NEURAL LOGIC MACHINES

Anonymous authors
Paper under double-blind review

ABSTRACT

We propose Neural Logic Machines (NLMs), a neural-symbolic architecture for both inductive learning and logic reasoning. NLMs exploit the power of both neural networks—as function approximators for probabilistic distributions, and logic programming—as symbolic processor for objects with properties, relations, logic connectives, and quantifiers. After being trained on small-scale tasks (such as sorting short arrays), NLMs can learn the underlying logic rules, and generalize to arbitrarily large-scale tasks (such as sorting arbitrarily long arrays). In our experiments, NLMs achieve perfect generalization in a number of tasks, from relational reasoning tasks on family tree and general graphs, to decision making tasks including sorting, finding shortest paths, and the blocks world. Most of these tasks are hard to accomplish for neural networks or logical programming alone.

1 INTRODUCTION

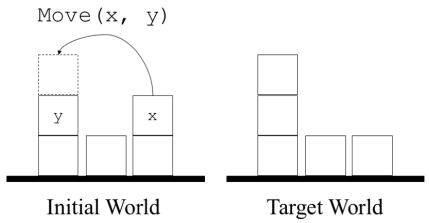
Deep learning has achieved great success in many applications such as speech recognition (Hinton et al., 2012), image classification (Krizhevsky et al., 2012; He et al., 2016), machine translation (Sutskever et al., 2014; Bahdanau et al., 2015; Wu et al., 2016; Vaswani et al., 2017), and game playing (Mnih et al., 2015; Silver et al., 2017). Starting from Fodor & Pylyshyn (1988), however, there has been a debate over the problem of systematicity (such as understanding recursive systems) in connectionist models (Fodor & McLaughlin, 1990; Hadley, 1994; Jansen & Watter, 2012).

Logic systems can naturally process symbolic rules in language and reasoning. Inductive logic programming (ILP) (Muggleton, 1991; 1996; Friedman et al., 1999) has been developed for learning logic rules from examples. Roughly speaking, given a collection of positive and negative examples, ILP learns a set of logic rules (with uncertainty) that entails all of the positive examples (with high probability) but none of the negative examples. Combining both symbols and probabilities, many problems arose from high-level cognitive abilities, such as systematicity, can be intrinsically resolved. However, due to an exponentially large combinatorial space of all possible rules, it is difficult for ILP to scale beyond small-sized rule sets (Dantsin et al., 2001; Evans & Grefenstette, 2018).

To illustrate the challenges for neural networks and logic programming, let us consider the classic blocks world problem (Nilsson, 1982; Gupta & Nau, 1992). In this problem, we have a set of blocks on a ground (Figure 1). We can move a block and place it on the top of another block or the ground, as long as the first block is *moveable* and the second block is *placeable*. A block is said to be moveable or placeable if there is no other block on it. The ground is always placeable. This implies that we can simultaneously place all blocks on the ground. Denote by $\texttt{Move}(x, y)$ the operation of moving block x onto y which can be a block or the ground. Given any initial configuration of the blocks world, our goal is to transform it to a target configuration via taking a sequence of `Move` operations.

Although the blocks world problem may appear simple at first glance, two major challenges exist in building a learning system to automatically accomplish such a task:

1. We expect the learning system to generalize to blocks worlds which contain arbitrarily more blocks than those used in training. For the readers who are not familiar with the blocks world problem to understand this challenge, they may look at the task of learning to sort arrays (e.g., see Vinyals et al. (2015)), where recurrent neural networks fail to generalize to arrays which are even just slightly longer than those used in training.
2. We expect the learning system to scale with the number of logic rules. Existing logic-based algorithms like ILP suffer an exponential computational complexity with respect to the number of logic rules (Dantsin et al., 2001; Evans & Grefenstette, 2018).



$\texttt{On}(x, y)$	True if x is on y
$\texttt{IsGround}(x)$	True if x is the ground
$\texttt{Clear}(x)$	True if there is no block on x
$\texttt{Moveable}(x)$	$\neg\texttt{IsGround}(x) \wedge \texttt{Clear}(x)$
$\texttt{Placeable}(x)$	$\texttt{IsGround}(x) \vee \texttt{Clear}(x)$
$\texttt{Move}(x, y)$	Move x onto y

Figure 1: (Left) A graphical illustration of the blocks world. Given an initial and a target worlds, the agent is required to move blocks to transform the initial configuration to the target one. (Right) A set of sentences used throughout the paper to define the blocks world.

In addition, in many other scenarios, we expect the learning system to deal with high-order relational data and quantifiers that go beyond the scope of typical graph-structured neural networks (Kipf & Welling, 2016). For example, to apply the transitivity on a relation: $\exists b\ r(a, b) \wedge r(b, c) \rightarrow r(a, c)$, we need a ternary relation among a tuple of objects (a, b, c).

In this paper, we propose Neural Logic Machines (NLMs) to address the challenges we discussed above. In a nutshell, NLMs offer a neural-symbolic architecture to implement first-order predicate calculus (FOPC). The key intuition behind NLMs is that probabilistic logic rules can be efficiently approximated by neural networks, and the wiring among neural modules can implement the logic quantifiers. Unlike NLMs, ILP relies on an external automated reasoning component.

The rest of the paper is organized as follows. We first revisit some useful definitions in FOPC and define our neural implementation of an FOPC induction system in Section 2. We refer interested readers about the training and technical details of NLM to the Appendix A. In Section 3 we evaluate the effectiveness of NLM on a broad set of tasks ranging from relational reasoning to decision making. We discuss related works in Section 4, and conclude the paper in Section 5.

2 Neural Logic Machines (NLM)

The NLM is a neural realization of (symbolic) logic machines. In general, a logic machine consists of two parts: a set of logic deduction rules, and a deduction system. Given a set of premises or assumptions, the deduction system applies rules in this set to draw a conclusion on a target statement. In the blocks world, a rule may be like $\neg\texttt{IsGround}(x) \wedge \texttt{Clear}(x) \implies \texttt{Moveable}(x)$, where x is a *variable* and can be grounded into a block or the ground. This rule can be used by the deduction system to deduce whether a given object is moveable or not.

2.1 Primitive Logic Rules

The following *primitive rules* are used by logic machines as basic building blocks:

1. **Boolean logic**. We use the template below for boolean logic:

$$\texttt{expression}(x_1, x_2, \cdots, x_n) \implies p(x_1, x_2, \cdots, x_n), \tag{1}$$

   where `expression` can be any *boolean* expression consisting of predicates over all variables $(x_1, \ldots, x_n)$ and $p(\cdot)$ is the conclusive predicate.

2. **Quantifiers**. We introduce quantifiers using two types of templates: expansion and reduction. Let p be a predicate, and we have

$$\textbf{(Expansion)} \qquad p(x_1, x_2, \cdots, x_n) \implies \forall x\ q(x_1, x_2, \cdots, x_n, x)\,, \tag{2}$$

   where $x \notin \{x_i\}_{i=1}^n$. The expansion operation constructs a new predicate q from p, by introducing a new variable x. For example, consider the following boolean logic expression

$$\texttt{Moveable}(x) \wedge \texttt{Placeable}(y) \implies \texttt{ValidMove}(x, y).$$

   This expression does not fit the template in Eq. 1 as some predicates on the LHS only take a *subset* of the variables as inputs. [1] However, it can be described using the expansion template:

[1] Although this is an axiom in FOPC and should be implemented by the deduction system, we implement this as a primitive rule to reduce the deduction complexity.

