Logical methods that learn logic programs from observations have been developed, but their practical use is limited since they cannot handle noisy input and need a huge amount of data to give accurate results. In this paper, we present a method that learns to distinguish different dynamical systems with delays based on Recurrent Neural Network (RNN). This method relies on Long Short-Term Memory (LSTM) to extract and encode features from input sequences of time series data. We show that the produced high dimensional encoding can be used to distinguish different dynamical systems and reproduce their specific behaviors.

Keywords: dynamical systems, Boolean networks, attractors, learning from interpretation transition, delayed systems

1 Introduction

Being able to learn the dynamics of an environment purely by observing has many applications. For example, in multi-agent systems where learning other agents' behavior without direct access to their internal state can be crucial for decision making [10]. In system biology, learning the interaction between genes can greatly help in the creation of drugs to treat sicknesses [17].

1.1 Problem Statement

Having an understanding of the dynamics of a system allows us to produce predictions of the system's behavior. Being able to produce predictions means that we can weigh between different options and evaluate their outcome from a given state without taking any action. In this way, learning about the dynamics of a system can aid in planning [13].

In most real world systems, we do not have direct access to the rules that govern the systems. What we do have, however, is the observation of the systems' state at a certain time step, or a series of observations if we look long enough. Therefore, the problem is to learn the dynamics of systems purely from the observations that we are able to obtain.

Copyright © by the paper's authors. Copying permitted for private and academic purposes.

In: Nicolas Lachiche, Christel Vrain (eds.): Late Breaking Papers of ILP 2017, Oéans, France, September 4-6, 2017, published at http://ceur-ws.org

Several learning algorithms have been proposed, that learn rules for a system, provided that the observations given cover every case that can happen within the system. However, most real world systems, particularly in system biology, obtaining data for even a short amount of time is difficult, time consuming and expensive. Therefore most current learning algorithms, while complete, are not practical in the biology setting. In addition to that, most real world observations that can be obtained are often full of random noise. Therefore, dealing with noise is also an integral part in solving this problem. The focus of this paper is therefore on being able to learn the rules despite some of the rules not having manifested in the observation. We also consider the setting in which actions from past states are able to have a delayed influence on the current state. In addition, our proposed model can also deal with noise within the data, that no previous approaches dealt with, as shown in the experiments section.

1.2 Proposed Approach

In this paper, we propose an approach to this problem utilizing Recurrent Neural Networks (RNN) to learn a logic program representation from a series of boolean state transitions. Our method is based on a framework called Learning from Interpretation Transition (LFIT) [9]. LFIT is an unsupervised learning algorithm, that can learn logic programs describing fully the dynamics of the system, purely by observing state transitions. In our approach, we construct two neural networks, one for encoding the observed state transitions, and another one of which to produce the logic program representation for the system. The idea behind this is that given a series of state transitions with a large enough length, it should be possible to uniquely identify the system. Therefore we can transform this into a classification problem, in which we attempt to classify which logic program a specific series of state transition belongs to. Neural networks are known to be good at performing classification, which makes them suitable tools for our proposed approach.

Our proposed approach works well even with a limited amount of data. This is possible because the neural network used in our model is not trained to model the dynamical system itself, but rather to output a classification of different systems. Therefore, it can be trained on artificial data prior to being applied to real data. Thus it is easy to see that the amount of data obtained has no direct relation with the performance of our model.

The rest of the paper is organized as follows. We cover some of the prior researches in Section 2, following by introducing the logical and neural network background required in Section 3. Then we present the RNN-LFIT approach in Section 4. We pursue by presenting an experimental evaluation demonstrating the validity of an approach in Section 5 before concluding the paper in Section 6.

2 Related Work

2.1 Standard LFIT

One way of implementing the LFIT algorithm is by relying on a purely logical method. In [9], such an algorithm is introduced. It constructs an NLP by doing bottom-up generalization for all the positive examples provided in the input state transition. An improved version of this algorithm, utilizing binary decision diagrams as internal data structures, was introduced in [15]. These methods, while proven to be theoritically correct, generate rules from every positive examples. The resulting NLP has been proven to be non-minimal, and thus not very *human-friendly*. To allow practical use of the resulting NLP, a method for learning minimal NLP was introduced in [14]. In [16], an algorithm that learns delayed influences, that is cause/effect relationship that may be dependent on the previous k time steps, is introduced. Another recent development in the prolongation of the logical approach to LFIT is the introduction of an algorithm which deals with continuous values [18].

This class of algorithms that utilizes logical methods, are proven to be *complete* and *sound*, however a huge disadvantage with these methods is that the resulting NLP is only representable of the observations that have been fed to the algorithm thus far. Any observations that did not appear in the input, will be predicted as either to be always true or always false depending on the algorithm used.

