Learning Logic Programs Using Neural Networks by Exploiting Symbolic Invariance

Yin Jun Phua[1,2] and Katsumi Inoue[1,2]

[1] The Graduate University for Advanced Studies, SOKENDAI, Tokyo, Japan
[2] National Institute of Informatics, Tokyo, Japan {phuayj,inoue}@nii.ac.jp

Abstract. Learning from Interpretation Transition (LFIT) is an unsupervised learning algorithm which learns the dynamics just by observing state transitions. LFIT algorithms have mainly been implemented in the symbolic method, but they are not robust to noisy or missing data. Recently, research works combining logical operations with neural networks are receiving a lot of attention, with most works taking an extraction based approach where a single neural network model is trained to solve the problem, followed by extracting a logic model from the neural network model. However most research work suffer from the combinatorial explosion problem when trying to scale up to solve larger problems. In particular a lot of the invariance that hold in the symbolic world are not getting utilized in the neural network field. In this work, we present a model that exploits symbolic invariance in our problem. We show that our model is able to scale up to larger tasks than previous work.

Keywords: Neural network · LFIT · Interpretability · Neural-Symbolic AI.

1 Introduction

We interact with various different complex dynamic systems in our everyday lives. These systems with various components evolve over time in a manner that we can observe. Understanding the influences between these components within these dynamic systems provide valuable insights. With the knowledge of the influences and relationships between the components, it becomes possible to manipulate the dynamic system for a desired outcome or to plan for certain events. In most real world systems however, we do not have direct access to the underlying rules that govern them. Most of the time though, we are able to obtain observations of the systems. Learning from Interpretation Transition (LFIT) [12] is an unsupervised learning algorithm which learns the dynamics just by observing state transitions. Under the LFIT framework, the dynamics of the system is represented as normal logic programs (NLP). LFIT can be applied to multi-agent systems, where learning other agents' behavior can be crucial for decision making, or even to biological systems [19], where knowing the interaction between genes can lead to huge breakthrough in developing drugs to cure illnesses.

LFIT algorithms have mainly been implemented in two different methods, the symbolic method and the neural network method. The symbolic method utilizes logical operations to learn and induce logic programs [18]. This family of algorithms have been well-developed and has the nice property of being interpretable. It is also known to scale well to larger systems. However, it is not without its downsides. For example, it only learns what it can see, i.e. the prediction for data not in the training set will always be incorrect. In real world applications this is a particularly huge problem because we cannot always obtain perfect observations of a system. Another shortcoming is that logical operations employed by this family of algorithms lack ambiguous notations. This means that any error or noise present within the data will be reflected directly in the learned model.

In recent years, research works combining logical operations with neural networks have gained a lot of attention [3]. Whilst the motivation for most of these research lie in the interpretability of neural networks, the ability to introduce ambiguity to the input data is also perceived as one of their strengths. As such, most of the work has been focused on inspecting trainable parameters and extracting rules from it. In a similar spirit applying LFIT to neural networks, there is NN-LFIT [9], where a neural network is trained to model the system and a logic program is extracted based on the weights of the trained neural network. NN-LFIT has been demonstrated to be able to generalize from small amount of training data, and it is also robust against noisy or erroneous data. There is also an extension of this work by [20] where instead of brute forcing all values to construct a truth table, the authors used heuristics to derive the logical rules from the trained neural network.

One of the big problems that leads to the scalability problem in methods combining neural networks with symbolic is the combinatorial explosion problem. While symbolic methods also suffer from the combinatorial explosion problem, the application of neural networks towards symbolic problems have multiple different ways of blowing up combinatorially. In δILP [6], the scalability problem is so severe that each predicate has to be limited to only 2 clauses whereas symbolic methods have no such limits. In δLFIT, the number of variables were limited to 5 variables due to memory constraint, where real world application often require thousands of variables. We observe that several invariances that hold in symbolic space are not considered in neural network methods. We refer to these invariances as *symbolic invariance*. One such invariance is the order of inputs. In LFIT, inputs are represented as a set of state transitions. The ordering of the states within each transition are significant given the ordering conveys the meaning of time, but the ordering for the transitions as an input doesn't matter to the algorithm. Whereas in neural network methods, different ordering of the transitions are presented as different inputs, and thus all permutation of the input transitions are treated as different points in the learning space.

