Compositional Neural Logic Programming

Son N. Tran
University of Tasmania
sn.tran@utas.edu.au

Abstract

This paper introduces Compositional Neural Logic Programming (CNLP), a framework that integrates neural networks and logic programming for symbolic and sub-symbolic reasoning. We adopt the idea of compositional neural networks to represent first-order logic predicates and rules. A voting backward-forward chaining algorithm is proposed for inference with both symbolic and sub-symbolic variables in an argument-retrieval style. The framework is highly flexible in that it can be constructed incrementally with new knowledge, and it also supports batch reasoning in certain cases. In the experiments, we demonstrate the advantages of CNLP in discriminative tasks and generative tasks.

1 Introduction

The integration of neural networks and symbolic knowledge has been an important research topic in artificial intelligence for decades. The combination of connectionist models for low-level information processing and logic programs for high-level decision making can offer improvements in inference efficiency and prediction performance [Towell and Shavlik, 1994; Serafini and d'Avila Garcez, 2016; Cohen *et al.*, 2017; Tran and d'Avila Garcez, 2018; Riveret *et al.*, 2020; Yang *et al.*, 2017]. However, the co-existence of symbolic and sub-symbolic variables makes it challenging for the reasoning. Although sub-symbolic variables can be incorporated nicely in neuro-symbolic systems to infer the truth values of symbolic variables [Donadello *et al.*, 2017; Manhaeve *et al.*, 2018], there is still a question of how we can achieve general reasoning where any variables, either they are symbolic or sub-symbolic, can be inferred. For example, we would like to have a reasoning algorithm to answer not only a question $+$ $=?$ but also $+$ $= 8$. Such type of reasoning would provide great flexibility to improve the learning and inference for both discriminative and generative tasks.

It has been shown that machine reasoning can be done by manipulating knowledge acquired from learnable models, such as neural networks [Bottou, 2014]. The composition of neural networks can offer modularity, explainability, and recursion [Wang *et al.*, 2019; Pierrot *et al.*, 2019], and therefore, is well suited for symbolic knowledge representation and reasoning. In this paper, we present a compositional approach to integrate neural networks and symbolic knowledge for effective learning and efficient reasoning. In particular, we propose *Compositional Neural Logic Programming* framework (CNLP), a neural network-based structure consisting of different sub-networks, called here as *neural predicates*, each represents a first-order logic predicate in a knowledge base (KB). In this work, we introduce two types of neural predicates, namely symbolic neural predicate and compositional neural predicate. A symbolic neural predicate is an auto-encoder $\mathcal{N}_{\mathsf{P}}^{AE}$ representing an M-arity (logic) predicate P of only symbolic arguments. The auto-encoder is constructed using facts from the predicate. A compositional neural predicate is a combination of discriminative and generative networks to present an $(M+N)$-arity predicate P of N tensor arguments and M symbolic arguments. We use the term "tensor" to refer to sub-symbolic representation, such as real-valued scalars, vectors, matrices, and multi-dimensional arrays, etc.. In CNLP, neural predicates are chained through their shared variables to compose a network for logic rules.

For reasoning, we propose a *voting backward-forward chaining* algorithm to iteratively select and infer the neural predicates to deliver the answer for an argument-retrieval query. The algorithm starts with the head predicate of a rule and goes backward to work recursively on the predicates in the body of the rule. During the process, each of those predicates is voted to perform inference and the returned results are forwarded to infer the unassigned variables of other predicates, and subsequently, the head predicate. To this end, we design a voting mechanism to prioritise the neural predicates based on their types and the number of unassigned variables left to infer, so as to reduce the search complexity. The key advantage of CNLP is the flexibility that offers three benefits. First, the compositional architecture of CNLP allows it to represent new knowledge from existing knowledge incrementally. Second, it supports reasoning with both symbolic variables and tensor variables. Third, as a neural network, CNLP can perform batch learning, as well as batch reasoning in certain cases. In the experiments, we show the advantage of CNLP in both discriminative and generative tasks.

