Learning Logic Program Representation from Delayed Interpretation Transition Using Recurrent Neural Networks

Yin Jun Phua
Tokyo Institute of Technology

Sophie Tourret
National Institute of Informatics

Katsumi Inoue
Tokyo Institute of Technology
National Institute of Informatics

Abstract

Having a method to understand the interactions and delayed influences between components of dynamical systems can provide useful applications to biological and other dynamical systems. In this paper, we present a method relying on Recurrent Neural Networks (RNN) that can learn to distinguish the nature of different systems. This method utilizes Long Short-Term Memory (LSTM) to extract and encode features from the input sequence of time series data. We also show that the produced high dimensional encoding can be used to distinguish time series that originate from different dynamical systems.

Introduction

Learning from Interpretation Transition (LFIT) is an unsupervised learning algorithm which learns the dynamics of an environment just by observing state transitions. Applications for such learning algorithms can range from multi-agent systems, where learning other agents' behavior can be crucial for decision making, to systems biology, where knowing the interaction between genes can greatly help in the creation of drugs to treat sickness. This paper introduces an algorithm utilizing Recurrent Neural Network (RNN) to perform LFIT. The proposed approach outputs a high dimensional matrix representation of the logic program that describes the dynamics of a Boolean system. In this paper, we show that the learned matrix representation is equivalent to the Normal Logic Program (NLP) that can be used to describe these dynamics. This approach extends the approach described in the paper (Gentet, Tourret, and Inoue 2016), which uses a feed-forward neural network to learn 1 step transitions, by constructing an NLP from state transitions in a delayed environment. Neural networks are known to perform well in tasks like function approximation and prediction. By utilizing neural networks, we hope to be able to perform LFIT on data with noises and continuous data, where traditional approaches cannot be applied (Inoue, Ribeiro, and Sakama 2014). Previously, application of neural networks in inductive logic programming involves training the neural network to model the dynamics of the Boolean system. The approach proposed in this paper differs in that the neural network is not trained to model the dynamical system, but rather to output a representation of the system.

To the best of our knowledge, there is only one available article (Khan et. al 2016) about constructing models of dynamical systems using RNNs. However this approach suffers from its important need of training data, that increases exponentially as the number of variables grow. This is a well-known computational problem called the curse of dimensionality (Donoho 2000). In most practical cases, especially in biological systems, sufficient training data cannot be obtained to rely on this method. Thus, having a method that achieves high performance with a small amount of training data is of great importance.

The rest of the paper is organized as follows. We will first cover the basics of LFIT, which is the framework that this work is based on. Next we will explain some basic knowledge of recurrent neural network that is required to understand this paper. We then describe the method used in (Khan et. al 2016). Then we will explain our approach of RNN-LFIT. We explain the experiments done and show the experimental results in the next section. Then finally, we will dicuss the results observed and conclude the paper.

LFIT

A *normal logic program* (NLP) is a set of *rules* of the form

$$A \leftarrow A_1 \wedge \cdots \wedge A_m \wedge \neg A_{m+1} \wedge \cdots \wedge \neg A_n \qquad (1)$$

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 $b(R) = \{A_1, \ldots, A_m, \neg A_{m+1}, \ldots, \neg A_n\}$. Atoms that appear positively in $b(R)$ are denoted as $b^+(R) = \{A_1, \ldots, A_m\}$, while those that appear negatively are denoted as $b^-(R) = \{A_{m+1}, \ldots, A_n\}$. 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 $T_P : 2^{\mathcal{B}} \rightarrow 2^{\mathcal{B}}$:

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

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. When an LFIT algorithm only considers 1 step transitions, it is also called LF1T (pronounced "L-F-one-T"). There has been 2 main logical method algorithms developed for LF1T (Inoue, Ribeiro, and Sakama 2014), (Inoue, Ribeiro 2015). The first algorithm relies on a generalization scheme to simplify the created rules with each new transition observed. The second algorithm acts in the opposite way, by specializing the rules to cover each new transition.

LFkT

LFkT is the extension of LF1T to *Markov(k) systems*, to learn from k-step transitions. Given a number of time steps k, the timed Herbrand base of a logic program P, denoted by $\mathcal{B}_k$, is as follows:

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

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 $I \subseteq \mathcal{B}_k, J \subseteq \mathcal{B}$. An algorithm for LFkT is described in (Ribeiro et. al 2014).