This data inefficiency combined with memory space explosion, seems to lead to most work in the neural symbolic field to be currently impractical for real world application. We speculate that being much more data efficient and reducing memory usage could lead to a much more practical application. In this work we hope to demonstrate a method to exploit this invariance, although it is not a generally applicable method, we hope that this will inspire other works as well.

In this paper, we propose the δLFIT+ model which aim to solve the scalability problem, building on top of δLFIT. This paper is structured as follows, we will first cover some necessary background on LFIT and some preliminary for δLFIT in Section 2. Next we will present our methods of δLFIT+ in Section 3. Following that, we will show our experimental results in Section 4. In Section 5, we will discuss some of our observations. Next, we will describe some related works in Section 6. Finally, we will be summarizing our work and discussing some further research that are possible in Section 7.

2 Background

2.1 LFIT

The LFIT algorithm strives to learn an NLP that explains the observation obtained from a dynamic system. The dynamics of a changing system with respect to time can be represented by introducing time as an argument. Therefore we can consider the state of an atom A at time t as $A(t)$. A dynamic rule can then be described as follows:

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

this rule means that, if all of $A_1, A_2, \ldots, A_m$ is true at time t, and all of A_{m+1}, $A_{m+2}, \ldots, A_n$ is false at time t, then the head A will be true at time $t+1$. $A_1, A_2, \ldots, A_m$ can be denoted as b^+ representing the positive literals that appear in the rule while $A_{m+1}, A_{m+2}, \ldots, A_n$ can be denoted as b^- representing the negative literals that appear in the rule. However, noting that even though rule (1) has time arguments in each of the atoms, t and $t+1$ only appear in the right hand side and left handside respectively, and thus can be omitted [11]. Therefore rule (1) can be equally expressed as the following propositional rule:

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

A can be any one of $A_1, \ldots, A_n$ and this will not be considered a cyclic rule, due to the implicit time parameter they will be considered as separate literals. However, in the paper, we would like to treat A on the left and on the right equally, therefore we will be referring to A and any other literals as a variable, which is different from the propositional variable.

Given a set P of such propositional rules, we can simulate the state transition of a dynamical system with the T_P operator.

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

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

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.

The LFIT algorithm, in its simplest form, can be described as an algorithm that requires an input of a set of state transitions $S = \{(I, T_P(I)) \mid I \in E\}$ and an initial NLP P_0, then outputs an NLP P such that P is consistent with the input S.

2.2 Rule Classification

To build a classifier, all the possible rules that a system can have given the Herbrand base $\mathcal{B}$ have to be known and enumerated. If there are no restrictions, the number of rules that are valid is infinite. Therefore some restrictions have to be placed.

First consider the following operation.

Definition 1 (Simplification of Rules). *A rule can be simplified according to the following operations:*

- $a \wedge a$ *is simplified to* a
- $\neg a \wedge \neg a$ *is simplified to* $\neg a$
- $a \wedge \neg a$ *and* $\neg a \wedge a$ *is simplified to* $\bot$

where a *is an atom.*

Since we are only considering a non-cyclic NLP, a body that has been simplified to $\bot$ is considered to be equivalent of having an empty body $a \leftarrow$.

With such operation, the following can now be considered.

Definition 2 (Minimal Rule). *A rule is considered to be minimal, if its logical formula cannot be simplified further.*

For every Herbrand base $\mathcal{B}$, a finite ordered set of rules $\tau(\mathcal{B})$ that contains all minimal rules for any system that has Herbrand base $\mathcal{B}$ can be generated. In a classification scenario, each rule needs to be assigned to a class. To ease this, a deterministic approach of mapping each rule to an index, which corresponds to a class, is defined.

Definition 3 (Length of a rule). *The length of a rule* $R \in \tau(\mathcal{B})$ *is defined as* $\|b(R)\|$.

Definition 4 (Index of element in ordered set). *Let* S *be an ordered set, the index of element* $e \in S$*, is defined as* $\sigma_S(e) = \|S_{<e}\|$*, where* $\sigma_S : S \mapsto \mathbb{N}$ *and* $S_{<e} = \{x \mid x < e, x \in S\}$.

Definition 5 (Ordered Herbrand Base). *The ordered Herbrand base* $\mathcal{B}_o$ *contains the same elements as* $\mathcal{B}$ *except each element has an ordered relation* $<$.