2.2 NN-LFIT

To deal with the shortcomings stated in the previous paragraph, an algorithm that utilizes neural networks (NN) was proposed [6]. This method starts by training a feed-forward NN to model the system that is being observed. The NN, when fully trained, should predict the next state of the system when provided with the current state observation. Then, there is a pruning phase where weak connections inside the NN are removed in a manner that doesn't affect the prediction accuracy. After the pruning phase, the algorithm extracts rules from the network

based on the remaining connections within the NN. To do so, a truth table is constructed for each variable. The truth table contains variables only based on observing the connections from the outputs to the inputs of the trained and pruned NN. A simplified rule is then constructed from each truth table. In [6], it is shown that despite reducing the amount of training data, the resulting NLP is still surprisingly accurate and representative of the observed system. However, this approach does not deal with systems that have inherent delays.

2.3 Other NN-based Approaches

There are also several other approaches attempting to tie NNs with logic programming [4,5]. In [4], the authors propose a method to extract logical rules from trained NNs. The method proposed deals directly with the NN model, and thus imposes some restrictions on the NN architecture. In particular, it was not made to handle delayed influences in the system. In [5], a method for constructing NNs from logic program is proposed, along with a method for constructing RNNs. However this approach requires background knowledge, or a certain level of knowledge about the observed system (such as an initial NLP to improve on) before being applicable.

In [11], the authors proposed a method for constructing models of dynamical systems using RNNs. However, this approach suffers from its important need of training data, which increases exponentially as the number of variables grow. This is a well-known computational problem called the curse of dimensionality [2].

In contrast to these methods, the method proposed in this paper does not assume there exists a direct relation between the trained RNN model and the observed system. Our model aims at *classifying* a series of state transition to the system that generated it, whereas each of the NN based approaches listed above aims to train a NN model that predicts the next state of the observed system.

3 Background

3.1 LFIT

The main goal of LFIT is to learn a normal logic program (NLP) describing the dynamics of the observed system. NLP is a set of rules of the form

$$A \leftarrow A_1 \wedge A_2 \cdots \wedge A_m \wedge \neg A_{m+1} \wedge \cdots \wedge \neg A_n \tag{1}$$

where A and A_i are propositional atoms, $n \geq m \geq 0$. $\neg$ and $\wedge$ are the symbols for logical negation and conjunction. For any rule R of the form 1, the atom A is called the head of R and is denoted as $h(R)$. The conjunction to the right of $\leftarrow$ is called the body of R. We represent the set of literals in the body of R as $b(R) = \{A_1, \ldots, A_m, \neg A_{m+1}, \ldots, \neg A_n\}$. The set of all propositional atoms that appear in a particular Boolean system is denoted as the Herbrand base $\mathcal{B}$.

An Herbrand interpretation I is a subset of $\mathcal{B}$. For a logic program P and an Herbrand interpretation I, the immediate consequence operator (or T_P operator) is the mapping $T_P : 2^{\mathcal{B}} \rightarrow 2^{\mathcal{B}}$:

$$T_P(I) = \{h(R) \mid R \in P, b^+(R) \subseteq I, b^-(R) \cap I = \emptyset\}. \tag{2}$$

Given a set of Herbrand interpretations E and $\{T_P(I) \mid I \in E\}$, the LFIT algorithm outputs a logic program P which completely represents the dynamics of E.

In the case of Markov(k) systems (i.e. systems with delayed effects of at most k time steps), we can define the timed Herbrand base of a logic program P, denoted by $\mathcal{B}_k$, as follows:

$$\mathcal{B}_k = \bigcup_{i=1}^{k} \{v_{t-i} \mid v \in \mathcal{B}\} \tag{3}$$

where t is a constant term which represents the current time step. Given a *Markov(k)* system S, if all rules $R \in S$ are such that $h(R) \in \mathcal{B}$ and $b(R) \in \mathcal{B}_k$, then we represent S as a logic program P with Herbrand base $\mathcal{B}_k$. A trace of execution T of S is a finite sequence of states of S. We can define T as $T = (x_0, \ldots, x_n), n \geq 1, x_i \in 2^{\mathcal{B}}$. Thus a k-step interpretation transition is (I, J) where $I \subseteq \mathcal{B}_k, J \subseteq \mathcal{B}$.

3.2 Neural Network

A multi-layer perceptron (MLP) is a type of feed-forward neural network. An MLP usually consists of one input layer, one or more hidden layer and an output layer. Each layer is fully connected, and the output layer is activated by a non-linear function. MLPs can be trained using backpropagation by gradient descent.