Previous approaches for LFIT that do not rely on neural networks require the data to be discretized before learning can be done. If the discretization step is wrong, every step that follows will also be wrong. Therefore an approach where discretization is not needed is very appealing. In addition, with the notable exception of NN-LFIT (Gentet, Tourret, and Inoue 2016), the LFIT family of algorithms do not generalize to transitions that did not appear within the training data.

RNN

RNNs are an extension of feed-forward neural networks that deal with sequence to sequence mapping (Sutskever, Vinyals, and Le 2014). Given a sequence $(x_1, \ldots, x_M)$, using 3 weight matrices W^{hx}, W^{hh}, W^{yh} and 3 bias vectors b_h, b_y, h_0, a standard RNN calculates the following:

$$h_t = \sigma(W^{hx}x_t + W^{hh}h_{t-1} + b_h)$$
$$y_t = W^{yh}h_t + b_y$$

and outputs a sequence $(y_1, \ldots, y_T)$, where M may differ from T. The vector h_t represents the hidden state for each time step, and σ is the sigmoid function, which is defined as $\sigma(x) = 1/(1 + \exp(-x))$.

However, standard RNNs have problems learning long term dependencies due to their nonlinear iterative nature (Bengio, Simard, and Frasconi 1994). RNNs also suffer from the vanishing gradient problem (Bengio, Simard, and Frasconi 1994). To mitigate these problems, an RNN architecture called Long Short-Term Memory (LSTM) was introduced. LSTM has been popular in many sequence to sequence mapping application such as machine translation (Sutskever, Vinyals, and Le 2014). In this paper, we apply LSTM to LFkT, benefitting from its generalization power and natural handling of delays.

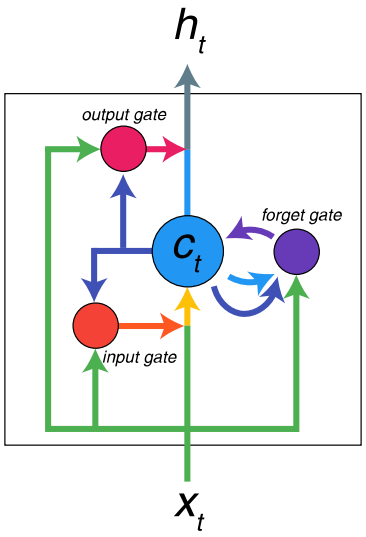


Figure 1: An illustration of an LSTM memory cell

Our approach uses the LSTM model described in (Hochreiter, Schmidhuber 1997). 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 the 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. The input gate decides how "much" of the input show influence the hidden state. The forget gate decides how "much" of the past hidden state should influence the current hidden state. The output gate is responsbile for deciding how "much" of the current hidden state to let through to the output. A visual illustration of a single LSTM memory cell is shown at Figure 1.

LSTM networks can be trained by using gradient descent, by using backpropagation through time (BPTT) (Graves, Schmidhuber 2005). During BPTT, the LSTM is unfolded across time steps. Thus the length of the input sequence is fixed. Variable sequence length mapping can be achieved by padding zeros, however that is out of the scope of this paper.

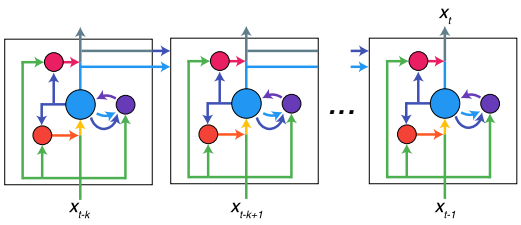


Figure 2: BPTT training for LSTM network

Related Work

In (Khan et. al 2016), the authors proposed a method utilizing RNNs to extract the topology of a gene regulatory network (GRN). A GRN represents the complex interregulatory relationships among genes, and is said to be critical to understanding diseases and creating drugs to cure them. GRNs have also been successfully modelled with Boolean networks (Albert, Othmer 2003).

The proposed method in (Khan et. al 2016) involves first constructing the neural network based on background knowledge. Once the structure of the network is finalized, it is trained on the available data to predict the genetic expressions. The results in their paper shows that, while the method achieved very high accuracy in predicting the topology for small artificial GRNs compared to other available methods, their method however was not able to achieve similar level of performance on larger GRNs.