Related work. Reasoning with symbolic and sub-symbolic variables has been studied in Logic Tensor Network (LTN) [Serafini and d'Avila Garcez, 2016; Donadello *et al.*, 2017] and DeepProbLog [Manhaeve *et al.*, 2018]. LTN employs neural networks to represent truth values of predicates and combines them using fuzzy logic connectives to express the degree of truth of a logic formula. To infer a variable, LTN needs to search for the assignments that satisfy the formula, i.e. those have a high degree of truth. Different from that, CNLP directly infers the values of the variables via a chaining procedure, and therefore is easier to work on a knowledge base with recursive rules. In DeepProbLog [Manhaeve *et al.*, 2018], deep neural networks are integrated to produce neural predicates for the probabilistic logic programming system. However, DeepProbLog needs to ground the logic program for each query, unlike CNLP where batch computation can be applied for learning and reasoning. Furthermore, although both LTN and DeepProbLog allow real-valued objects to be fed into the systems the reasoning process still focuses on inferring symbolic variables. As far as we know CNLP is the first neuro-symbolic system that demonstrates the ability to reason both symbolic and sub-symbolic variables. The compositional structure of CNLP is inspired by the idea of embedding subprograms to compose new programs in Neural Programmer-Interpreters (NPI) [Reed and de Freitas, 2016] and AlphaNPI [Pierrot *et al.*, 2019]. However, CNLP is distinct in that it models the logical relations of subprograms, known as predicates and rules in the logic world.

2 Compositional Neural Logic

2.1 Overview

A first-order knowledge base (KB) is a set of facts and rules. A fact is the grounding atom of a predicate from a substitution θ. For example, $\mathsf{P}(x, y)$ is the fact from $\mathsf{P}(X, Y)$ when applying the substitution $\theta = \{X/x, Y/y\}$ that maps the variables X, Y to the symbolic objects x, y respectively. A first-order logic rule $\mathsf{P}_{head}(.) \leftarrow \mathsf{P}_1(.) \wedge \mathsf{P}_2(.) \wedge ... \wedge \mathsf{P}_B(.)$ consists of a *head predicate* P_{head} and a list of *body predicates* P_b, $b \in \{1, ..., B\}$. We use $\mathsf{P}(.)$ to denote a predicate of any arity. P_b is an *intermediate predicate* if it is also the head predicate of other rules. A rule is recursive if its head predicate also appears in its body. This predicate is known as a *recursive predicate*. Finally, we denote a predicate which only appears in the bodies of rules as *in-body predicate*. We can query a knowledge base to validate a fact, e.g. whether $\mathsf{P}(x, y)$ is $True$ or $False$, or to find all possible values of the unassigned variables that satisfy the query. The latter is known as argument-retrieval query. For example, a query $\mathsf{P}(X = x, Y)$ (or $\mathsf{P}(x, Y)$ for short) would ask for the values y of the unassigned variable Y that make $\mathsf{P}(X = x, Y = y) \equiv True$, given the assignment $X = x$.

In this paper, we generalise the first-order knowledge base to include real-valued objects in addition to symbolic objects. A real-valued object can be a scalar (e.g. velocity of a car), vector (e.g. a word embedding), matrix (e.g. an image), etc.. In what follows we will introduce compositional neural logic, a method to represent the facts and rules of the generalised knowledge base as a set of neural networks, one connects to others through shared variables. We use tensors to represent the objects and neural networks to represent the predicates. Such neural networks are called *neural predicates*. A *neural rule* is a group of neural networks, each represents a predicate in the body of the rule.

2.2 Neural Predicates

Let us denote $\mathcal{C}^{\mathcal{S}}$ as a set of classes of symbolic objects and $\mathcal{C}^{\mathcal{T}}$ as a set of classes of real-valued objects. An i-th symbolic object of class c-th ($C^{\mathcal{S}}_c$) in $\mathcal{C}^{\mathcal{S}}$ is denoted as o^c_i. Its grounding is an one-hot vector $\mathbf{o}^c_i \in \{0, 1\}^{|C^{\mathcal{S}}_c|}$, where $\mathbf{o}^c_i[i] = 1$ and $\mathbf{o}^c_i[i'] = 0$ for all $i' \neq i$. A real-valued object of a class $C^{\mathcal{T}}_k \in \mathcal{C}^{\mathcal{T}}$ is a tensor, for example an audio segment $\in \mathbb{R}^{45056}$, a gray-scale image of a hand written digit $\in \mathbb{R}^{28\times 28}$, or a natural image of a car $\in \mathbb{R}^{32\times 32\times 3}$, etc..