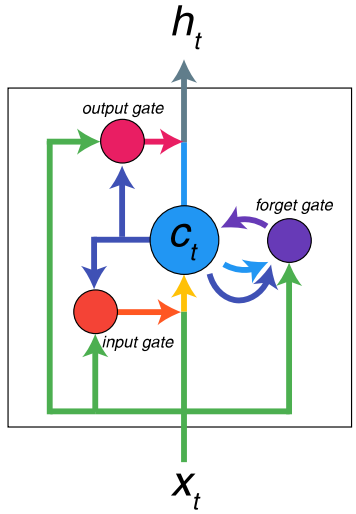

Fig. 1: An LSTM memory cell

The other neural network that we use to learn the system's dynamics is Long Short-Term Memory (LSTM) [8]. LSTM is a form of RNN that, contrary to earlier RNNs, can learn long term dependencies and do not suffer from the vanishing gradient problem. It has been popular in many sequence to sequence mapping application such as machine translation [19]. An LSTM consists of a memory cell for each time step, and each memory cell has an input gate i_t, an output gate o_t and a forget gate f_t. When a sequence of n_X time steps $X = \{x_1, x_2, \ldots, x_{n_X}\}$ is given as input, LSTM calculates the following for each time step:

$$\begin{pmatrix} i_t \\ f_t \\ o_t \\ l_t \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \cdot \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$
$$c_t = f_t \cdot c_{t-1} + i_t \cdot l_t$$
$$h_t = o_t \cdot c_t$$

where W is a weight matrix, h_t is the output of each memory cell, c_t is the hidden state of each memory cell and l_t is the input to each memory cell. σ is the sigmoid function. The input gate decides how much of the input influences the hidden state. The forget gate decides how much of the past hidden state influences the current hidden state. The output gate is responsible for deciding how much of the current hidden state influences the output. A visual illustration of a single LSTM memory cell is shown in Figure 1.

LSTM networks can be trained by performing backpropagation through time (BPTT) [7]. In BPTT, the LSTM is trained by unfolding across time steps, and then performing gradient descent to update the weights, as illustrated in Figure 2. A direct consequence of BPTT is that the LSTM can only be trained on fixed-length data. One way of overcoming this is by using truncated BPTT [20]. In truncated BPTT, the sequence is truncated into subsequences, and backpropagation is performed on the subsequences.

It can easily be seen that the connections in an LSTM model are complex, and it can be very complicated to attempt to extract or derive relations from the inner architecture of the network. Therefore we forgo the approach of extracting rules from the model, and propose a different method which instead utilizes the LSTM to classify the different inputs depending on the system that generated them.

4 Model

In this section, we propose an architecture for performing LFIT. It consists of an encoder and decoder for the state transitions, and a neural network for performing LFIT. A visualization of the architecture is shown in Figure 3 and 4. The input for the whole model is the sequence of state transitions obtained from observing the target system. The output of the model is an encoding of an approximation of the logic program representation in