1. $\texttt{Moveable}(x) \implies \forall z\, \texttt{MoveableX}(x, z)$; (by Eq. 2)
2. $\texttt{Placeable}(y) \implies \forall z\, \texttt{PlaceableY}(y, z)$; (by Eq. 2)
3. $\texttt{MoveableX}(x, y) \wedge \texttt{PlaceableY}(y, x) \implies \texttt{ValidMove}(x, y)$. (by Eq. 1)

The other template is for reduction:

(Reduction) $$\forall x\, p(x_1, x_2, \cdots, x_n, x) \implies q(x_1, x_2, \cdots, x_n)\,, \tag{3}$$

where the $\forall$ quantifier can also be $\exists$. The reduction operation eliminates a variable in a predicate via the $\forall$ or $\exists$ quantifier. As an example, consider the rule to deduce moveability of an object,

$$\neg\texttt{IsGround}(x) \wedge \neg\big(\exists y\, \texttt{On}(y, x)\big) \implies \texttt{Moveable}(x)\,,$$

which can be expressed using primitive rules as follows:

1. $\forall y\, \neg\texttt{On}(y, x) \implies \texttt{Clear}(x)$; (by Eq. 3)
2. $\neg\texttt{IsGround}(x) \wedge \texttt{Clear}(x) \implies \texttt{Moveable}(x)$. (by Eq. 1)

Remark. It can be verified that any FOPC rules can be decomposed into a set of NLM primitive rules by recursively resolving the outermost quantifier. Accordingly, logic deduction can be implemented as a sequential procedure of applying NLM primitive rules.

2.2 Neural Boolean Logic

We start from the modeling of a single boolean expression in the form of Eq. 1. Corresponding to NLM's neural boolean logic is a tensor data structure to represent inputs and intermediate computation results. In NLM, a group of $C_t^{(1)}$ unary predicates grounded on m objects is represented by a tensor of shape $m \times C^{(1)}$, describing a group of "properties of objects". We can also describe $C^{(2)}$ "pairwise relations between objects" as a group of binary predicates, represented by a tensor of shape $m \times (m-1) \times C^{(2)}$. Higher order relations can be similar represented by tensors of higher ranks. In practice, we set a maximum rank of the tensors, called the *breadth* of the NLM.

In contrast to symbolic ILP approaches that assign `true`/`false` values to logic expressions, the NLM takes a probabilistic approach as in probabilistic FOPC. Each statement s is associated with a real value, denoted $\mathrm{prob}(s)$, that represents $\Pr[\text{statement } s \text{ is } \texttt{true}]$. Thus, components in the tensors take values in $[0, 1]$.

A probabilistic boolean expression can now be viewed as a mapping of the following form:

$$\texttt{expression}\big(\mathrm{prob}(p_1(x_1, \cdots, x_n)), \cdots, \mathrm{prob}(p_k(x_1, \cdots, x_n))\big) \implies \mathrm{prob}(p(x_1, \cdots, x_n))\,,$$

where $\mathcal{B} = \{p_1, p_2, \cdots, p_k\}$ is a set of base predicates and p is the conclusive predicate. If we allow p to belong to $\mathcal{B}$, then *recurrent* rules can be constructed.

In general, the inputs $\{\mathrm{prob}(p_1), \ldots, \mathrm{prob}(p_k)\}$ are not independent of each other and should be modeled by a joint distribution. This is done in NLM by exploiting the expressiveness of neural networks, which model the probabilities with feed-forward multi-layer perceptrons (MLPs):

$$\mathrm{prob}(p) = \sigma\Big(\mathrm{MLP}\Big(\mathrm{prob}(p_1(\cdots)), \ldots, \mathrm{prob}(p_k(\cdots)); \theta\Big)\Big)\,, \tag{4}$$

where σ is the sigmoid nonlinearity and θ the trainable parameters of the network.

Example. Consider the blocks world described earlier. An object a, which may be a block or ground, is represented by its position coordinates (x_a, y_a). (Further details are found in Section 3.) To handle numerals, we first transform the input coordinates into $C = 6$ binary relations between objects: $\texttt{Left}(a, b)$ (whether a is to the left of b, or mathematically, $\mathbf{1}[x_a < x_b]$), and similarly, $\texttt{SameX}(a, b)$, $\texttt{Right}(a, b)$, $\texttt{Up}(a, b)$,$\texttt{SameY}(a, b)$ and $\texttt{Down}(a, b)$, where $a \neq b$. Therefore, the input to the NLM is a tensor of shape $m \times (m-1) \times C$, where m is the total number of objects.

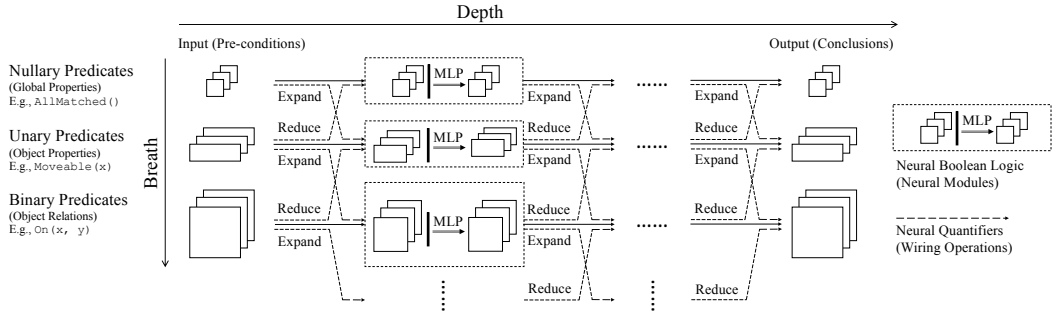


Figure 2: An illustration of Neural Logic Machines (NLM). NLM takes object properties and relations as input, performs sequential logic deduction, and outputs conclusive properties or relations of the objects. Implementation details can be found in Section 2.3.

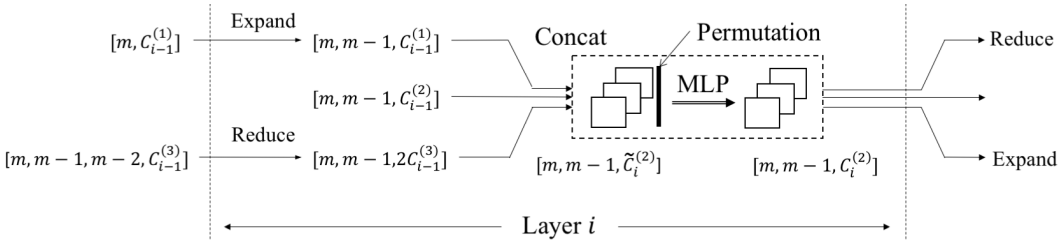


Figure 3: An illustration of the computational block inside NLM for binary predicates at layer i. $C_i^{(j)}$ denotes the number of output predicates of layer i group j. $[\cdot]$ denotes the shape of the tensor.

2.3 Neural logic deduction

Figure 2 gives the overall layer-wise structure of an NLM. Each layer is designed to efficiently realize rules derived from the primitive templates in Sec. 1. As the the number of layers increases, higher levels of abstraction can be formed. For example, some predicate outputted by the first layer may represent $\texttt{Clear}(x)$, while the second layer can output more complicated predicates like $\texttt{Moveable}(x)$. Thus, forward propagation in an NLM can be interpreted as recursive applications of logic rules on the input pre-conditions to deduce a conclusion, which is encoded by the output of the last layer. The number of layers of an NLM is called its *depth*. The rest of this subsection provides details of the deduction process, focusing on a particular layer i (Figure 3).

Forward propagation at layer i. At any particular layer i, NLM takes a set of known facts $\mathcal{I}_i$ as input from the previous layer and outputs a new set of facts $\mathcal{O}_i$. For the first layer, its input is the pre-conditions, such as the current configuration in the blocks world. The facts are tensors that encode relations among multiple objectives, as described in Sec. 2.2.

Computation at each layer is grouped by the ranks of the tensors in $\mathcal{I}_t$. *Intra*-group computation thus involves tensors of the same rank. It is to implement neural logic expressions of the form of Eq. 1 that have the same number of variables in the predicates. In contrast, *inter*-group computation is used to enable quantification (Eqs. 2 & 3). Together, intra- and inter-group computation brings to NLM the full capacity of modeling any FOPC rules, up to the extent limited by the NLM's depth and breadth.