The relation $<$ on $\mathcal{B}_o$ can be defined arbitrarily, but in most cases, the lexicographical ordering is the most convenient. Now, consider a set of rules $\tau_l(\mathcal{B}_o)$, where $\tau_l(\mathcal{B}_o) = \{R \mid \|b(R)\| \leq l, R \in \tau(\mathcal{B}_o)\} \subseteq \tau(\mathcal{B}_o)$ which contains all rules that are less than or equal to length l. The number of rules in $\tau_l(\mathcal{B}_o)$ can be given by the following formula:

$$\|\tau_l(\mathcal{B}_o)\| = \begin{cases} 1 & \text{if } l = 0, \\ \|\tau_{l-1}(\mathcal{B}_o)\| + \binom{n}{l} \times 2^l & \text{if } l > 0. \end{cases} \tag{3}$$

where $n = \|\mathcal{B}_o\|$ is the number of elements in the Herbrand base and $\binom{n}{k}$ represents the binomial coefficient.

Also consider the ordered set $\widetilde{\tau}_l(\mathcal{B}_o) = \{R \mid \|b(R)\| = l, R \in \tau(\mathcal{B}_o)\}$ containing all the rules R that are exactly of length l. The ordered relation for $\widetilde{\tau}_l$ is defined by first ordering the negation by marking the negative literals as 1s and positive literals as 0s. With that, the position of negations in the literals can be mapped into a binary number. Note that there is no information loss here, as the binary number mapping is only used for ordering. Next, we look at each atom in the rule and order them according to $\mathcal{B}_o$. In this relation, $\{a, b\} < \{a, c\} < \{\neg a, b\} < \{\neg b, c\} < \{\neg a, \neg c\}$.

The index of a rule R $\sigma_{\tau(\mathcal{B}_o)}(R)$ can be obtained by performing the following calculation:

$$\sigma_{\tau(\mathcal{B}_o)}(R) = \|\tau_{l-1}(\mathcal{B}_o)\| + \sigma_{\widetilde{\tau}_l(\mathcal{B}_o)}(R)$$

where $l = \|b(R)\|$ is the length of the rule R.

Recall that in a rule, an atom can be present as either positive, negative, or not be present at all. Therefore the number of possible rules for an n-variable system ($n = \|\mathcal{B}\|$) is $\|\tau_n(\mathcal{B}_o)\| = 3^n$. Thus, for an n-variable system, the total number of classes the to be classified is $3^n \times n$, with each variable in the system taking the head of the rules.

2.3 δLFIT

δLFIT [16] is composed of an LSTM and a feed forward layer. The input for δLFIT L is a continuous sequence of state transitions, in which starting from an initial state I_0, the T_P operator is applied repeatedly to get the entire sequence $L = (I_0, T_P(I_0), T_P(T_P(I_0)), \dots)$. The hidden state from the LSTM is then fed into the feed forward layer. The output of the feed forward layer is then fed to a sigmoid layer, from which the probability for each rules can be obtained. δLFIT assigns each rule, in combination of each head, to 1 output node. Therefore the number of output nodes in δLFIT for a given n variable system is $3^n \times n$.

3 δLFIT+

Several techniques have been applied to the original δLFIT model to make it much more memory efficient and data efficient. These techniques are described in the following sub sections.

3.1 Input Sequence Invariance

In the original δLFIT model, changing the order of the input sequence changes the output. To mitigate this issue, the dataset needs to be augmented with various permutation of the ordering, however this makes the dataset really big. A similar problem existed in the image processing community, whereby each permutation of the image has to be augmented into the dataset. This was the case until a pooling operation in the form of convolution came along.

Conventionally, recurrent neural networks (RNN) are used in order to process sequences. Each element in the sequence takes different amount of operations to reach the output, therefore the semantic of a sequence is inherently reflected in the RNN structure. The LSTM architecture used in δLFIT is an example of an RNN. Recent state of the art however, heavily preferences transformer [21] and the attention mechanism to process sequence. Compared to RNN, transformer is relatively flat with no inherent structure for representing order of sequences. A positional encoding which is directly added to the input value is provided as a means of indicating each elements' positions. There might be an inclination to just remove the positional encoding and expect that the resulting architecture to be sequence invariant. However due to the mathematical operations performed, changing the ordering of the sequence will result in different calculations. To achieve true sequence invariance, we look to the set transformer architecture.