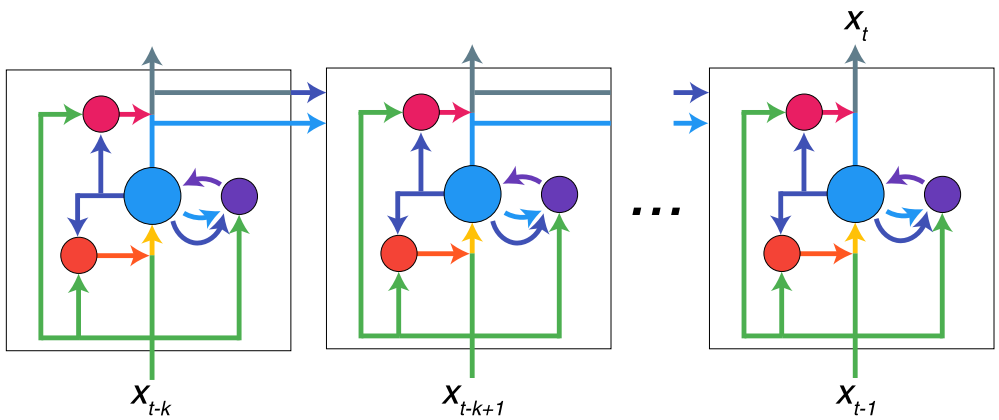

Fig. 2: Unfolding of an LSTM network for BPTT training

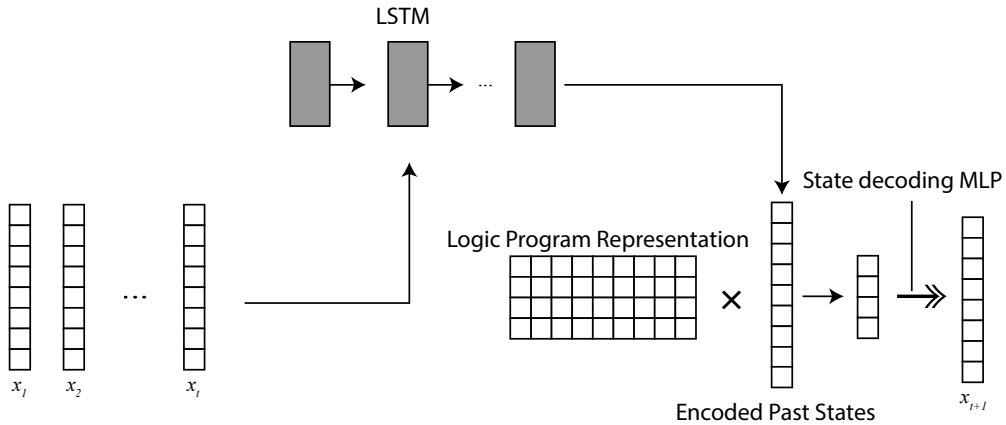

Fig. 3: A visualization of the proposed architecture, where an encoder LSTM will receive a series of state vectors and encode them into a single vector, then we will multiply the logic program representation matrix and produce a vector, which will be decoded by an MLP to produce the predicted state.

matrix form. However, we will be performing evaluation of the performance of the model based on the predicted state.

Given a series of state transitions $X_T = (x_1, x_2, \ldots, x_T)$, where $x_t \in [0, 1]$ represents the state of the system at time t, our goal is to predict x_{T+1}. Note that to be able to deal with noise and continuous values, we are not restricting the domain of x_t to $\mathbb{Z}_2$. If we obtain a representation of X_T in the form of a vector $\mathbf{x}$, we can learn a matrix $\mathbf{P}$, with which we can perform matrix multiplication as $\mathbf{Px} = x_{T+1}$. This can be thought of as performing the T_P operator in algebraic space.

The training objective function of the model is defined as:

$$\min_{\mathbf{W}} \frac{1}{n} \sum_{i=1}^{n} (x_{T+1}^{(i)} - y_{T+1}^{(i)})^2 + \lambda \|\mathbf{W}\|_2^2 \tag{4}$$

where $\mathbf{W}$ is the set of neural network weights, x_{T+1} is the prediction of the model, y_{T+1} is the truth state, $\|\mathbf{W}\|_2^2$ is the weight decay regularization [12] with hyperparameter λ.

The input state transition is fed to both the encoder and the LFIT model, as can be seen in the figure. We describe the responsibilities of the three neural network models in the following sections.

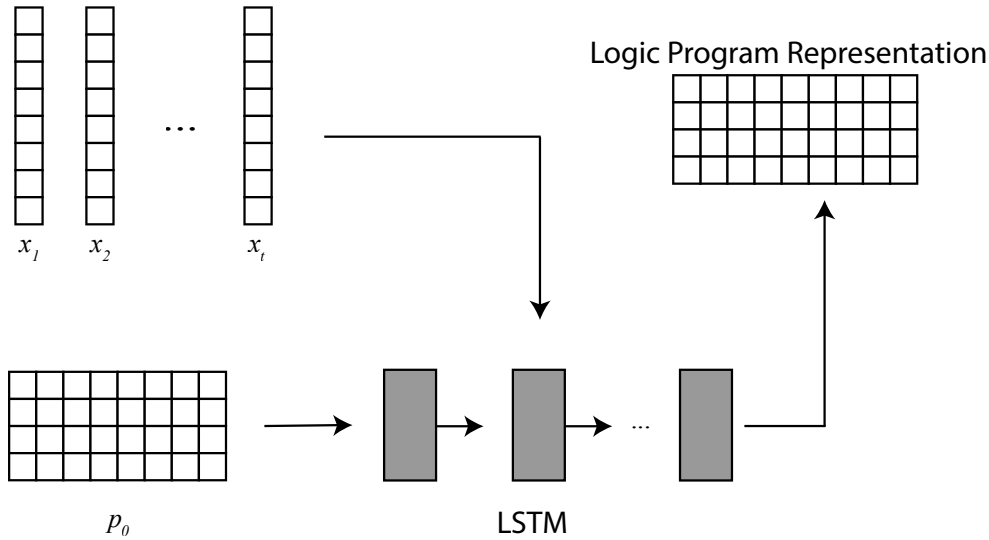

Fig. 4: A visualization of the LSTM model that is responsible for performing the LFIT. It receives as input a matrix p_0, a series of state vectors $x_1, \ldots, x_t$ and outputs a matrix.

4.1 Autoencoder

The autoencoder for the input sequences is responsible for encoding discrete time series into a feature vector that can later be manipulated by the neural network. This sequence of vectors is then encoded into one feature vector of dimension $2 \times k_a \times l_a$, where k_a denotes the number of memory cell units in the autoencoder LSTM and l_a denotes the number of LSTM layers. This amount is doubled because both c and h, which represent the state of the memory cell, are considered.

4.2 LFIT Network

This LSTM network can be thought of as performing LFIT. This network takes as input the state transitions and an initial program encoding and outputs a program encoding that is consistent with the observations, which is the same as the definition of the LFIT algorithm. Although in practice, this network is responsible for classifying the series of state transition to the corresponding logic program representation.