Intra-group computation is to realize neural logic expressions (Eq. 1). Consider a specific output predicate at layer i: $p^{(\text{O})}(x_1, x_2, \cdots x_n)$. It is conditioned on all input predicates $p_j^{(\text{I})}$ in the *same* group (*i.e.*, having same numbers of parameters) from input $\mathcal{I}_{i-1}$, and grounded on *arbitrary permutations* of $\{x_1 \cdots x_n\}$. As an example, a tenary predicate $p^{(\text{O})}(a, b, c)$ is conditioned on $p_j^{(\text{I})}(a, b, c)$, $p_j^{(\text{I})}(a, c, b)$, $p_j^{(\text{I})}(b, a, c)$, $p_j^{(\text{I})}(b, c, a)$, $p_j^{(\text{I})}(c, a, b)$, and $p_j^{(\text{I})}(c, b, a)$ (all permutations of the parameters) and for all j (all input tenary predicates).

More precisely, consider a specific group of $\widetilde{C}_i^{(n)}$ predicates having n parameters grounded on m symbols at layer i, the input can be represented by a tensor of shape $[m, m-1, \ldots, m-n+1, \widetilde{C}_i^{(n)}]$,

where $\widetilde{C}_i^{(n)}$ indicates the number of input predicates of group n. The permutation step (shown in Figure 3) permute all n variables, resulting in a new tensor of shape $[m, m-1, \dots, m-n+1, C_i^{(n)}]$, where $C_i^{(n)} = n!\widetilde{C}_i^{(n)}$. A feed-forward network $\mathrm{MLP}_i^{(n)}$ is then applied to this tensor. Note that the MLP depends only on the layer (i) and group (n); the same parameters are used on all sets of n objects. Such a parameter sharing mechanism is crucial to the generalization ability of NLM to problems ov varying sizes.

Inter-group computation relies a wiring mechanism to connect (vertically) consecutive groups between two adjacent layers to realize quantifiers. These connections are labeled "Expand" or "Reduce" in Figures 2 and 3.

The expansion template (Eq. equation 2) introduces a new and distinct variable to obtain a new predicate. NLM realizes this template by adding a new dimension to the input tensor. More precisely, a predicate defined over n variables and grounded on m symbols is represented by a tensor U of size $m^{\underline{n}} = m \times (m-1) \times (m-2) \times \cdots \times (m-n+1)$. After the expansion, the new predicate is represented by another tensor U_{expand} of size $m^{\underline{n-1}} = m^{\underline{n}} \times (m-n)$, where each component in U is repeated $(m-n)$ times along the newly introduced dimension in U_{expand}.

The reduction template (Eq. equation 3) eliminates a variable x via quantifiers. NLM realizes $\exists$ by taking the maximum value along the dimension corresponding to x, and $\forall$ by taking the minimum similarly. More precisely, a predicate defined over $n+1$ variables $(x_1, \dots, x_n, x)$ and grounded on m symbols is represented by a tensor U of size $m^{\underline{n-1}} = m \times (m-1) \times \cdots \times (m-n)$. After reduction, the resulting tensor has a new size of $m^{\underline{n}}$, whose components are the maximum/minimum values along the last dimension of U that corresponds to the eliminated variable x.

Example. We finish this subsection, continuing with the blocks world to illustrate the forward propagation in NLM. For concreteness, consider the group for *binary* predicates at layer i in Figure 3. The computation of the module begins with the concatenation of the output of (vertically) consecutive blocks (unary, binary, ternary) from the (horizontally) previous layer $i-1$. Formally, denote this input by $I_i^{(2)}(x, y)$, for all pairs $x \neq y$, where subscript i refers to the layer and the superscript "(2)" refers to the group (for binary predicates). Then, $I_i^{(2)}$ contains $\widetilde{C}_i^{(2)} \triangleq C_{i-1}^{(1)} + C_{i-1}^{(2)} + 2C_{i-1}^{(3)}$ predicates for each pair of objects, where $C_{i-1}^{(d)}$ is the number of d-ary predicates outputted by the previous layer. Note that we need to multiply $C_{i-1}^{(3)}$ by 2 because there are two quantifiers ($\forall$ and $\exists$) for reduction. In summary, the shape of $I_i^{(2)}$ is $[m, m-1, \widetilde{C}_i^{(2)}]$.

The output for each pair of objects (x, y) is computed by: $O_i^{(2)}(x, y) = \mathrm{MLP}_i^{(2)}(I_i^{(2)}(x, y) \oplus I_i^{(2)}(y, x))$, where $\oplus$ denotes concatenation. Expansion reflects Eq. 2, which produces a tensor of shape $[m, m-1, C_{i-1}^{(1)}]$ by replicating the previous output $O_{i-1}^{(1)}$ for $m-1$ times. Reduction is always performed on the last parameter, reflecting Eq. 3. The number of parameters in the block described above is $2! \times \widetilde{C}_i^{(2)} \times C_i^{(2)}$ (when using 1-layer MLP), and the computation ($\mathrm{MLP}_i^{(2)}$) is performed individually and identically on each pair of objects.

2.4 Expressiveness

The expressive power of NLM depends on multiple factors:

1. The depth of NLM (*i.e.*, number of layers) restricts the maximum number of deduction steps.
2. The breadth of NLM (*i.e.*, maximum number of variables in all predicates considered) limits the order of relations among objects.
3. The number of predicates used at each layer ($C_i^{(d)}$ in Figure 3). In our experiments, this number is often small (*e.g.*, 8 or 16).
4. In Eq. equation 4, the expressive power of MLP (number of hidden layers and number of hidden neurons) restricts the complexity of the boolean logic that can be represented. In our experiments, we usually prefer shallow networks (*e.g.*, 1 or 2 layers) with a small number of hidden neurons (*e.g.*, 8 or 16). This can be viewed as a low-dimension regularization on the logic complexity and encourages the learned rule to be simple.

3 EXPERIMENTS

In this section, we show that NLM can solve a broad set of tasks, ranging from relational reasoning to decision making. Furthermore, we train NLM on problems of small sizes and show that it generalizes to larger problems of arbitrary sizes. In the experiments, Softmax-Cross-Entropy loss is used for supervised learning tasks, and REINFORCE (Sutton & Barto, 1998) is used for reinforcement learning tasks.

Due to space limitation, interested readers are referred to Appendix A for details of training (including curriculum learning) in the decision making tasks, and Appendix B for more implementation details (such as residual connections (He et al., 2016)), hyper-parameters (such as model depths), and model selection rules.

3.1 BASELINES

We consider two baselines as representatives of the connectionist and symbolicist: Memory Networks (MemNN) (Sukhbaatar et al., 2015) and Differentiable Inductive Logic Programming (∂ILP) (Evans & Grefenstette, 2018), a state-of-the-art ILP framework. We also make comparison with other models such as Neural Turing Machines (NTMs) and graph neural networks whenever eligible.

For MemNN, in order to handle an arbitrary number of inputs (properties, relations), we adopt the method from Graves et al. (2016). Specifically, each object is assigned with a unique identifier (a binary integer ranging from 0 to 255), as its "name". The memory of MemNN is now a set of "pre-conditions". For unary predicates, the memory slot contains a tuple $(\texttt{id}(x), 0, \texttt{properties}(x))$ for each x, and for binary predicates $p(x, y)$, the memory slot contains a tuple $(\texttt{id}(x), \texttt{id}(y), \texttt{relations}(x, y))$, for each pair of (x, y). Both $\texttt{properties}(x)$ and $\texttt{relations}(x, y)$ are length-k vectors v, where k is the number of input predicates. We number each input predicate with an integer $i = 1, 2, \cdots, k$. If object x has a property $p_i(x)$, then $v[i] = 1$; otherwise, $v[i] = 0$. If a pair of objects (x, y) have relation $p_i(x, y)$, then $v[i] = 1$; otherwise, $v[i] = 0$. We extract the key and value for MemNN's to lookup on the given pre-conditions with 2-layer multi-layer perceptrons (MLP). MemNN relies on iterative queries to the memory to perform relational reasoning. Note that MemNN takes a sequential representation of the multi-relational data.