Definition 1. *A neural predicate is a neural network $\mathcal{N}_{\mathsf{P}}$ representing a predicate P to form an atom $\mathsf{P}(T_1, ..., T_N, S_1, ..., S_M)$ of mixed variables. Here, T_n ($n = 1, .., N$) is a tensor variable for the real-valued objects of class $C^{\mathcal{T}}_{k_n}$ and S_m ($m = 1, ..., M$) is a symbolic variable for the symbolic objects of class $C^{\mathcal{S}}_{c_m}$.*

All variables in a neural predicate (NP) can be inferred, i.e. given an assignment of a subset of the variables, we would like to infer the values of unassigned variables. However, the realization of this neural predicate is non-trivial. Therefore, in this paper we consider two special cases: a neural predicate of mixed variables where only one variable can be inferred at a time; and a neural predicate of all symbolic variables where multiple variables can be inferred at a time.

Symbolic Neural Predicates

A symbolic neural predicate (SNP) is a neural network presenting an M-arity predicate P of only symbolic variables $(S_1, ..., S_M)$. Such neural predicate can be modelled by a generalised version of the Neural Tensor Network [Socher *et al.*, 2013], as in LTN [Serafini and d'Avila Garcez, 2016], where the network is trained from samples of positive and negative facts. However, similar to [Towell and Shavlik, 1994], we are interested in encoding facts and rules from a knowledge base into a network's structure. Therefore, we propose a method to construct an auto-encoder $\mathcal{N}^{AE}_{\mathsf{P}}$ from all facts of a predicate without the need for learning. In this auto-encoder, each variable S_m is represented as a pair of (input,output) vectors ($\mathbf{s}_m$,$\hat{\mathbf{s}}_m$) to perform a transformation $\mathbf{s}_1, ..., \mathbf{s}_M \overset{\mathcal{N}^{AE}_P}{\mapsto} \hat{\mathbf{s}}_1, ..., \hat{\mathbf{s}}_M$. $\mathcal{N}^{AE}_P$ is characterised by a set of binary weight matrices $\{W^m \in \{0, 1\}^{|C^{\mathcal{S}}_{c_m}| \times J} | m = 1, .., M\}$, with J is the number of hidden units. To integrate the facts, we initialise the weights to 0s, and for each fact j-th $\mathsf{P}(o^{c_1}_{i_1}, ..., o^{c_M}_{i_M})$ we set $w^1_{i_1 j} = ... = w^M_{i_M j} = 1$.

Given an assignment of the symbolic variables $S_m = o^{c_m}_{i_m} \in \mathcal{C}^{S}_{c_m}$, we infer the unassigned variables $S_{\hat{m}}$ by using the symbolic neural predicate to compute $\hat{\mathbf{s}}_{\hat{m}}$, with $\mathbf{s}_m = \mathbf{o}^{c_m}_{i_m}$ (the grounding of $o^{c_m}_{i_m}$) and $\mathbf{s}_{\hat{m}} = \{0\}^{|C^{\mathcal{S}}_{c_{\hat{m}}}|}$ (a vector of zeros) as inputs. Such inference of the unassigned variables are carried out by the encoding-decoding mechanism. The encoding step computes the hidden state $\mathbf{h} = g(\boldsymbol{\alpha})$, where:

$$\alpha_j = (\sum_{m=1}^{M} \mathbf{s}_m \mathbf{w}_j^m \geq \sum_m \mathbf{s}_m \{1\}^{|C_{c_m}^{\mathcal{S}}|^\top}) \tag{1}$$

is the Boolean encoding for the hidden unit j. Here, $\mathbf{w}_j^m$ is column j of W^m and $\{1\}^{|C_{c_m}^{\mathcal{S}}|}$ is a vector of all ones. $\alpha_j \equiv 1$ ($True$) indicates that there is a fact that matches the assigned variables from which we can find the values for the unassigned variables. If none of the Boolean encodings is $True$, then no answers are given (unable to infer) and $\mathbf{h} = \boldsymbol{\alpha}$ is a zero vector. In the other case, there can be more than one answer for the unassigned variables, and depending on different circumstances that we want to get all the answers or just one sample, as follows.