Proposed Approach

Our proposed approach uses 3 LSTM networks for training. We call these networks the encoder, the decoder and the future predictor. The encoder network is responsible for converting the input sequence data into the logic program representation. The decoder network outputs the input sequence data, given the initial time step of the sequence and the logic program representation. The future predictor network predicts the time step following the final time step x_M of the input sequence X, given the logic program representation and x_M.

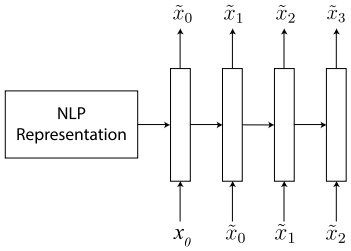
Here we consider a dynamical system S as a vector x_t of Boolean variables $x_t^{(1)}, x_t^{(2)}, \ldots, x_t^{(|\mathcal{B}|)}$ evolving through time and the time series data X with M time steps is the sequence of vectors $(x_1, x_2, \ldots, x_M)$. When RNN-LFIT is used, the input length must be larger than the maximum delay k within the system.

In this context, the encoder LSTM computes the function:

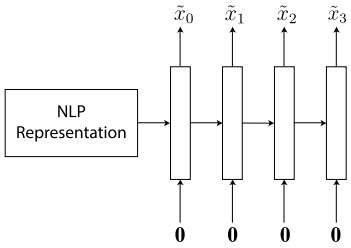
$$T = \text{EncoderRNN}(x_0, x_1, \ldots, x_M)$$

assuming the LSTM has n hidden nodes and m layers, the output T is an $m \times n$ matrix.

To restore the original input sequence, we can apply the T_P operator on the initial time step M times. This operator is approximated by the decoder, which takes the encoding T as an input in addition to the initial state x_0. Formally, it computes $\tilde{X}$:



(a) Conditional Structure



(b) Uncondintional Structure

Figure 3: The structure of conditional and unconditional decoder

$$\tilde{X} = \text{DecoderRNN}(x_0, T)$$

where $\tilde{X} = (\tilde{x}_0, \tilde{x}_1, \ldots, \tilde{x}_M)$. The output of the LSTM network in the decoder is passed through a sigmoid function to limit the range to $[0, 1]$. We can consider both a conditional structure and unconditional structure for the decoder network. A conditional structure is a structure where the LSTM is fed as input at each time step the result of the previous time step as illustrated on Figure 3a. An unconditional structure is a structure where the LSTM is only fed the encoding of the logic program and relies on the evolution of its inner state to generate the outputs at each time step, as illustrated in Figure 3b.

Finally, the future predictor produces the future time steps $y_{M+1}, y_{M+2}, \ldots, y_N$ given the final state x_M as an input. The future predictor model can be thought as computing the following:

$$Y = \text{PredictorRNN}(x_M, T)$$

where $Y = (y_{M+1}, y_{M+2}, \ldots, y_N)$. As with the decoder, the future predictor can be used in a conditional and an unconditional way.

RNN-LFIT is the combination of the encoder, decoder and the future predictor. The structure of RNN-LFIT is shown in Figure 4. Depending on whether the decoder and the future predictor are both conditioned or are both unconditioned, RNN-LFIT is also referred to as conditioned or unconditioned.

Experiments

The dataset used to train the model is randomly generated. To generate the data, we first decide the number of variables, the minimum and maximum number of literals in each rule

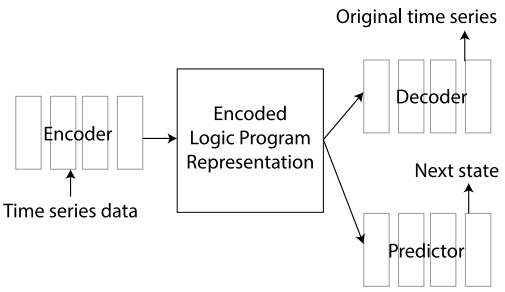


Figure 4: Architecture of RNN-LFIT

and the maximum delay in the system, then randomly generate n NLPs. We have listed some of the generated NLPs in the appendix. Next we randomly set an initial state and perform the T_P operator N times to get a sequence of data. Thus the first M data points are used as an input to the encoder network, and then the next $N-M$ data points are used to check the ouput of the future predictor model.

We train the encoder to produce a representation, which the decoder can then use to reproduce the original input, and which also allows the future predictor to predict the next states. The objective function is written as follows:

$$1/|\mathcal{S}| \sum_{(Y,X)\in\mathcal{S}} \log p(Y, \tilde{X}|X)$$

where we want to maximize the log probability of a correct prediction Y and a correct reconstruction of the original input sequence $\tilde{X}$ when given the input sequence X. $\mathcal{S}$ denotes the training set.