For ∂ILP, the set of pre-conditions of the symbols is used direclty as input of the system.

3.2 FAMILY TREE REASONING

Family tree is a benchmark for inductive logic programming, where the machine is given a family tree containing m members. The family tree is represented by the following relations (predicates): `IsSon`, `IsDaughter`, `IsFather` and `IsMother`. The machine is asked to reason out other properties of family members or relations between them. Our results are summarized in Table 1.

For MemNN, we treat the problem of relation prediction as a question answering task. For example, to determine whether member x has a father in the family tree, we input $\texttt{id}(x)$ to MemNN as the question. MemNN then performs multiple queries to the memory and updates its hidden state. The finishing hidden state is used for classifying whether $\texttt{HasFather}(x)$. For relations (binary predicates), the corresponding MemNN takes the concatenated embedding of $\texttt{id}(x)$ and $\texttt{id}(y)$ as the question.

For ∂ILP, we take the grounded probability of the "target" predicate as the output; for NLM we take the corresponding group of output predicates at the last layer (for property prediction, we take unary predicates, while for relation prediction we take binary predicates) and classify the property or relation with a linear regression model on the grounded probabilities.

All models are trained on instances of size 20 and tested on instances of size 20 and 100 (size is defined as the number of family members). The models are trained with fully supervised learning (labels are available for all objects or pairs of objects). During the testing phase, the accuracy is evaluated (and averaged) on all objects (for properties such as `HasFather`) or pairs of objects (for relations such as `IsUncle`). `MGUncle` is defined as one's maternal great uncle, which is also used by Differentiable Neural Computer (DNC) (Graves et al., 2016). We report the performance of MemNN in the format of Micro / Macro accuracy. We also try our best to replicate the setting

Table 1: Comparison among MemNN, ∂ILP and the proposed NLM in family tree and graph reasoning, where m is the size of the testing family trees or graphs. Both ∂ILP and NLM outperform the neural baseline and achieve perfect accuracy (100%) on our test set. Note the N/A mark for ∂ILP, which cannot scale up to the task 2-OutDegree.

Family Tree	MemNN		∂ILP		NLM (Ours)	
	$m = 20$	$m = 100$	$m = 20$	$m = 100$	$m = 20$	$m = 100$
HasFather	99.9% / 99.9%	59.8% / 65.2%	100%	100%	100%	100%
HasSister	86.3% / 85.5%	59.8% / 66.4%	100%	100%	100%	100%
IsGrandparent	96.5% / 84.7%	97.7% / 63.7%	100%	100%	100%	100%
IsUncle	96.3% / 85.8%	96.0% / 64.0%	100%	100%	100%	100%
IsMGUncle	99.7% / 98.4%	98.4% / 81.7%	100%	100%	100%	100%
Graph	**MemNN**		**∂ILP**		**NLM (Ours)**	
	$m = 10$	$m = 50$	$m = 10$	$m = 50$	$m = 10$	$m = 50$
AdjacentToRed	95.2% / 94.6%	93.1% / 91.9%	100%	100%	100%	100%
4-Connectivity	92.3% / 90.5%	81.3% / 88.0%	100%	100%	100%	100%
6-Connectivity	67.6% / 58.8%	43.9% / 67.9%	100%	100%	100%	100%
1-OutDegree	99.8% / 99.7%	78.6% / 81.2%	100%	100%	100%	100%
2-OutDegree	81.4% / 61.8%	96.7% / 87.7%	N/A	N/A	100%	100%

used by Graves et al. (2016), and as a comparison, in the task of "finding" the MGUncle instead of "classifying", DNC reaches the accuracy of 81.8%.

3.3 General graph reasoning

We further extend the Family tree to general graphs and report the reasoning performance in Table 1.

We treat each nodes in the graph as an object (symbol). The (undirected) graph is input into the model in the form of a "Connected" relation between nodes. Besides, an extra property color represented by one-hot vectors is defined for every nodes. A node has the property of AdjacentToRed if it is connected with a red node. k-Connectivity is a relation between two nodes in the graph, which is true if two nodes are connected by a path with length at most k. A node has property k-OutDegree if its out-degree is k. Note that except for AdjacentToRed, all other properties or relations requires reasoning over more than 2 nodes. As an example, a human-written logic rule can be $\exists_b \exists_c \forall_d \texttt{Connected}(a, b) \wedge \texttt{Connected}(a, c) \wedge \neg \texttt{Connected}(a, d) \implies 2\texttt{-OutDegree}(a)$ where a, b, c and d are distinct nodes in the graph. ∂ILP fails to scale up to this task (Evans & Grefenstette, 2018).

All models are trained on instances of size 10 and tested on instances of size 10 and 50 (size is defined as the number of nodes in the graph).

3.4 Blocks World

We also test NLM's capability of decision making in the classic blocks world domain (Nilsson, 1982; Gupta & Nau, 1992) by slightly extending the model to fit the formulation of Markov Decision Process (MDP) in reinforcement learning.

Shown in Figure 1, an instance of the blocks world environment I contains two worlds: the initial world and the target world. Each world contains the ground and m blocks. The task is to take actions in the operating world and make its configuration same as the target world. Each object (blocks or ground) can be represented by four properties: world_id, object_id, coordinate_x, coordinate_y. The ground has a fixed coordinate $(0, 0)$. The input is the result of the numeral comparison among all pairs of objects (which may come from different worlds). Taking the x-coordinate as an example, the comparison produces three relations for each pairs of objects $(i, j), i \neq$

Table 2: Comparison between MemNN and the proposed NLM in the blocks world, sorting integers, and finding shortest paths, where m is the number of blocks in the blocks world environment or the size of the arrays/graphs in sorting/path environment. Both models are trained on instances of size $m \leq 12$ and tested on instances of size $m = 10$ and $m = 50$. The performance is evaluated by two metrics and separated by "/": the probability of completing the task during the test, and the average `Moves` used by the agents when they complete the task. There is no result for ∂ILP since it fails to scale up. MemNN fails to complete the blocks world within the maximum $m \times 4$ `Moves`.

Task	MemNN		NLM (Ours)	
	$m = 10$	$m = 50$	$m = 10$	$m = 50$
`BlocksWorld`	0% / N/A	0% / N/A	100% / 12	100% / 84
`Sorting`	100% / 22	90% / 986.6	100% / 8	100% / 45
`Path`	45% / 13.3	12% / 42.7	100% / 4	100% / 4

j: $\texttt{Left}(i, j)$ (whether i is on the left of j, or mathematically, $\mathbf{1}[x_i < x_j]$), $\texttt{SameX}(i, j)$ and $\texttt{Right}(i, j)$.

The only operation is $\texttt{Move}(i, j)$, which moves object i onto the object j in the operating world if the object i is movable and object j is placeable. If the operation is invalid, it will have no effect. In our setting, an object i is movable if and only if it is not the ground and there are no blocks on it ($\forall_j \neg(\texttt{Up}(i, j) \wedge \texttt{SameX}(i, j))$). An object i is placeable if and only if it is the ground or there are no blocks on it.

To avoid the ambiguity of the x-coordinates while putting blocks onto the ground, we set the x coordinate of block i to be i when it is placed onto the ground. The action space in the game is $(m + 1) \times m$ where m is the number of blocks in the world and the $+1$ comes from the "ground". For both MemNN and NLM, we apply a shared MLP on the output relational predicates of each pair of objects $O^{(2)}_{last}(x, y)$ and compute an action score $s(x, y)$. The probability for $\texttt{Move}(x, y)$ is $\propto \exp s(x, y)$ (by taking a `Softmax`). The results are summarized in Table 2. For more discussion on the confidence bounds of the experiments, please refer to Appendix B.6

3.5 General algorithms

We further show NLM's ability to excel at algorithmic tasks. We view an algorithm as a sequence of primitive actions and cast this as an reinforcement learning problem.