Expandable batch inference. The number of $True$ values (1s) in the hidden layer corresponds to the number of possible answers for the unassigned variables. For efficiency, we need to expand the hidden state vector into a matrix for batch inference. This can be done by replicating the vector and multiplying it with an identity matrix $I \in \mathbb{R}^{J \times J}$ element-wise, before pruning the rows with all 0s elements, as $\mathbf{h} = g(\boldsymbol{\alpha}) = \text{prune-0s-rows}(\text{replicate}(\boldsymbol{\alpha}) \odot I)$.

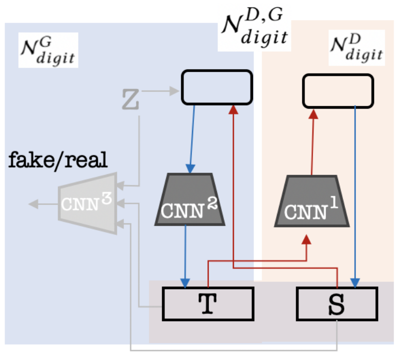
Sampling. The hidden layer can be treated as a softmax layer where only one $True$ hidden unit is sampled while the others will be set as $False$ (0). In this case, $\mathbf{h} = g(\boldsymbol{\alpha}) = \text{sample}(\text{soft-max}(\boldsymbol{\alpha}))$.

Both the encoding strategies above can be extended for batch inference, i.e. multiple inferences at the same time. After the encoding step, the decoding step is simply a linear transformation $\hat{\mathbf{s}}_{\hat{m}} = f^{\hat{m}}(\mathbf{h}) = \mathbf{h} W^{\hat{m}\top}$.

Compositional Neural Predicate

A compositional neural predicate (CNP) is a group of neural networks to present a predicate P of both symbolic variables $\mathbf{S} = \{S_1, ..., S_M\}$ and tensor variables $\mathbf{T} = \{T_1, ..., T_N\}$. We extend the idea of NADE [Larochelle and Murray, 2011] by factorising the joint distribution of the variables into a product of conditional distributions for individual variables $p(T_1, ..., T_N, S_1, ..., S_M) = \prod_n p(T_n|\mathbf{T}_{\setminus n}, \mathbf{S}) \prod_m p(S_m|\mathbf{T}, \mathbf{S}_{\setminus m})$, where $\mathbf{T}_{\setminus n}$ and $\mathbf{S}_{\setminus m}$ are all tensor and symbolic variables except T_n and S_m respectively. The distribution $p(T_n|\mathbf{T}_{\setminus n}, \mathbf{S})$ is modelled by a generative network $\mathcal{N}^G_{\mathsf{P},n}$ and the distribution $p(S_m|\mathbf{T}, \mathbf{S}_{\setminus m})$ is modelled by a discriminative network $\mathcal{N}^D_{\mathsf{P},m}$. We denote the CNP for the predicate P as $\mathcal{N}^{D,G}_{\mathsf{P}}$. Similar to SNP, a CNP represents a variable from a logic predicate as a pair input-output vectors ($\mathbf{t}_n$,$\hat{\mathbf{t}}_n$) or ($\mathbf{s}_m$,$\hat{\mathbf{s}}_m$). For inference, given an assignment of tensor variables $T_n = \mathbf{o}^{k_n}_{\mathcal{T}}$ ($\mathbf{o}^{k_n}_{\mathcal{T}} \in C^{\mathcal{T}}_{k_n}$), and symbolic variables $S_m = o^{c_m}_{i_m}$ ($o^{c_m}_{i_m} \in C^{\mathcal{S}}_{c_m}$) we infer unassigned variables $T_{\hat{n}}$ and $S_{\hat{m}}$ by using $\mathcal{N}^G_{P,\hat{n}}$ and $\mathcal{N}^D_{P,\hat{m}}$ to compute $\hat{\mathbf{t}}_{\hat{n}}$ and $\hat{\mathbf{s}}_{\hat{m}}$ respectively, with $\mathbf{t}_n = \mathbf{o}^{k_n}_{\mathcal{T}}$, $\mathbf{s}_m = \mathbf{o}^{c_m}_{i_m}$, $\mathbf{t}_{\hat{n}} = \{0\}^{\dim(C^{\mathcal{T}}_{k_{\hat{n}}})}$, and $\mathbf{s}_{\hat{m}} = \{0\}^{|C^{\mathcal{S}}_{c_{\hat{m}}}|}$ as inputs.