The produced output is the representation of the normal logic program. The observations are the same input sequence as that given to the autoencoder. The dimensions of the matrix output by this network is $(2 \times l_l \times k_l, 2 \times l_a \times k_a)$, where k_l denotes the number of memory cell units in this network and l_l denotes the number of layers.

In this work, the initial program is always set to $\emptyset$ and the LSTM network is trained to produce the complete normal logic program representation. In future work, it could be easily extended so as to accept background knowledge.

4.3 Decoder

The decoder is responsible for mapping the product of the NLP matrix and the state transition vector into a state vector that represents the predicted following state. The decoder can theoritically be any function that maps a continuous vector into a binary vector. We detail the model used in Section 4.4.

The goal of the architecture is to produce an encoding of past states, and an encoding of a normal logic program, that can then be multiplied together to predict the next state transition. This multiplication is a matrix $\times$ vector multiplication and produces a vector of $\mathbb{R}^n$ where n is the number of features in the logic program representation. This can be thought of as performing the T_p operator within linear geometric space. A MLP then decodes this vector into the desired boolean state vector.

With the encoding of the state transition and an initial program, the LFIT network learns to produce an encoded program based on the observed state transitions. This encoded program can then be used for prediction, and in future work we plan to decode it into a normal logic program thus making it possible to reason with it.

4.4 Model Details

In our experiment, the autoencoder takes a series of 10 state transitions, where each state is a 10 dimensional vector which represents the state of each variable within the system. The autoencoder LSTM model we trained has 2 layers, each with 512 memory cell units. The produced state representation is then multiplied by a $(2 \times 2 \times 512, 128)$ matrix, to produce a 128 dimension feature vector that represents the series of state transitions.

The LFIT model takes the same input as the encoder model, but the LSTM model has 4 layers, 4 being the dimension of the resulting feature vector for the predicted state, and has 1,024 hidden units which is twice the number of hidden units of the autoencoder model. The produced logic program representation is then transformed into $(4, 128)$ matrix by multiplying it with a $(2 \times 4 \times 1024, 4 \times 128)$ matrix and then reshaping.

The decoder model takes the resulting feature vector for the predicted state, which is a vector of 4 dimensions, and outputs a vector of 10 dimensions with each dimension representing the state of the variables within the system. The decoder model consists of a MLP with 1 hidden layer, and each layer has 8 hidden units. Each hidden layer is activated by ReLU (Rectified Linear Unit), which is a function that outputs 0 for all input less than 0, and is linear when the input is larger than 0. The final output layer is activated by a sigmoid function, which is defined as $\sigma(x) = 1/(1 + \exp(-x))$. The sigmoid function has a range of $[0, 1]$, which is suitable for our use where we want the MLP to output a boolean vector, with noise. The decoder model is simple, this is to avoid the decoder overfitting and thus preventing the LFIT model and the encoder model from learning.

5 Evaluation

We applied our model to learn the dynamics of Boolean networks from continuous time series. The Boolean network used in this experiment is adapted from Dubrova and Teslenko [3] and represents the cell cycle regulation of mammalians. The Boolean network is first encoded as a logic program. Each dataset represents a time series generated from an initial state vector of continuous values. The performance of the model is measured by taking the root mean-squared error (RMSE) between the predicted state and the true subsequent state. RMSE is defined as following:

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2} \tag{5}$$

where $\hat{y}_i$ denotes the predicted value and y_i is the actual value.

The initial state vector is generated by giving each of the 10 variables a random value between 0 and 1. Generated states are then mapped back to real values: 0 becomes $0.25 + \epsilon$ and 1 becomes $0.75 + \epsilon$, where $\epsilon \in (-0.25, 0.25)$, chosen randomly simulates the measurement noise.

We used the following training parameters for our experiment:

- Training steps: 10^4
- Batch size: 100
- Gradient descent optimizer: Adam, learning rate and various other parameters are left with the defaults for Tensorflow r1.2
- Dropout: probability of 0.3 per training step
- Regularization hyperparameter λ of 0.2

The model was implemented on Tensorflow r1.2 [1], and all experiments were done on Intel Xeon E5-2630 with 64 GiB of RAM and GTX 1080 Ti.

Training data is generated randomly by first randomly generating logic rules and grouping them together as NLPs. Then the initial state is set as the zero vector, and we continuously perform the T_P operator to generate all the consequent states. Variables referring to delays before the initial state is assumed to be 0. In order to assure effective training of the model, we only train on data that varies a lot. We do so by calculating the standard deviation of all states that are generated from a certain NLP, and only keeping those with standard deviation greater than or equal 0.4. We show some of the accepted NLPs in table 3.