Sorting. We first consider the problem of sorting integers. Given a length-m array a of integers, the algorithm needs to iterative swap elements to sort the array in ascending order. We treat each slot in the array as an object, and input their index relations (whether $i < j$) and numeral relations (whether $a[i] < a[j]$) to NLM or MemNN. The action space is $m \times (m - 1)$ indicating the pair of integers to be swapped. Table 2 summarizes the learning performance.

As the comparisons between all pairs of elements in the array are given to the agent, sorting the array within the maximum number of swaps is an easy task. A trivial solution is to randomly swap an inversion [2] in the array at each step.

Beyond being able to generalize to arrays of arbitrary length, with different hyper-parameters and random seeds, the learned algorithms can be interpreted as Selection-Sort, Bubble-Sort, *etc*. We include videos demonstrating some learned algorithms in our website [3]

Path finding. We also test the performance of finding a path (single-source single-target path) in a given graph as a sequential decision-making problem. Given an undirected graph represented by its adjacency matrix as relations, the algorithm needs to find a path from a start node s (with property $\texttt{IsStart}(s) = \texttt{True}$) to the target node t (with property $\texttt{IsTarget}(t) = \texttt{True}$). To restrict the number of deduction steps, we set the maximum distance between s and t to be 5 during the training

[2] https://en.wikipedia.org/wiki/Inversion_(discrete_mathematics)

[3] https://sites.google.com/view/neural-logic-machines

and set the distance between s and t to be 4 during the testing, which replicates the setting of Graves et al. (2016). Table 2 summarizes the result.

`Path` task here can be seen as an extension of bAbI task 19 (path finding) (Weston et al., 2015) with symbolic representation. As a comparison with graph neural networks, Li et al. (2015) achieved 99% accuracy on the bAbI task 19. Contrastively, we formulate the shortest path task as a more challenging reinforcement learning (decision-making) task rather than a supervised learning (prediction) task as in Graves et al. (2016). Specifically, the agent iteratively choose the next node $next$ along the path. At the next step, the starting node will become $next$ (at each step, the agent will move to $next$). As a comparison, in Graves et al. (2016), Differentiable Neural Computer (DNC) finds the shortest path with probability 55.3% in a similar setting.

4 Related works and discussions

Classic ILP and relational reasoning. Inductive logic programming (ILP) (Muggleton, 1991; 1996; Friedman et al., 1999) is a paradigm for learning logic rules derived from a limited set of rule templates from examples. Being a powerful way of reasoning over discrete symbols, it gains its success in various language-related applications, and has been integrated into modern learning frameworks (Kersting et al., 2000; Richardson & Domingos, 2006; Kimmig et al., 2012). Recently, Evans & Grefenstette (2018) introduces a differentiable implementation of ILP which works with connectionist models such as CNNs. Sharing the similar spirit, Rocktäschel & Riedel (2017) introduces an end-to-end differentiable logic proving system for knowledge base (KB) reasoning. A major challenge of these approaches is to scale up to a large number of complex rules.

Shown in Section 2.3, both computational complexity and parameter size of our method grow *polynomially* w.r.t. the number of allowed predicates (in contrast to the *exponential* dependence in ∂ILP (Evans & Grefenstette, 2018)), but factorially w.r.t. the breadth (same as ∂ILP). Therefore, our method can deal with more complex tasks such as the blocks world which requires using a large amount of intermediate predicates, while ∂ILP fails to search in a such large space.

Our work is also related to symbolic relational reasoning, which has a wide application in processing discrete data structures such as knowledge graphs and social graphs (Zhu et al., 2014; Kipf & Welling, 2016; Zeng et al., 2016; Yang et al., 2017). Most symbolic relational reasoning approaches such as (Yang et al., 2017; Rocktäschel & Riedel, 2017) are developed for KB reasoning, in which the predicates on both sides of a rule is known in the KB. Otherwise, the complexity will grow exponentially large w.r.t. the number of used rules for a conclusion, which is the case of the blocks world. Moreover, Yang et al. (2017) $\texttt{query}(\texttt{Y}, \texttt{X}) \leftarrow \texttt{R}_\texttt{n}(\texttt{Y}, \texttt{Z}_\texttt{n}) \wedge \cdots \wedge \texttt{R}_1(\texttt{Z}_1, \texttt{X})$, which is not for general reasoning. The key of Rocktäschel & Riedel (2017) is to learn subsymbolic embeddings of entities and predicates for efficient KB completion, which cannot be easily adapted to general reasoning tasks.

Modular networks (Andreas et al., 2016a;b; Mascharka et al., 2018) are proposed for the reasoning over subsymbolic data such as images and natural language question answering. Santoro et al. (2017) implements a visual reasoning system based on "virtual" objects brought by receptive fields in CNNs. Wu et al. (2017) tackles the problem of deriving structured representation from raw pixel-level inputs. Dai et al. (2018) combines structured visual representation and theorem proving system.

Graph neural networks and relational inductive bias. Graph convolution networks (GCNs) (Bruna et al., 2013; Li et al., 2015; Defferrard et al., 2016; Kipf & Welling, 2016) is a family of neural architectures working on graphs. As a representative, Gilmer et al. (2017) proposed a message passing modeling for various graph neural networks and graph convolution networks. GCNs have achieved great success in tasks with intrinsic relational structures. However, most of the GCNs operate on pre-defined graphs and only exploit the embeddings of nodes and the binary connections. This restricts the expressiveness of models and results in inferior results on general purpose reasoning tasks (Li et al., 2015).

We move forward by removing such restrictions and introducing a neural architecture capturing logical predicates defined on any sets of objects. Together with the neural quantifiers, we show that our model can realize the FOPC. Quantitative results supports the advance of the proposed model in a broad set of tasks ranging from resolving relational reasoning to modeling general algorithms (as a

decision-making process). Moreover, being fully differentiable, NLMs can be easily plugged into existing convolutional or recurrent neural architectures for logic reasoning.

Object-oriented RL. Our logic-driven game playing falls into the track of Object-Oriented MDP (OO-MDP) (Diuk et al., 2008) which models the environment as a collection of objects and their relations. State transition and policies are both defined over objects and their interactions. OO-MDP enables structured modeling of environments (Kansky et al., 2017), structured task definition by object-oriented instructions (Denil et al., 2017), and structured policy learning (Garnelo et al., 2016).

Neural abstraction machines and program induction. Neural Turing Machine (NTM) (Graves et al., 2014; 2016) enables general purpose neural problem solving such as sorting by introducing an external memory that mimics the execution of Turing Machine. However, the systematical generalization of these models has not attracted much research effort. Neural programming (Neelakantan et al., 2015; Reed & De Freitas, 2015; Kaiser & Sutskever, 2015) is recently introduced to solve problems by synthesizing computer programs, and partially resolved the issue of systematical generalization by introducing extra supervision (Cai et al., 2017). In Chen et al. (2017), more complex programs such as language parsing are studied. However, the neural programming approaches are usually hard to optimize in an end-to-end manner or requires strong supervisions (such as ground-truth programs).

5 Conclusions and Discussions

In this paper, we proposed a novel neural-symbolic architecture called Neural Logic Machines (NLMs) which can conduct first-order logic deduction. Our model is fully differentiable, and can be trained in an end-to-end fashion. Empirical evaluations showed that our method is able to learn the underlying logical rules from small-scale tasks, and generalize to arbitrarily large-scale tasks.

The promising results open the door for several research directions. First, the maximum depth of the NLMs is a hyperparameter to be specified for individual problems. Future work may investigate how to extend the model, so that it can adaptively select the right depth for the problem at hand. Second, it is interesting to extend NLMs to handle vector inputs with real-valued components. Currently, NLM requires symbolic input that may not be easily available in applications like health care where many input (*e.g.*, blood pressure) are real numbers. Third, training NLMs remains nontrivial, and techniques like curriculum learning has to be used. It is important to find an effective yet simpler alternative to optimize NLMs. Last but not least, we are investigating the use of NLM in natural language applications, where symbolic reasoning naturally occurs.