The set transformer architecture [14] is introduced here to make the network invariant to the input sequence. This drastically improves data efficiency and generalization. Set transformer is a variance of the original transformer architecture, but with sequence invariance. In the set transformer architecture, the multi head attention is used as the pooling operation. To ensure that the result of the pooling operation does not change with respect to the order of the input, self-attention is applied to the input. Recall that the multi head attention is a query-key-value operation described as follows [21]:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

the authors of the set transformer architecture observed that if $K = V$, different ordering of elements in K or V results in the same formula. Further, since QK^T is the pairwise relationship between the elements of both matrix, it is thus possible to get the relationship between each of the elements of the input. In the natural language processing world, this is usually equated to how each word in a sentence relates to each other. In the LFIT literature, this can be thought of as the relationship between each state.

3.2 Rule Length Sharing

The original δLFIT assigned each output node to each possible rule. This is obviously problematic as the number of outputs scale by $3 \times n^3$. The reality is much worse as the number of outputs is multiplied by the number of possible heads, which is the number of elements in the Herbrand base. So the first most obvious thing that can lead to reducing the number of outputs, is by reusing the same output node for different heads. This can be accomplished by having an extra input that indicates the head of the rule.

However, this still leaves us with the number of outputs scaling by n^3, which is still exponentially explosive. Here, another reuse strategy can be applied. Namely, the outputs can be reused for rules with different lengths. Each output, depending on the input, represents a different set of rules that is predicted by the neural network. So if the input indicates rule length of 1, then each of the output are rules of length 1.

Therefore, the number of output required now is just the maximum number of rules in one of the lengths. Consider the set $\widetilde{\tau}_l(H) = \{R \mid \|b(R)\| = l\}$ that contains all rules R of length l. From equation (3), the cardinality of this set at l is the following:

$$\|\widetilde{\tau}_l(H)\| = \binom{n}{l} \times 2^l$$

Given that $\lim_{n\to\infty} \frac{2^l}{\binom{n}{l}} = 0$, the maximum number of outputs scale to $O(\binom{n}{l})$. It is clear that $\arg\max_l \binom{n}{l} = \frac{n}{2}$, therefore the maximum number of outputs for any given n, is expected to be $\binom{n}{n/2} \times 2^{n/2}$. This is much less than the original $n \times 3^n$.

3.3 Subsumed Label Smoothing

We apply label smoothing by setting to a value μ in labels which are not the actual rules, but rules that are subsumed by actual rules. With the splitting of the length of the rules, the labels will be largely 0 for the majority of the dataset if only minimal logic programs are considered. For example, consider a logic program consisting of only rules with a maximum of 3 literals in the body. The labels for this logic program at length 4 and beyond will be 0. Since the majority of the dataset is 0, this will encourage the neural network to shortcut and output 0 all the time to minimize the objective. In natual language processing, it is a standard procedure to apply label smoothing to words unrelated as a form of regularization. This technique is applied here in a slightly different way, to help aid in training and not to penalize when the network gets too confident. Each rule that is subsumed in the logic program is set to μ. This avoids scenario where the label is just 0.

3.4 Label Imbalance

Due to the rule length sharing technique described in section 3.2, there is a huge imbalance in the number of positive labels in the dataset. As shown in figure 1, the number of positive labels for the first few classes are much larger. This is due to them being reused in a greater number of rule lengths. Comparing to

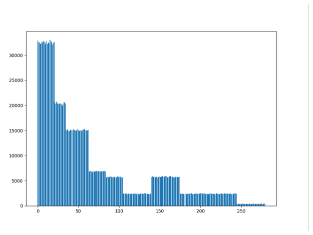

Fig. 1. The amount of positive labels for each class in the dataset.

the last few classes at the right edge, which are only used when $l = n/2$. To counteract this imbalance, a positive example weight is calculated based on the positive:negative example ratio, and then applied to the loss function.