Here, we consider two methods for training the model. One by training the model with data without noise, that is the training data is strictly $\mathbb{Z}_2$. Another way of training the model is by training on data with added noise. Each model is trained with 50 acceptable NLPs, generating 500 data points from each NLP, and training for a total of 4 hours. We evaluate each method in the following section.

5.1 Results

Dataset	RMSE (Original)	RMSE (Noisy)
1	0.27	0.28
2	0.27	0.28
3	0.26	0.26
4	0.27	0.26
5	0.27	0.28
6	0.27	0.27
7	0.27	0.28
8	0.27	0.28
9	0.27	0.27
10	0.27	0.27

Table 1: Results of the RMSE of the prediction made by the proposed model trained on *non-noisy* data on various datasets

Dataset	RMSE (Original)	RMSE (Noisy)
1	0.27	0.28
2	0.27	0.27
3	0.27	0.28
4	0.27	0.28
5	0.28	0.28
6	0.27	0.28
7	0.28	0.28
8	0.27	0.27
9	0.27	0.27
10	0.27	0.27

Table 2: Results of the RMSE of the prediction made by the proposed model trained on *noisy* data on various datasets

Table 1 shows the RMSE of the prediction made by the proposed model. Each dataset represents 50 datapoints from the same NLP, generated from different initial states. The results show that there is little difference in the accuracy between dataset with noise and without noise, which shows that the robustness of our model regarding the presence of noise in the input. In table 2, we show the performance of the model trained with data with noise. Comparing with table 1, the presence of noise in the training data doesn't affect the performance. Both models are equally robust in dealing with noise in the test data. The results obtained appear a little bit skewed due to being produced from the same system. We are planning to test the model in various other dataset when we can get access to them.

Figure 5 shows the graph of the learned representation for 8 different randomly generated NLPs based on principal component analysis (PCA). PCA is a popular technique for visualizing high dimensional embeddings [21]. As with the previous experiment, the model is fed with state transitions that were generated from the NLPs. The logic representation obtained from our model is a 4×128 matrix. We obtain the graph shown in Figure 5 by applying PCA on this matrix which extracts 3 of the dimensions that separate the data the most. Each dot in the graph is a representation learned separately from various state transitions from the logic program. Note that learned representations that are from different logic programs are clearly separated. For each dot plotted on the graph, they are actually multiple dots, representing different initial state generated from the same NLPs. The overlap for NLP 7 and NLP 8 that can be seen in the plot is due to the 2D projection of a 3D graph.

In this experiment, we observe that the model is able to identify the dynamics of the system solely based on a sequence of state transitions. We further expect that the accuracy of the predictions can be improved more by tweaking the neural network architecture.

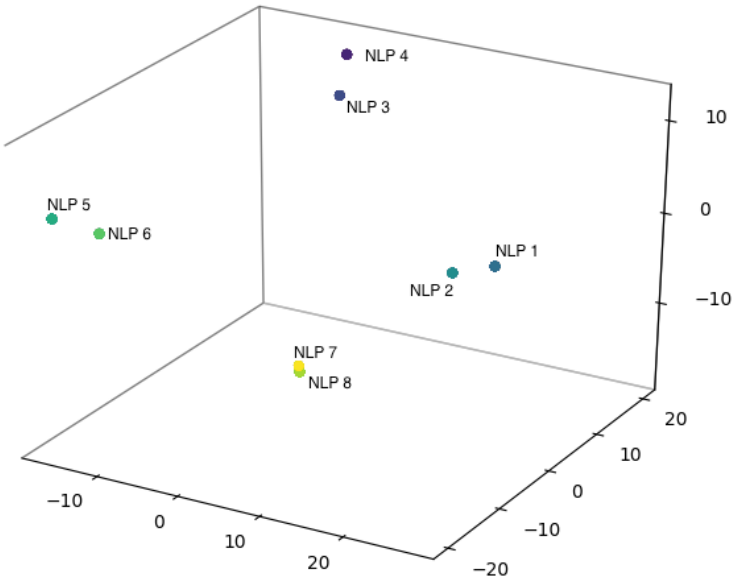

Fig. 5: PCA plot of the learned representation for NLPs based on input time series

6 Conclusion and Future Work

In this paper we propose a method for learning a matrix representation of dynamical systems with delays. One of the interesting aspects of this approach is that it produces a logic program representation in matrix form, which when multiplied with a feature vector of the past states, is able to compute a vector that represents the predicted state. This could lead to future works such as reasoning and performing induction purely in the algebraic space.

The main contribution of this work is to devise a method of modeling systems where only limited amounts of data can be collected. Without sufficient amount of data, purely logical methods cannot provide useful information, and attempts at training neural networks to model the system will result in overfitting. Therefore we speculate that generating artificial data in order to train a more generalized neural network may be a more successful approach in such cases. We also managed to show that the devised method is resilience to noise, where purely logical methods are not able to deal with.