REFERENCES

Jacob Andreas, Dan Klein, and Sergey Levine. Modular multitask reinforcement learning with policy sketches. *arXiv preprint arXiv:1611.01796*, 2016a.

Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. Neural module networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 39–48, 2016b.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *International Conference on Learning Representations*, 2015.

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pp. 41–48. ACM, 2009.

Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. Spectral networks and locally connected networks on graphs. *arXiv preprint arXiv:1312.6203*, 2013.

Jonathon Cai, Richard Shin, and Dawn Song. Making neural programming architectures generalize via recursion. *arXiv preprint arXiv:1704.06611*, 2017.

Xinyun Chen, Chang Liu, and Dawn Song. Learning neural programs to parse programs. *arXiv preprint arXiv:1706.01284*, 2017.

Wang-Zhou Dai, Qiu-Ling Xu, Yang Yu, and Zhi-Hua Zhou. Tunneling neural perception and logic reasoning through abductive learning. *arXiv preprint arXiv:1802.01173*, 2018.

Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Computing Surveys*, 33:374–425, 2001.

Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in Neural Information Processing Systems*, pp. 3844–3852, 2016.

Misha Denil, Sergio Gómez Colmenarejo, Serkan Cabi, David Saxton, and Nando de Freitas. Programmable agents. *arXiv preprint arXiv:1706.06383*, 2017.

Carlos Diuk, Andre Cohen, and Michael L Littman. An object-oriented representation for efficient reinforcement learning. In *Proceedings of the 25th international conference on Machine learning*, pp. 240–247. ACM, 2008.

Richard Evans and Edward Grefenstette. Learning explanatory rules from noisy data. *Journal of Artificial Intelligence Research*, 61:1–64, 2018.

Jerry Fodor and Brian P McLaughlin. Connectionism and the problem of systematicity: Why smolensky's solution doesn't work. *Cognition*, 35(2):183–204, 1990.

Jerry A Fodor and Zenon W Pylyshyn. Connectionism and cognitive architecture: A critical analysis. *Cognition*, 28(1-2):3–71, 1988.

Nir Friedman, Lise Getoor, Daphne Koller, and Avi Pfeffer. Learning probabilistic relational models. In *IJCAI*, volume 99, pp. 1300–1309, 1999.

Marta Garnelo, Kai Arulkumaran, and Murray Shanahan. Towards deep symbolic reinforcement learning. *arXiv preprint arXiv:1609.05518*, 2016.

Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. *arXiv preprint arXiv:1704.01212*, 2017.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471, 2016.

Naresh Gupta and Dana S Nau. On the complexity of blocks-world planning. *Artificial Intelligence*, 56(2-3):223–254, 1992.

Robert F Hadley. Systematicity in connectionist language learning. *Mind & Language*, 9(3):247–272, 1994.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4700–4708, 2017.

Peter A Jansen and Scott Watter. Strong systematicity through sensorimotor conceptual grounding: an unsupervised, developmental approach to connectionist sentence processing. *Connection Science*, 24(1):25–55, 2012.

Łukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms. *arXiv preprint arXiv:1511.08228*, 2015.

Ken Kansky, Tom Silver, David A Mély, Mohamed Eldawy, Miguel Lázaro-Gredilla, Xinghua Lou, Nimrod Dorfman, Szymon Sidor, Scott Phoenix, and Dileep George. Schema networks: Zero-shot transfer with a generative causal model of intuitive physics. *arXiv preprint arXiv:1706.04317*, 2017.

Kristian Kersting, Luc De Raedt, and Stefan Kramer. Interpreting bayesian logic programs. In *Proceedings of the AAAI-2000 workshop on learning statistical models from relational data*, pp. 29–35, 2000.

Angelika Kimmig, Stephen Bach, Matthias Broecheler, Bert Huang, and Lise Getoor. A short introduction to probabilistic soft logic. In *Proceedings of the NIPS Workshop on Probabilistic Programming: Foundations and Applications*, pp. 1–4, 2012.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pp. 1097–1105, 2012.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

David Mascharka, Philip Tran, Ryan Soklaski, and Arjun Majumdar. Transparency by design: Closing the gap between performance and interpretability in visual reasoning. *arXiv preprint arXiv:1803.05268*, 2018.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 02 2015. URL http://dx.doi.org/10.1038/nature14236.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pp. 1928–1937, 2016.

Stephen Muggleton. Inductive logic programming. *New generation computing*, 8(4):295–318, 1991.

Stephen Muggleton. Stochastic logic programs. *Advances in inductive logic programming*, 32: 254–264, 1996.

Arvind Neelakantan, Quoc V Le, and Ilya Sutskever. Neural programmer: Inducing latent programs with gradient descent. *arXiv preprint arXiv:1511.04834*, 2015.

Nils J Nilsson. *Principles of Artificial Intelligence*. Springer Science & Business Media, 1982.

Scott Reed and Nando De Freitas. Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279*, 2015.

Matthew Richardson and Pedro Domingos. Markov logic networks. *Machine learning*, 62(1-2): 107–136, 2006.

Tim Rocktäschel and Sebastian Riedel. End-to-end differentiable proving. In *Advances in Neural Information Processing Systems*, pp. 3791–3803, 2017.

Adam Santoro, David Raposo, David G Barrett, Mateusz Malinowski, Razvan Pascanu, Peter Battaglia, and Tim Lillicrap. A simple neural network module for relational reasoning. In *Advances in neural information processing systems*, pp. 4974–4983, 2017.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

Sainbayar Sukhbaatar, arthur szlam, Jason Weston, and Rob Fergus. End-to-end memory networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett (eds.), *Advances in Neural Information Processing Systems 28*, pp. 2440–2448. Curran Associates, Inc., 2015. URL http://papers.nips.cc/paper/5846-end-to-end-memory-networks.pdf.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger (eds.), *Advances in Neural Information Processing Systems 27*, pp. 3104–3112. Curran Associates, Inc., 2014. URL http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pp. 6000–6010, 2017.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems*, pp. 2692–2700, 2015.

Jason Weston, Antoine Bordes, Sumit Chopra, Alexander M Rush, Bart van Merriënboer, Armand Joulin, and Tomas Mikolov. Towards ai-complete question answering: A set of prerequisite toy tasks. *arXiv preprint arXiv:1502.05698*, 2015.

Ronald J Williams and Jing Peng. Function optimization using connectionist reinforcement learning algorithms. *Connection Science*, 3(3):241–268, 1991.

Jiajun Wu, Joshua B Tenenbaum, and Pushmeet Kohli. Neural scene de-rendering. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 699–707, 2017.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

Fan Yang, Zhilin Yang, and William W Cohen. Differentiable learning of logical rules for knowledge base reasoning. In *Advances in Neural Information Processing Systems*, pp. 2316–2325, 2017.

Wenyuan Zeng, Yankai Lin, Zhiyuan Liu, and Maosong Sun. Incorporating relation paths in neural relation extraction. *arXiv preprint arXiv:1609.07479*, 2016.

Yuke Zhu, Alireza Fathi, and Li Fei-Fei. Reasoning about Object Affordances in a Knowledge Base Representation. In *European Conference on Computer Vision*, 2014.

Supplementary Material for Neural Logic Machines

This supplementary material is organized as follows. First, we provide more details for our training method and introduce the curriculum learning used for reinforcement learning tasks in Appendix A. Second, in Appendix B, we provide more implementation details and hyper-parameters of each task in Section 3. Afterwards, we provide deferred discussion of NLM extensions in Appendix C. Finally, we also associate visualization demo videos in the supplementary to illustrate the capability of our trained NLM models on the sorting, shortest path and blocks world tasks. We also provide a minimal implementation of NLM in TensorFlow for reference at the end of the supplementary material (Appendix D).