3.5 Network Architecture

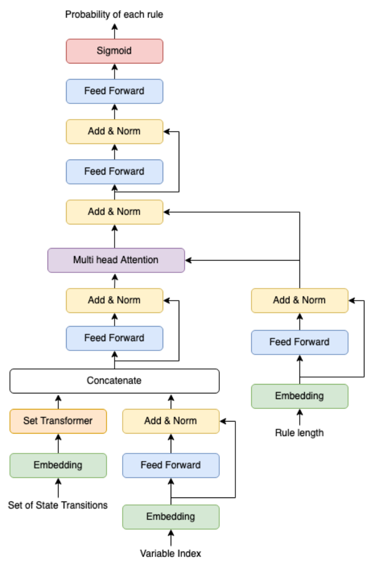

Fig. 2. The network architecture of δLFIT+

The network architecture is designed such that it predicts logical rules for each variable and rule length separately. Each variable and rule length reuse the same parameters and are differentiated through input.

The neural network receives a set of state transitions as input, the variable index and the length of the rule to produce predictions for. The set of state transitions is a set $S_x = \{(I, \theta_x) \mid I \in E\}$ where E is a set of Herbrand interpretations.

θ_x denotes a boolean value of whether the atom x belongs in the interpretation of the next state $T_P(I)$. Since in LFIT, rules are learned by categorizing interpretations in positive examples and negative examples, a similar approach is taken here as well. $(I, 1)$ denote all positive examples while $(I, 0)$ denote all negative examples. More concretely, I is represented as a vector $v \in \{0, 1\}^n$, where n is the size of the Herbrand base. Each element in vector v is set to 0 if it is not in the interpretation and 1 if it is. Note that the index position of each atom in v is order-dependent. This vector v can then be interpreted into a binary number. For an Herbrand base of size n, v can range from 0 to 2^n. To consider whether a given state is a positive example or negative example, 0 to 2^n denotes negative example and $2^n + 1$ to $2^{(n+1)}$ denotes positive example.

The two other inputs, variable index and length of the rule are provided to the network as integers. The network will then learn an approriate embedding for these features to produce a prediction.

The entire neural network architecture is depicted in Figure 2. Each input is fed into an embedding layer. The embedding layer contains learnable weights, followed by a GELU activation [10], and a LayerNorm [1].

The state transition features are then fed into the set transformer described in section 3.1. The other inputs are fed through a feed forward residual layer. The feed forward residual layer typically consist of two affine transformations and a GELU activation. The output of this feed forward layer is then added to the output of the embedding layer followed by a LayerNorm.

The output from the state transitions and variable index layers are then concatenated into a single vector, representing the state transitions and the respective variable to learn from. The result of the concatenation is then fed through another feed forward residual layer.

A multi head attention is then applied, with the key and value both being the features from state transitions and variable. The query being the rule length. The intuition for this is to allow the network to focus on features that are relevant on the rule length it is being asked to predict. The output from the multi head attention layer is then added with the query of rule length, followed by a LayerNorm. This result is then fed through another feed forward layer.

A final feed forward layer is applied, and the output is an element-wise sigmoid. The result represents the probability of each rule described in section 2.2, with respect to the input rule length l and the head of the rule x, for it to be included in the logic program. A threshold δ_t is used to determine the rules to include, note that more than 1 rule can be simultaneously chosen for the same head x.

3.6 Training Methods

Training is performed by generating artificial training data. Each data point corresponds to a randomly generated normal logic program. A random sequence of state transitions that conforms to the deterministic semantics is first generated. In deterministic semantics, a single state only transitions to another particular state. The symbolic bottom-up LFIT algorithm is then applied to this random

Algorithm 1: The δLFIT+ algorithm

```
Input  : Set of transitions $S = \{(I, T_P(I) \mid I \in E\}$, Herbrand base $\mathcal{B}$
Output: Logic program $P$
$n := \|\mathcal{B}\|$;
$P := \{\}$;
foreach $x \in \mathcal{B}$ do
    Encode each element $s \in S$ into binary numbers $S_x$;
    $R_x := \{\}$;
    for $l = 0$ to $n$ do
        $R_{xl} := \delta$LFIT+ $(S_x, x, l)$;
        /* $R_{xl}$ contains all rules of the form $x \leftarrow \beta$ where $\beta$ is a
           conjuction of literals and $\|\beta\| = l$                              */
        Simplify $R_{xl}$;
        $R_x = R_x \cup R_{xl}$;
    end
    Simplify $R_x$ ;          // $R_x$ contains all rules of the form $x \leftarrow \beta$
    /* After simplification $R_x$ will contain one or more rules that
       are not subsumed by each other                                       */
    $P = P \cup R_x$;
end
return $P$
```

sequence to learn a minimal logic program. The data point is then the sequence of state transitions, and the minimal logic program as the label. For each data point, the full initial states (all 2^n states) are generated. However during training, only 80% of the data point is selected to improve the generalizability of the network.