Example 1. *Let us construct a SNP for predicate* $\mathsf{digit}(T, S)$*, where $T \in \mathbb{R}^{28\times 28}$ is a tensor variable of class C_T = \{* *,* *, ...\}, and S is a symbolic variable of class C_S = \{0,1, 3, 4, 5, 6, 7, 8, 9\}. A compositional neural network for predicate* $\mathsf{digit}(T, S)$ *is showed in Figure 1. This neural predicate $\mathcal{N}^{D,G}_{digit}$ consists of a discriminative neural network $\mathcal{N}^{D}_{digit}$ and a conditional generative adversarial network (a.k.a CGAN [Mirza and Osindero, 2014]) $\mathcal{N}^{G}_{digit}$. The generative network $\mathcal{N}^{G}_{digit}$ consists of two convolutional neural networks, a generator (CNN^2) and a discriminator (CNN^3). CNN^2 is trained to fool CNN^3, whose role is to classify fake and real samples, by generating realistic samples with random inputs Z. In CNLP, $\mathcal{N}^{D,G}_{digit}$ uses CNN^2 to infer tensor variable T and CNN^1 to infer symbolic variable S, given an assignment of the other. For example, CNN^1 in $\mathcal{N}^{D,G}_{digit}$ would predict $S = 6$ for the query* $\mathsf{digit}(T =$ *, S) while the CNN^2 would generate T =* *for the query* $\mathsf{digit}(T, S = 6)$.

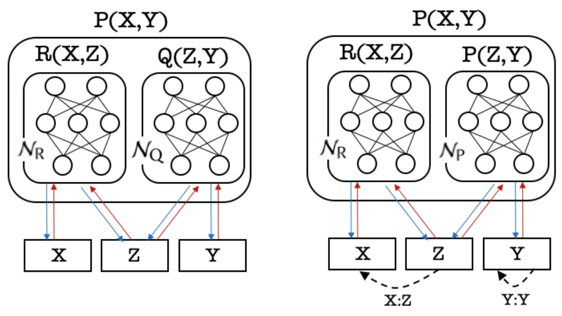


$\mathcal{N}^{G}_{digit}$
$\mathcal{N}^{D,G}_{digit}$
$\mathcal{N}^{D}_{digit}$
Z
fake/real
CNN^3
CNN^2
CNN^1
T
S

Figure 1: Compositional Neural Predicate $\mathcal{N}^{D,G}_{digit}$

2.3 Neural Rules

Definition 2. *A neural rule is a composition of B neural predicates $\mathcal{N}_{\mathsf{P}_1}$, $\mathcal{N}_{\mathsf{P}_2}$, ..., $\mathcal{N}_{\mathsf{P}_B}$ representing the rule $\mathsf{P}_{head}(.) \leftarrow \mathsf{P}_1(.) \wedge \mathsf{P}_2(.) \wedge ... \wedge \mathsf{P}_B(.)$, where $\mathcal{N}_{\mathsf{P}_b}$ and $\mathcal{N}_{\mathsf{P}_{b'}}$ ($b, b' \in \{1, ..., B\}$, $b \neq b'$) share the same layers for the variables which appears in both $\mathsf{P}_b(.)$ and $\mathsf{P}_{b'}(.)$.*



(a) Compositional Neural Network for the non-recursive rule $\mathsf{P}(X,Y) \leftarrow \mathsf{R}(X,Z) \wedge \mathsf{Q}(Z,Y)$