While training, we found that deep LSTM perform better than shallow ones. Deeper LSTM models are said to have better performance because they have a higher degree of expressivity (Sutskever, Vinyals, and Le 2014), therefore in this experiment, we used a 5 layer LSTM model. Listed below are additional details regarding the training of RNN-LFIT:

- the parameters of all the networks are initialized using a normal distribution of mean 0 and variance 1,
- the learning is performed using Adam stochastic optimization (Kingma, Ba 2014),
- the learning batches contain 64 sequences each.

In all the experiments, we evaluate the accuracy of the predictor. To do so, we convert the predictor output y_t back to a propositional vector r_t in the following way:

$$\forall i, r_t^{(i)} = \begin{cases} 1, & y_t^{(i)} \geq \alpha, \\ 0, & else \end{cases}. \qquad (2)$$

where α is a chosen threshold in $[0, 1]$.

In order to evaluate different methods and experiment results, we use a metric called Accuracy (ACC). Accuracy signifies the fraction of the correct prediction made by the network across all the predictions. The formula is written as follows:

$$\text{ACC} = \frac{\text{Correct Predictions}}{\text{All Predictions}}$$

We first evaluate the performance of the model with 14 variables and a maximum delay of 5. Then we increase the maximum delay to find if it is correlated with the accuracy of RNN-LFIT. Finally, we increase the number of variables and measure the impact of this change on the accuracy of RNN-LFIT.

All experiments and training are performed with code written using Tensorflow (Tensorflow 2017). The actual training is done on CUDA 7.5 enabled Tensorflow v0.12.0, running on Python 2.7. The OS used was Fedora 24, running on Core i5 6600K, 32GB of memory. The GPU that was used is a GTX 980. Most of the training took 4-6 hours, but once the network has been trained, it can then be used on as many time series data as wanted without any further training.

RNN-LFIT Structure	Accuracy
Unconditional without decoder	0.80
Unconditional, full	0.84
Conditional, full	0.82

Table 1: Accuracy of the RNN-LFIT variants

Maximum delay k	Length of input sequence	Accuracy
6	8	0.83
7	8	0.87
8	10	0.85
9	10	0.80
10	12	0.83

Table 2: Unconditional RNN-LFIT with varying maximum delays

Experimental Results

The results of the first experiment, with 4 variables and a maximal delay of 5, are shown in table 1. Each LSTM network has 6 memory cells and 5 layers. They were all trained and tested on the same sets of data. The training set is made up of 8,192 different sequences generated with 4 different NLPs. For evaluation, α was set to 0.3, and for testing we used 8,192 new sequences generated from 4 NLPs different from the ones used to generate the training set. From the results, we can observe that without the decoder, accuracy is a little lower, and unconditional RNN-LFIT perform better than conditional ones.

In the second experiment, we increase the maximum delay of the NLPs used to generate the dataset, while the number of variables remains unchanged. As seen in table 2, no significant drop in accuracy is observed when the maximum delay is increased.

In previous works (Ribeiro et. al 2014), increasing the number of variables caused the number of parameters within the network to increase, thus more training data was needed to maintain the accuracy. However the main application in

Number of variables $\|\mathcal{B}\|$	Accuracy
15	0.85
20	0.84
25	0.85
30	0.84

Table 3: Unconditional RNN-LFIT with varying number of variables

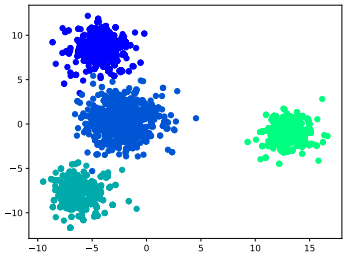


Figure 5: This figure shows a 2-dimensional LDA projection of the learned NLP representation after processing the input sequence. The difference in shades represents sequences generated by different NLPs. Note that different NLPS are neatly clustered.

(Ribeiro et. al 2014), i.e. modelling gene regulatory networks, assumes most of the time that the training data is limited. Thus to compare our method with previous methods, we train RNN-LFIT with a larger number of variables. Results in table 3 shows that our method displays no degradation in performance even with increased variables. All experiments on the varying number of variables are performed on datasets which have a maximum delay of $k = 5$, and 8,192 sequences generated from 4 randomly generated NLPs.