Subsumed label smoothing is applied randomly to 20% of the rules each batch. For each subsumed rule, the label gets 0.25 added to its value. It is possible for a single rule to be subsumed by multiple rules, therefore a maximum of 0.75 is also applied, to not let the network confuse with the true label.

3.7 δLFIT+ Algorithm

The δLFIT+ algorithm is described in algorithm 1. The resulting logic program is produced by iterating through every variable in the Herbrand base, and every length up to the number of variables. Simplification of the rules is performed at each step to eliminate rules such as $a \leftarrow b$ and $a \leftarrow \neg b$. Only the top 10 rules that exceeds 0.97 probability is selected into R_{xl}.

4 Experiments

To evaluate our methods, we perform several experiments comparing the performance with and without our improvements. The network is implemented in PyTorch [15]. We also perform comparisons with the original δLFIT model.

Boolean Network	δLFIT	δLFIT+^{3}	δLFIT+^{5}	δLFIT+^{7}
3-node (a)	0.095	0.271	0.271	0.271
3-node (b)	0.054	0.188	0.208	0.208
Raf	0.253	0.188	0.208	0.208
5-node	0.142	-	0.278	0.325
7-node	-	-	-	0.223
WNT5A [22]	-	-	-	0.194

Table 1. The MSE for the state transitions generated by the predicted logic programs compared to δLFIT.

Evaluations are performed on boolean networks taken from the PyBoolNet [13] repository, which are commonly used for such methods.

4.1 Experimental Methods

The network was trained with the Adadelta [23] optimizer. After training for 10 epochs, the network is evaluated by feeding data from the boolean networks. We then produce state transitions from the predicted logic program, and calculated the mean squared error between the state transitions from the predicted logic program and the boolean network.

During training, we only focus on one of the variables and one particular rule length for each data point. The selection for the variable is randomized within each batch, but the selection for the rule length is fixed for each batch, instead randomizing between different batches. This is done primarily for computational efficiency reasons. For each data point, we pick 80% of the states from all possible states for training. E.g., for 5 variables there are $2^5 = 32$ possible states, thus we pick 25 state transitions for training. These 25 states are picked randomly each epoch. Validation is performed on data point withheld from the training process. The results are taken only from 1 run of the experiment, due to the lengthiness in the process of training, and also when we did multiple runs on the smaller systems, we did not observe significant variance in the results.

4.2 Baseline

Baseline results are shown in table 1. δLFIT wasn't able to deal with more than 5 variables therefore the results are omitted for 7 variable networks. The top number next to δLFIT+ represents the number of variables that was trained on. δLFIT+^{3} means that the specific network was trained with 3 variables, and thus have 12 output nodes. Networks trained with more variables have more output nodes and thus can deal with larger boolean networks, together with smaller networks.

4.3 Regular Transformer

In this experiment, we replaced the set transformer in the network architecture to a regular transformer. The results are shown in table 2. Results are taken by randomizing the order of the state transitions and taking the average.

Boolean Network	δLFIT+^{5}	δLFIT+$^5_{\neg T}$	δLFIT+$^5_{\neg S}$
3-node (a)	0.271	0.313	0.271
3-node (b)	0.208	0.292	0.208
Raf	0.208	0.271	0.208
5-node	0.278	0.375	0.378

Table 2. The MSE for the state transitions generated by the predicted logic programs with and without set transformers, and with and without label smoothing

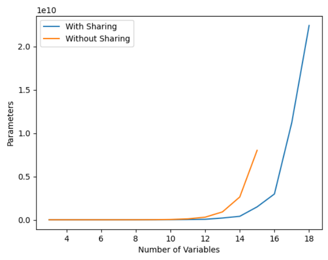

Fig. 3. Number of parameters in 10 billions, of the network with rule length sharing compared to without rule length sharing

4.4 Rule Length Sharing

We show the difference in number of parameters with rule length sharing compared to without rule length sharing in figure 3. Without this, the number of parameters start exploding from 14 variables. However, with rule length sharing this effect is delayed until 16 variables. Contrasting with the state of the art natural language processing model like GPT-3, which has 175 billion parameters [2], there are still some leeway in terms of network architecture, however it will require state of the art hardware and large amount of computational time to actually train it. We did not perform any experiments to validate the difference in terms of accuracy because it will require a significant restructuring in the training process. We think that the restructuring is sufficiently impactful that the difference in accuracy that we will be measuring will not just be the effect of sharing rule lengths, but also the difference in the entire training process.