A Training Method and Curriculum Learning

In this section, we provide hyper-parameters details of our training method and introduce the exam-guided curriculum learning used for reinforcement learning tasks. We also provide the details of data generation.

A.1 Training method

We optimize both NLM and MemNN with Adam Kingma & Ba (2014) and use a learning rate of $\alpha = 0.005$.

For all supervised learning tasks (i.e. family tree and general graph tasks), we use Softmax-Cross-Entropy as loss function, and use a batch size of 4.

For reinforcement learning tasks (i.e. the blocks world, sorting and shortest path tasks), We use REINFORCE Sutton & Barto (1998) algorithm for optimization. Each training batch is composed of a single episode of play. Similar to A3C Mnih et al. (2016), we add policy entropy term in the objective function (proposed by Williams & Peng (1991)) to help exploration. The update function for parameters θ of policy π is

$$\Delta\theta = \alpha[v_t \nabla_\theta \log \pi(a_t|s_t;\theta) + \beta \nabla_\theta H(\pi(s_t;\theta))],$$

where H is the entropy function, s_t and a_t are the state and action at time t, v_t is the discounted reward starting from time t. The hyper-parameter β is set according to different environments and learning stages depending on the demand of exploration.

In all environments, the agent receives a reward of value 1.0 when it completes the task within a limited number of steps (which is related to the number of objects). To encourage the agent to use as few moves as possible, we give a reward of -0.01 for each move. The reward discount factor γ is 0.99 for all tasks.

Table 3: Hyper-parameters for reinforcement learning tasks. The meaning of the hyper-parameters could be found in Section A.1 and Section A.2. For the `Path` environment, the step limit is set to the actual distance between the starting point and the targeting point, to encourage the agents to find the shortest path.

Task	Range	Step Limit	β_{init}	Ω	Epochs	Train Epoch Episodes	Evaluation Episodes
`Sorting`	$m \in [4, 10]$	$2m$	0.01	0.5	5	200	200
`Path`	$m \in [3, 12]$	*opt*	0.1	0.5	40	600	3000
`Blocks World`	$m \in [2, 12]$	$4m$	0.2	0.6	50	1000	3000

A.2 Curriculum learning guided by exams and fails

Inspired by the education system of humans, we employ a exam-guided curriculum learning (Bengio et al., 2009) approach for training Neural Logic Machines. We heuristically label each training sample with its complexity. Training samples are grouped by their complexity (as *lessons*). For example, in

Algorithm 1: Curriculum learning guided by exams and fails

```
Function train(model M, lessons 𝓛):
    for ℓ ∈ 𝓛 do
        for i = 0, 1, ··· ℓ.max_epochs do
            accuracy, pos, neg ← evaluate(M, ℓ);        // Take the exam and collect samples.
            if accuracy > ℓ.threshold then
                break;                                  // Enter the next lesson if pass the exam.
            for j = 0, 1, ··· K do
                data ∼ balanced sampling from pos and neg;
                Optimize M with data;
```

the game of BlocksWorld, we can consider the number of blocks in a game instance as its complexity. During the training, we organize the samples given to the model in an order and illustrates gradually more complex ones. We periodically test models' performance on novel samples with the same complexity as the ones in its recent lessons (*exams*). The well-performed model (whose accuracy is above a certain threshold) will *pass* the exam and receive a new lesson (of harder training samples). The exam-guided curriculum learning exploits the previously learned knowledge to ease the learning of more complex samples. Moreover, the performance on the final exam above a threshold indicates the *graduation* of models.

Each lesson contains the example with same number of objects in our experiments. For example, the first lesson in the blocks world contains all possible instances containing 2 blocks (in each world). The instances in second lesson contain 3 blocks in each world. And in the last lesson (totally 11 lessons) there are 12 blocks in each world. We report the range of the curriculum in Table 3 for three tasks.

Another essential ingredient for the efficient training of NLMs is the recording of models' failure cases. Specifically, we keep track of two sets of training samples: positive and negative. During the exam, the samples successfully finished by the model will be added to the positive set while the ones that the model fails to finish the task will be added to the negative set. All training samples are sampled from the positive set with probability Ω and from the negative set with probability $1 - \Omega$. This balanced sampling prevents models from getting stuck at sub-optimal solutions. Algorithm 1 illustrates the pseudo-code of the curriculum learning guided by exams and fails.

The evaluation process ("exam") randomly samples examples from 3 recent lessons. The agent goes through these examples and gets the success rate (finishing the task means success) as its performance, which is used to judge that whether the agent passes the exam by comparing to a lesson-related threshold. As we want a perfect model, the threshold of the last lesson ("final exam") is 100%. We linearly decrease the threshold by 0.5% for each former lessons, to prevent over-fitting(*e.g.*, the threshold of the first lesson in the blocks world is 95%). After the "exam", the examples are collected into positive and negative pools according to the outcome (success or not). During the training, we use balanced sampling for choosing training examples from positive and negative pools with probability Ω from positive. The hyper-parameters Ω, the number of epochs, the number of episodes in each training epoch and the number of episodes in one evaluation are shown in Table 3 for three tasks.

B Implementation details and hyper-parameters

This section provides more implementation details for the model and experiments, and summarizes the hyper-parameters used in experiments for our NLM and the baseline algorithm MemNN.

B.1 Residual connection.

Analog to the residual link in (He et al., 2016; Huang et al., 2017), we add residual connections to our model. Specifically, for each layer illustrated in Figure 2, the base predicates (inputs) are concatenated to the conclusive predicates (outputs) group-wisely. That is, input unary predicates are

concatenated to the deduced unary predicates while input binary predicates are concatenated to the conclusive binary predicates.

B.2 HYPER-PARAMETERS FOR NLM

Table 4 shows hyper-parameters used by NLM for different tasks. In supervised learning tasks, a model is called "graduated" if its training loss is below a threshold depending on the task (usually 1e-6). In reinforcement learning tasks, an agent is called "graduated" if it can pass the final exam, i.e., get 100% success rate on the evaluation process of the last lesson.

We note that in the random generated cases, the number of maternal great uncle (`IsMGUncle`) relation is relatively small. This make the learning of this relation hard and results in a graduation ratio of only 20%. If we increase the maximum number of people in training examples to 30, the graduation ratio will grows 50%.

Table 4: Hyper-parameters for Neural Logic Machines. The definition of depth and breadth are illustrated in Figure 2. "Res." refers to the use of residual links. "Grad." refers to the ratio of successful graduation in 10 runs with different random seeds, which partially indicates the difficulty of the task. "Num. Examples/Episodes" means the maximum number of examples/episodes used to train the model in supervised learning and reinforcement learning cases.

	Tasks	Depth	Breath	Res.	Grad.	Num. Examples/Episodes
Family Tree	`HasFather`	4	3	×	100%	50,000 examples
	`HasSister`	4	3	×	100%	50,000 examples
	`IsGrandparent`	4	3	×	100%	100,000 examples
	`IsUncle`	4	3	×	90%	100,000 examples
	`IsMGUncle`	4	3	×	20%	200,000 examples
General Graph	`AdjacentToRed`	4	3	×	90%	100,000 examples
	`4-Connectivity`	4	3	×	100%	50,000 examples
	`6-Connectivity`	8	3	✓	60%	50,000 examples
	`1-OutDegree`	4	3	×	100%	50,000 examples
	`2-OutDegree`	5	4	✓	100%	100,000 examples
General Algorithm	`Sorting`	3	2	✓	100%	1,000 episodes
	`Path`	5	3	✓	60%	24,000 episodes
`BlocksWorld`		7	3	✓	40%	50,000 episodes

B.3 HYPER-PARAMETERS FOR MEMNN

We set the number of iters/episodes used for baseline algorithms to be same as NLM. For the memory networks, each pre-condition in the memory is embedded into a key space and a value space. The dimensions of the spaces are 16 and 32 respectively. The hidden size of the LSTM in MemNN is 64. The number of queries is set to be 4 across all tasks (except that the `Sorting` task uses 1 query only). Empirically, we search for the optimal hyper-parameters but find that they has little effect on the performance.