Discussion

Figure 5 shows a 2-dimensional linear discriminant analysis (LDA) projection (McLachlan 2004) of the tested NLP representations. Each point in the graph represents the output produced by the encoder after reading the input sequence. The different shades represent data sequences generated by different NLPs. There are 4 NLPs generated in this figure, and the different NLPs are neatly clustered, suggesting it should be possible to recover an NLP describing the time series by mapping the output space using artifically made data. The 4 NLPs that correspond to the different shades in the chart is listed in appendix .

Conclusion

In this paper, we presented RNN-LFIT, an LSTM based approach for learning to distinguish time series data according to the rules governing them. The experimental results show that we succeed in capturing features unique to the reules behind each dataset.

Contrary to previous approaches, RNN-LFIT does not learn to model the dynamic of the system that is inputted. Instead it learns characteristics that uniquely identify this system. A strong point of this approach is that it can be trained with as many artificially generated time series as needed, ensuring accurate results even when the real data available is very sparse.

Obtaining a representation that captures the characteristics and features of the time series data is a big step towards understanding the dynamics of a system. In future works, we are planning to convert the representation into NLPs to make it understandable by humans and study the behavior of RNN-LFIT on datasets that are continuous and noisy.

Appendices

Generated NLPs

Table 4 shows the 4 NLPs generated that are used to test the networks. The generation setting used is 14 variables, maximum delay of 5 and maximum literals of 2 for each rule.

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

Table 4: Generated NLPs

References

Enguerrand Gentet, Sophie Tourret, and Katsumi Inoue, 2016. Learning from Interpretation Transition using Feed-Forward Neural Network. *Inductive Logic Programming*.

Katsumi Inoue, Tony Ribeiro, and Chiaki Sakama, 2014. A BDD-Based Algorithm for Learning from Interpretation Transition. Inductive Logic Programming: Revised Selected Papers from the 23rd International Conference, Vol. 8812, pp. 4763.

Katsumi Inoue and Tony Ribeiro, 2015. Learning Prime Implicant Conditions from Interpretaion Transition. Inductive Logic Programming: Revised Selected Papers from the 24th International Conference, Vol. 9046, pp. 108125.

Tony Ribeiro, Morgan Magnin, Katsumi Inoue, and Chiaki Sakama, 2014. Learning delayed influences of biological systems. *Frontiers in Bioengineering and Biotechnology*, Vol. 2, p. 81.

Abhinandan Khan, Sudip Mandal, Rajat Kumar Pal, and Goutam Saha, 2016. Construction of gene regulatory networks using recurrent neural networks and swarm intelligence. *Scientifica*, Vol. 2016.

Y. Bengio, P. Simard, and P. Frasconi, 1994. Learning long-term dependencies with gradient descent is difficult. *Trans. Neur. Netw.*, Vol. 5, No. 2, pp. 157166.

Sepp Hochreiter and Jurgen Schmidhuber, 1997. Long short-term memory. *Neural Comput.*, Vol. 9, No. 8, pp. 17351780, November 1997.

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le, 2014. Sequence to sequence learning with neural networks. *CoRR*, Vol. abs/1409.3215.

Nitish Srivastava, Elman Mansimov, and Ruslan Salakhudinov, 2015. Unsupervised learning of video representations using lstms. In David Blei and Francis Bach, editors, *Proceedings of the 32nd International Conference on Machine Learning* (ICML-15), pp. 843852. JMLR Workshop and Conference Proceedings.

Diederik P. Kingma and Jimmy Ba, 2014. Adam: A method for stochastic optimization. *CoRR*, Vol. abs/1412.6980.

McLachlan, G. J, 2004. Discriminant Analysis and Statistical Pattern Recognition. *Wiley Interscience.*.

Tensorflow. https://www.tensorflow.org/, 2017.

Alex Graves and Jurgen Schmidhuber, 2005. Framewise phoneme classification with bidirectional LSTM and other neural network architectures. *Neural Networks*, 18(56):602610, July 2005.

R. Albert, HG. Othmer, 2003. The topology of the regulatory interactions predicts the expression pattern of the segment polarity genes in Drosophila melanogaster. *J. Theor. Biol*.

D. L. Donoho, 2000. High-dimensional data analysis: the curses and blessings of dimensionality. *AMS Math Challenges Lecture*, pp. 132.