(b) Compositional Neural Network for the recursive rule $\mathsf{P}(X,Y) \leftarrow \mathsf{R}(X,Z) \wedge \mathsf{P}(Z,Y)$

Figure 2: Neural Rules

We model a non-recursive rule with no recursive/intermediate predicates by constructing a neural predicate for the predicates in the body of the rule. The neural predicates connect together through the shared layers which present the same variables. An intermediate predicate in a rule is modelled by chaining to the neural rule where it is the head. In the case of a recursive rule, the chaining applies to the rule itself. If two predicates in the body have the same name $\mathsf{P}_b = \mathsf{P}_{b'}$, they share the same neural predicate or/and same chaining to a neural rule (for intermediate predicates).

Example 2. *Figure 2a shows a neural rule for a non-recursive rule* $\mathsf{P}(X,Y) \leftarrow \mathsf{R}(X,Z) \wedge \mathsf{Q}(Z,Y)$. *The neural rule is composed of two neural networks,* $\mathcal{N}_\mathsf{R}$ *for* $\mathsf{R}(X,Z)$ *and* $\mathcal{N}_\mathsf{Q}$ *for* $\mathsf{Q}(Z,Y)$. *Both* $\mathcal{N}_\mathsf{R}$ *and* $\mathcal{N}_\mathsf{Q}$ *connect to the layer for the variable* Z. *We use two-way connections to demonstrate that any variables can be inferred and also can be used for the inference of other variables. Similarly, Figure 2b illustrates a neural rule for recursive rule* $\mathsf{P}(X,Y) \leftarrow \mathsf{R}(X,Z) \wedge \mathsf{P}(Z,Y)$ *with two neural predicates* $\mathcal{N}_\mathsf{R}$ *and* $\mathcal{N}_\mathsf{P}$. *This neural rule has recursive connections from* Z *to* X *and from* Y *to itself.*

2.4 Learning

Training of CNLP can be done in a distributed manner for all neural predicates. Symbolic neural predicates are constructed from facts as mentioned in 2.2. The number of hidden units in a SNP $\mathcal{N}_\mathsf{P}^{AE}$ is linearly proportional to the number of the facts for the corresponding predicate P. In the case of compositional neural predicates, we train the constituent neural networks from examples (facts) simultaneously. Since the neural networks in CNPs are loosely coupled, they retain their dependency during the training, thus improving efficiency.

3 Reasoning

Reasoning in compositional neural logic is to apply the neural networks to find answers to argument-retrieval queries. This can be done by combining search strategies in first-order logic with inference methods in neural networks. Here, we apply a search algorithm to select neural networks to infer unassigned variables until the answers are delivered. For example, in order to answer a query $\mathsf{P}(X = x, Y)$ in Example 1, instead of searching the KB for the objects of Y that satisfy P, we only need to search for the predicates that support P and use neural predicates to infer the unassigned variables to find the answer. In this case, we would search through the predicates in the body of the rule $\mathsf{P}(X,Y) \leftarrow \mathsf{R}(X,Z) \wedge \mathsf{Q}(Z,Y)$ and select $\mathcal{N}_\mathsf{R}$ to infer Z, given that the assignment $X = x$ is provided. The result of Z (e.g. z) then is fed to the neural predicate $\mathcal{N}_\mathsf{Q}$ to infer Y. We apply a similar process for the other rule $\mathsf{P}(X,Y) \leftarrow \mathsf{R}(X,Z) \wedge \mathsf{P}(Z,Y)$ except that at the last step we need to map $\{X : Z, Y : Y\}$ and recursively call a query $\mathsf{P}(X = z, Y)$ to find Y, hence the recursive connections in Figure 2b. The recursion occurs until a termination condition is met.