As future work, we are planning to adapt the current method to take as input a partial program as background knowledge to the network and to decode the NLP representation into logical form to allow humans to reason with. We also hope to evaluate the predictions made by this model with other similar models.

References

1. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., Zheng, X.: TensorFlow: Large-scale machine learning on heterogeneous systems (2015), https://www.tensorflow.org/, software available from tensorflow.org
2. Donoho, D.L.: High-dimensional data analysis: The curses and blessings of dimensionality. In: AMS Math Challenges Lecture. p. 132 (2000)
3. 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), 1393–1399 (2011)
4. d'Avila Garcez, A.S., Broda, K., Gabbay, D.M.: Symbolic knowledge extraction from trained neural networks: A sound approach. Artificial Intelligence 125(1), 155–207 (2001)
5. d'Avila Garcez, A.S., Zaverucha, G.: The connectionist inductive learning and logic programming system. Applied Intelligence 11(1), 59–77 (1999)
6. Gentet, E., Tourret, S., Inoue, K.: Learning from interpretation transition using feed-forward neural network. In: Proceedings of ILP 2016, CEUR Proc. 1865. pp. 27–33 (2016)
7. Graves, A., Schmidhuber, J.: Framewise phoneme classification with bidirectional lstm and other neural network architectures. Neural Networks 18(5), 602–610 (2005)

8. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural computation 9(8), 1735–1780 (1997)
9. Inoue, K., Ribeiro, T., Sakama, C.: Learning from interpretation transition. Machine Learning 94(1), 51–79 (2014)
10. Jennings, N.R., Sycara, K., Wooldridge, M.: A roadmap of agent research and development. Autonomous Agents and Multi-Agent Systems 1(1), 7–38 (Jan 1998), https://doi.org/10.1023/A:1010090405266
11. Khan, A., Mandal, S., Pal, R.K., Saha, G.: Construction of gene regulatory networks using recurrent neural networks and swarm intelligence. Scientifica 2016 (2016)
12. Krogh, A., Hertz, J.A.: A simple weight decay can improve generalization. In: Proceedings of the 4th International Conference on Neural Information Processing Systems. pp. 950–957. NIPS'91, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1991), http://dl.acm.org/citation.cfm?id=2986916.2987033
13. Martínez, D., Alenyà, G., Ribeiro, T., Inoue, K., Torras, C.: Relational reinforcement learning for planning with exogenous effects. Journal of Machine Learning Research 18(78), 1–44 (2017), http://jmlr.org/papers/v18/16-326.html
14. Ribeiro, T., Inoue, K.: Learning prime implicant conditions from interpretation transition. In: ILP 2015, pp. 108–125. Springer (2015)
15. Ribeiro, T., Inoue, K., Sakama, C.: A BDD-based algorithm for learning from interpretation transition. In: Proc. ILP 2013, LNAI 8812. pp. 47–63. Springer (2014)
16. Ribeiro, T., Magnin, M., Inoue, K., Sakama, C.: Learning delayed influences of biological systems. Frontiers in bioengineering and biotechnology 2 (2014)
17. Ribeiro, T., Magnin, M., Inoue, K., Sakama, C.: Learning multi-valued biological models with delayed influence from time-series observations. In: 14th IEEE International Conference on Machine Learning and Applications, ICMLA 2015, Miami, FL, USA, December 9-11, 2015. pp. 25–31 (2015)
18. Ribeiro, T., Tourret, S., Folschette, M., Magnin, M., Borzacchiello, D., Chinesta, F., Roux, O., Inoue, K.: Inductive learning from state transitions over continuous domains. In: Proceedings of ILP 2017, to appear. Springer (2017)
19. Sutskever, I., Vinyals, O., Le, Q.V.: Sequence to sequence learning with neural networks. In: Advances in neural information processing systems. pp. 3104–3112 (2014)
20. Williams, R.J., Peng, J.: An efficient gradient-based algorithm for on-line training of recurrent network trajectories. Neural Computation 2, 490–501 (1990)
21. Wold, S., Esbensen, K., Geladi, P.: Principal component analysis. Chemometrics and Intelligent Laboratory Systems 2(1), 37 – 52 (1987), http://www.sciencedirect.com/science/article/pii/0169743987800849, proceedings of the Multivariate Statistical Workshop for Geologists and Geochemists