4.5 Without Label Smoothing

In this experiment, we trained the network without applying subsumed label smoothing. The results are shown in table 2 with the results for δLFIT+$^5_{\neg S}$ being one without label smoothing applied. No clear performance difference was found with the 3 variable networks, but the network that trained without label smoothing performed significantly worse with the 5 node boolean network.

5 Discussion

In general, we observed that there is a drop in accuracy across the board as the outputs get reused. δLFIT+ performed poorer on the other networks, except in the boolean network Raf, where δLFIT struggled because the same state was repeated. We also observed that δLFIT+^{3} performing better than δLFIT+^{5} for 3 variable networks. Memory efficiency has also improved from δLFIT, which allowed us to easily train a network with 7 variables. However, generating training data for 8 variables and beyond required too much time with the amount of computational resource that we had, therefore we were unable to train the networks.

With the rule length sharing method proposed, we were able to cut the number of parameters in the model by half. This allowed us to use larger number of batch size during training to increase training speed. However with the reduced number of output nodes, this might have led to a potential loss in terms of accuracy, as now each output node represents multiple rules, compared to one output node one rule in δLFIT.

Subsumed label smoothing did not have any visible impact on smaller networks with 3 variables, but with 5 variables the effects were visible. In particular, with the same hyperparameter, we observed that δLFIT+$^5_{\neg S}$ was overfitting, meaning training loss decreased very rapidly while validation loss remained high. Adding subsumed label smoothing helped apply some regularization to δLFIT+^{5}.

δLFIT+ took 1 day to generate and train for 3 variable systems, while it took 5 days to generate and train for 7 variable systems. Compared to other LFIT methods, the runtime for δLFIT+ is significantly higher. However, most methods only focus on learning a single system whereas δLFIT+ learns a general n-variable system. The advantage for this is that, in real world applications, novel systems that are being investigated often have only small amount of observation data available. A technique that only works with the obtained observation data will overfit, whereas δLFIT+ can avoid the overfitting problem.

6 Related Work

The combination of symbolic methods and neural networks, recently also referred to as Neural-Symbolic AI (NSAI), has been an active area of research for several years [8], but has recently gained attention. Much of the attention stems from the lack of interpretability in deep learning, where there have been so much success in various different fields. In the work done by Garcez and Zaverucha [7], the authors proposed an algorithm that first translates a logic program that is provided as background knowledge into a neural network. The neural network is then trained with examples, then a logic program is extracted based on the weights of the neural network. In another work done by Garcez et al. [4], the authors proposed a method to extract logic program from a neural network that was trained from scratch. However, these methods place severe constraints on the architecture of the neural network and a heavy assumption on the neural network model itself. Thus they are unable to capitalize on the recent advancements in neural network and a new method of extraction needs to be devised before they can be applied to new architectures.

In δILP [6], the authors introduced an algorithm that incorporates differentiable machine learning and ILP. Again, the basic idea here is that given an ILP task, the algorithm learns differentiable parameters that allow it to solve the ILP task. A logic program is then subsequently extracted from the learned parameters. While this method suffers from an exponential dependency on the number of allowed predicates, and thus has scalability problems, the main difference with our approach is that we do not perform extraction to obtain logic programs.

More recently, Dong et al. [5] proposed Neural Logic Machines (NLM) that solved to a certain extent the scalability problem that δILP faced. This work joins a different class of family like Neural Programmer-Interpreters [17], in which neural networks are used to approximate the semantics of a program. Our work differs in that we use neural networks to classify different semantics for different programs, where as NLM is learning to emulate the semantics of a program.

7 Conclusion

In this work, we have shown a method to exploit symbolic invariance with neural network. We described several other improvement techniques to help achieve better data efficiency and memory efficiency. We then performed several experiments and contrast it with δLFIT. We also showed the effectiveness of several of our techniques.