B.4 DATA GENERATION

We use random generation to generate training and testing data. more details and specific parameters used to generate the data could be found in our open source code.

In family tree tasks, we mimic the process of families growing using a time-line. For each new created person, we randomly sample the gender and parents (could be none, indicating not included in the family tree) of the person. We also maintain lists of singles of each gender, and randomly pick

two from each list to be married (each time when a person was created). We randomly permute the order of people.

In general graph tasks (include `Path`), We adopt the generation method from Graves et al. (2016), which samples m nodes on a unit square, and the out-degree k_i of each node is sampled. Then each node connects to k_i nearest nodes on the unit square. In undirected graph cases, all generated edges are regarded as undirected edges.

In `Sorting`, we randomly generate permutations to be sorted in ascending order.

In `Blocks World`, We maintain a list of placeable objects (the ground included). Each new created block places on one randomly selected placeable object. Then we randomly shuffle the id of the blocks.

B.5 BLOCKS WORLD

In the blocks world environment, to better aid the reinforcement learning process, we train the agent on an auxiliary task, which is to predict the validity of the actions. This task is trained by supervised learning using cross-entropy loss. The overall is a sum of this loss (with a weight of 0.1) and the loss of REINFORCE.

We did not choose the `Move` to be taken directly based on the relational predicates at the last layer of NLM. Instead, we manually concatenate the object representation from the current and the target configuration, but having the same object ID. Then for each pair of objects, their relation representation are constructed by the concatenation of their own object representation. An extra fully-connected layer is applied to the relational representation, followed by a `Softmax` layer over all pairs of objects. We choose action based on the Softmax score.

B.6 ACCURACY DISCUSSION

We cannot directly prove the accuracy of NLM by looking at the induced rules as in traditional ILP systems. Alternatively, we take an empirical way to estimate its accuracy by sampling testing examples. Throughout the experiments section, all accuracy statistics are reported on 1000 random generated data.

To show the confidence of this result, we test a specific trained model of `Blocks World` task with 100,000 samples. We get *no* fail cases in the testing. According to the multiplicative form of Chernoff Bound [4], We are 99.7% confident that the accuracy is at least 99.98%.

C NEURAL LOGIC MACHINES (NLM) EXTENSIONS

Reasoning over noisy input: integration with neural perception. Recall that NLM is fully differentiable. Besides taking logic pre-conditions (binary values) as input, the input properties or relations can be derived from other neural architectures (*e.g.*, CNNs). As a preliminary example, we replace input properties of nodes with images from the MNIST dataset. A convolutional neural network (CNN) is applied to the input extracting multiple features for future reasoning. CNN and NLM can be optimized jointly. This enables reasoning over noisy input.

We modify the `AdjacentToRed` task in general graph reasoning to `AdjacentToNumber0`. In detail, each node has a visual input from the MNIST dataset indicating its number. We say `AdjacentToNumber0`(x) if and only if a node x is adjacent to another node with number 0. We use LeNet LeCun et al. (1998) to extract visual features for recognizing the number of each node. The output of LeNet for each node is a vector of length 10, with sigmoid activation.

We follow the train-test split from the original MNIST dataset. The joint model is trained on 100,000 training examples ($m = 10$) and gets 99.4% accuracy on 1000 testing examples ($m = 50$). Note that the LeNet modules are optimized jointly with the reasoning about `AdjacentToNumber0`.

Recurrent reasoning. In NLM, all logic deductions are performed by the forward propagation of the neural architecture. As discussed in Section 2.3, the number of deduction steps (the depth) restricts

[4] https://en.wikipedia.org/wiki/Chernoff_bound#Multiplicative_form_(relative_error)

the expressiveness of NLM, especially when the task requires recurrent reasoning. We preliminarily study a recurrent extension of NLM in the graph connectivity task (whether two nodes are connected in the given graph).

In the recurrent version of NLM, the weights of the neural boolean logic modules at depth i are shared with the neural boolean logic modules at depth $i + k$, where $k = 2$ is the length of the recurrent period. The depth (number of steps) of the recurrent deduction is manually set according to the size of the graph (ensure that $2^{rollouts} \geq m$). We leave how the deduction depth (number of steps) can be automatically computed as a future work. We use the graphs with the number of nodes m ranging from 2 to 10 as the training data. We use graphs with size 50 as the testing data. The recurrent version of NLM is trained on 50,000 training examples and gets 100% accuracy on 1000 random generated testing examples. The hyper-parameters for the model (except the depth) are set to be the same as the `4-connectivity` task.

D IMPLEMENT NLM IN TENSORFLOW

The following python code contains a minimal implementation for one Neural Logic Machines layer with breadth equals 3 in TensorFlow. The `neural_logic_layer_breath3` is the main function. The syntax is highlighted and is best viewed in color.

```
from itertools import permutations
import tensorflow as tf
from tensorflow.layers import dense

def expand(input, M):
  """Expands input at its second last dimension (e.g., [B, ..., Ni, Nj] to [B, ..., Ni, M, Nj]) by replicating tensors."""
  ndims = input.get_shape().ndims + 1
  multiples = [M if i == ndims - 2 else 1 for i in range(ndims)]
  return tf.tile(tf.expand_dims(input, -2), multiples)

def reduce(input, M):
  """Reduces max and min at the second last dimension, except for diagonal elements."""
  mask = _reduce_mask(input, M)[tf.newaxis, ..., tf.newaxis]
  return tf.concat([
    tf.reduce_max(input * mask, -2),
    tf.reduce_min(input * mask + (1 - mask), -2)
  ], -1)

def neural_logic(input, hidden_dim):
  """An MLP layer applied on permutations of the input."""
  return dense(_input_permutations(input), hidden_dim, activation=tf.sigmoid)

def neural_logic_layer_breath3(input0, input1, input2, input3, M, hidden_dim, residual):
  """A neural logic layer with breath 3.
  Args:
    input0: float Tensor of shape [B, hidden_dim], nullary predicates.
    input1: float Tensor of shape [B, M, hidden_dim], unary predicates.
    input2: float Tensor of shape [B, M, M, hidden_dim], binary predicates.
    input3: float Tensor of shape [B, M, M, M, hidden_dim], tenary predicates.
```

```
    M: int, number of objects.
    hidden_dim: int, hidden dimension.
    residual: boolean, use the residual link or not.
  Returns:
    4 float Tensors, output nullary, unary, binary tenary predicates respectively.
  """
  agg0 = tf.concat([input0, reduce(input1, M)], -1)
  agg1 = tf.concat([input1, expand(input0, M), reduce(input2, M)], -1)
  agg2 = tf.concat([input2, expand(input1, M), reduce(input3, M)], -1)
  agg3 = tf.concat([input3, expand(input2, M)], -1)
  outputs = [neural_logic(x, hidden_dim) for x in [agg0, agg1, agg2, agg3]]
  if residual:
    outputs = [tf.concat([x, y], -1) for x, y in zip(outputs, [input0, input1, input2, input3])]
  return outputs

def _reduce_mask(input, M):
  dimension = input.get_shape().ndims - 2
  base = 1.0 - tf.eye(M)
  if dimension < 2: return tf.constant(1.0)  # Identity.
  elif dimension == 2: return base # Diagonal excluded.
  elif dimension == 3: return tf.expand_dims(base, 2) * tf.expand_dims(base, 1) * tf.expand_dims(base, 0)  # Mask out all tuples (x, y, z) that x == y or y == z or z == x.
  else: raise NotImplementedError()

def _input_permutations(input):
  dimension = input.get_shape().ndims - 2
  if dimension < 2: return input
  else: return tf.concat([
    tf.transpose(input, [0] + list(perm) + [1 + dimension])
    for perm in permutations(range(1, 1 + dimension))
  ], -1)
```