$a_t \leftarrow f_{t-5} \wedge \neg d_{t-4} \wedge \neg i_{t-1} \wedge \neg g_{t-1} \wedge \neg g_{t-4} \wedge \neg d_{t-1}$
$b_t \leftarrow \neg d_{t-1} \wedge \neg d_{t-5}$
$c_t \leftarrow \neg b_{t-1}$
$d_t \leftarrow \neg c_{t-1} \wedge \neg i_{t-5} \wedge \neg f_{t-3} \wedge \neg c_{t-2} \wedge \neg i_{t-1} \wedge \neg h_{t-1} \wedge \neg a_{t-1} \wedge \neg d_{t-3} \wedge$
$\neg d_{t-5}$
$e_t \leftarrow e_{t-2} \wedge \neg e_{t-1} \wedge \neg a_{t-3} \wedge \neg f_{t-4} \wedge \neg j_{t-5}$
$f_t \leftarrow b_{t-2} \wedge g_{t-1} \wedge h_{t-5} \wedge \neg i_{t-2} \wedge \neg f_{t-2}$
$g_t \leftarrow \neg d_{t-1} \wedge \neg g_{t-1}$
$h_t \leftarrow \neg i_{t-1}$
$i_t \leftarrow \neg e_{t-4} \wedge \neg j_{t-1} \wedge \neg d_{t-2} \wedge \neg g_{t-5} \wedge \neg c_{t-2} \wedge \neg i_{t-5} \wedge \neg g_{t-3} \wedge \neg j_{t-2} \wedge \neg i_{t-1}$
$j_t \leftarrow b_{t-3} \wedge c_{t-4} \wedge \neg j_{t-2} \wedge \neg c_{t-3}$

$a_t \leftarrow b_{t-3} \wedge g_{t-2} \wedge f_{t-4} \wedge j_{t-4} \wedge \neg c_{t-5} \wedge \neg e_{t-2} \wedge \neg a_{t-4} \wedge \neg h_{t-3} \wedge \neg i_{t-3} \wedge$
$\neg h_{t-1} \wedge \neg e_{t-3} \wedge \neg c_{t-1} \wedge \neg c_{t-2} \wedge \neg a_{t-5}$
$b_t \leftarrow \neg e_{t-4} \wedge \neg c_{t-3} \wedge \neg i_{t-3} \wedge \neg f_{t-3} \wedge \neg b_{t-2} \wedge \neg i_{t-5} \wedge \neg i_{t-3} \wedge$
$\neg a_{t-5} \wedge \neg f_{t-5}$
$c_t \leftarrow \neg c_{t-3} \wedge \neg b_{t-1} \wedge \neg c_{t-5} \wedge \neg j_{t-2} \wedge \neg b_{t-5} \wedge \neg i_{t-2} \wedge \neg a_{t-5} \wedge \neg b_{t-3}$
$d_t \leftarrow \neg e_{t-4} \wedge \neg a_{t-5} \wedge \neg e_{t-4}$
$e_t \leftarrow \neg g_{t-1}$
$f_t \leftarrow h_{t-1} \wedge e_{t-3} \wedge c_{t-3} \wedge \neg a_{t-2} \wedge \neg g_{t-4}$
$g_t \leftarrow \neg f_{t-5}$
$h_t \leftarrow \neg e_{t-5}$
$i_t \leftarrow \neg j_{t-3} \wedge \neg a_{t-5} \wedge \neg i_{t-4}$
$j_t \leftarrow \neg g_{t-5} \wedge \neg e_{t-5} \wedge \neg d_{t-1}$

$a_t \leftarrow \neg e_{t-2}$
$b_t \leftarrow \neg b_{t-4}$
$c_t \leftarrow j_{t-1} \wedge f_{t-1} \wedge f_{t-2} \wedge d_{t-1} \wedge h_{t-5} \wedge \neg g_{t-3} \wedge \neg c_{t-5}$
$d_t \leftarrow \neg g_{t-3} \wedge \neg b_{t-5} \wedge \neg c_{t-3} \wedge \neg b_{t-5} \wedge \neg j_{t-3} \wedge \neg h_{t-2} \wedge \neg f_{t-5} \wedge \neg d_{t-2} \wedge$
$\neg c_{t-5}$
$e_t \leftarrow g_{t-2} \wedge g_{t-4} \wedge f_{t-5} \wedge j_{t-3} \wedge e_{t-1} \wedge \neg j_{t-1} \wedge$
$\neg a_{t-1} \wedge \neg f_{t-1} \wedge \neg e_{t-4}$
$f_t \leftarrow f_{t-5} \wedge b_{t-5} \wedge g_{t-5} \wedge \neg j_{t-2} \wedge \neg c_{t-5} \wedge \neg i_{t-5} \wedge \neg g_{t-4} \wedge \neg g_{t-5} \wedge$
$\neg f_{t-2} \wedge \neg f_{t-3} \wedge \neg h_{t-4}$
$g_t \leftarrow a_{t-2} \wedge d_{t-3} \wedge \neg g_{t-2} \wedge \neg c_{t-3}$
$h_t \leftarrow \neg j_{t-5} \wedge \neg e_{t-4} \wedge \neg g_{t-5} \wedge \neg f_{t-1}$
$i_t \leftarrow \neg e_{t-4}$
$j_t \leftarrow \neg i_{t-5}$

Table 3: Example NLPs that are randomly generated and used for training