There are several more improvements that can be made to further increase data efficiency, like attempting to exploit invariance in the ordering of the variables. The ordering of variables are invariant as a whole, but has to be dependent on each other which makes it non-trivial to exploit, therefore further research needs to be done in order to achieve this. We can also apply this technique to other semantics, such as systems that have delays, or are asynchronous.

References

1. Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016.
2. Tom B. Brown, Benjamin Mann, et al. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020.
3. Artur S. d'Avila Garcez, Marco Gori, Luís C. Lamb, Luciano Serafini, Michael Spranger, and Son N. Tran. Neural-symbolic computing: An effective methodology for principled integration of machine learning and reasoning. *CoRR*, abs/1905.06088, 2019.
4. A.S d'Avila Garcez, K Broda, and D.M Gabbay. Symbolic knowledge extraction from trained neural networks: A sound approach. *Artificial Intelligence*, 125(1):155 – 207, 2001.

5. Honghua Dong, Jiayuan Mao, Tian Lin, Chong Wang, Lihong Li, and Denny Zhou. Neural logic machines. *CoRR*, abs/1904.11694, 2019.
6. Richard Evans and Edward Grefenstette. Learning explanatory rules from noisy data. *CoRR*, abs/1711.04574, 2017.
7. Artur S Avila Garcez and Gerson Zaverucha. The connectionist inductive learning and logic programming system. *Applied Intelligence*, 11(1):59–77, 1999.
8. Artur S. d'Avila Garcez, Dov M. Gabbay, and Krysia B. Broda. *Neural-Symbolic Learning System: Foundations and Applications*. Springer-Verlag, Berlin, Heidelberg, 2002.
9. Enguerrand Gentet, Sophie Tourret, and Katsumi Inoue. Learning from interpretation transition using feed-forward neural network. In *Proceedings of ILP 2016, CEUR Proc. 1865*, pages 27–33, 2016.
10. Dan Hendrycks and Kevin Gimpel. Bridging nonlinearities and stochastic regularizers with gaussian error linear units. *CoRR*, abs/1606.08415, 2016.
11. Katsumi Inoue. Logic programming for boolean networks. In Toby Walsh, editor, *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, pages 924–930. IJCAI/AAAI, 2011.
12. Katsumi Inoue, Tony Ribeiro, and Chiaki Sakama. Learning from interpretation transition. *Machine Learning*, 94(1):51–79, 2014.
13. Hannes Klarner, Adam Streck, and Heike Siebert. PyBoolNet: a python package for the generation, analysis and visualization of boolean networks. *Bioinformatics*, 33(5):770–772, 12 2016.
14. Juho Lee, Yoonho Lee, Jungtaek Kim, Adam R. Kosiorek, Seungjin Choi, and Yee Whye Teh. Set transformer. *CoRR*, abs/1810.00825, 2018.
15. Adam Paszke et al. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.
16. Yin Jun Phua and Katsumi Inoue. Learning logic programs from noisy state transition data. In Dimitar Kazakov and Can Erten, editors, *Inductive Logic Programming - 29th International Conference, ILP 2019, Plovdiv, Bulgaria, September 3-5, 2019, Proceedings*, volume 11770 of *Lecture Notes in Computer Science*, pages 72–80. Springer, 2019.
17. Scott Reed and Nando De Freitas. Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279*, 2015.
18. Tony Ribeiro and Katsumi Inoue. Learning prime implicant conditions from interpretation transition. In Jesse Davis and Jan Ramon, editors, *Inductive Logic Programming*, pages 108–125, Cham, 2015. Springer International Publishing.
19. Tony Ribeiro, Morgan Magnin, Katsumi Inoue, and Chiaki Sakama. Learning delayed influences of biological systems. *Frontiers in Bioengineering and Biotechnology*, 2:81, 2015.
20. Teemu Rintala et al. Using boolean network extraction of trained neural networks to reverse-engineer gene-regulatory networks from time-series data. 2019.
21. Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.
22. Yufei Xiao and Edward R. Dougherty. The impact of function perturbations in Boolean networks. *Bioinformatics*, 23(10):1265–1273, 03 2007.
23. Matthew D. Zeiler. ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701, 2012.