3.1 Voting Backward-Forward Chaining

In this section, we propose "voting backward-forward chaining" (VBFC), a novel reasoning algorithm for compositional neural logic. The algorithm is defined as a function INFER(P, $\mathbf{X}_{assigned}$, KB, MC) where P is the query predicate, $\mathbf{X}_{assigned}$ is a set of assigned variables, KB is the knowledge base, and MC is a memory cache. The function takes in a predicate P and its assigned variables $\mathbf{X}_{assigned}$ from a query and answers the values for the unassigned variables. It combines the searching strategy in backward chaining and knowledge deduction in forward chaining with the use of neural networks. VBFC starts with searching through the knowledge base (KB) for predicates that match the query. If there exists a symbolic neural predicate or a compositional neural predicate for P, the neural predicate would be used to infer unassigned variables. If P is a head predicate, CNLP works through the predicates in the body of its rule, in a similar way as backward chaining [Russell and Norvig, 2003]. However, different from the backward chaining algorithm, where facts are searched to confirm or refute a hypothesis, reasoning with CNLP is argument-retrieval which infers facts, including generating new facts in the process, as in forward chaining [Russell and Norvig, 2003]. After a fact is inferred, it would be used by other neural predicates or neural rules for the next inference step. Since CNLP consists of different types of neural predicates, as well as neural rules, we define a voting procedure to coordinate the reasoning, i.e. to decide which predicate to infer first to deduce knowledge for the inference of other predicates. Readers are advised to refer to Appendix A for the detailed description of the algorithm.

3.2 Batch Reasoning

Batch reasoning refers to the process of executing a batch of queries at the same time. In general, this is not possible for arbitrary queries, especially with a recursive knowledge base, because the inference for each query would expand in different paths. Fortunately, in the case where all queries share the same voting steps, batch reasoning is feasible. In a knowledge base without recursive rules, the voting is decided by the types of predicates and variables. Therefore, batch reasoning can be applied to the queries from the same predicate with the same list of assigned variables.

3.3 Reasoning With Recursive Rules

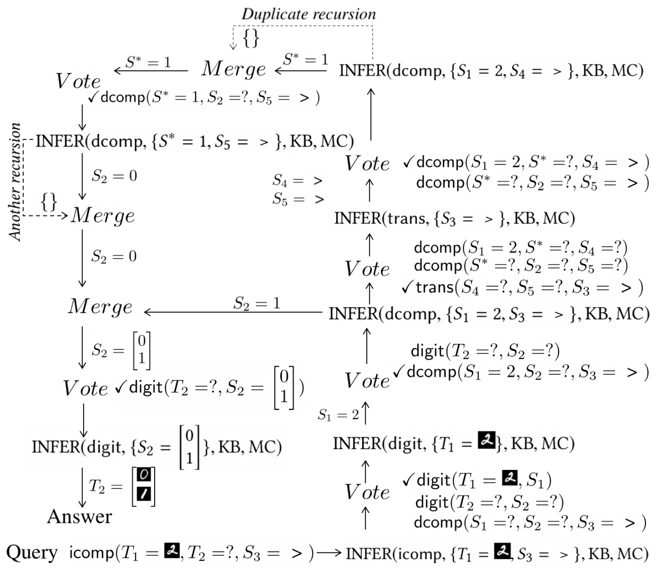
When applying Voting Backward-Forward Chaining to recursive rules, we need to *merge* the results from different branches and also define the conditions for termination. In the case where a recursive predicate is voted, the same procedure will be applied to the neural rule the predicate is chaining to. A recursion occurs until duplicate inference is detected, i.e. calling INFER on the same predicates and the assignment of variables which has been inferred earlier in the process. To this end, we maintain a memory cache to keep track of the states of the recursive calls. In Example 3 we demonstrate how VBFC works with a knowledge base to compare digit images.

Example 3. *Let us consider a knowledge base for comparison of digit images (Comparison KB) as follows:*
Rules:
$\mathsf{icomp}(T_1, T_2, S_3) \leftarrow \mathsf{digit}(T_1, S_1) \wedge \mathsf{digit}(T_2, S_2) \wedge \mathsf{dcomp}(S_1, S_2, S_3)$
$\mathsf{dcomp}(S_1, S_2, S_3) \leftarrow \mathsf{dcomp}(S_1, S^*, S_4) \wedge \mathsf{dcomp}(S^*, S_2, S_5) \wedge \mathsf{trans}(S_4, S_5, S_3)$
Facts:

$\mathsf{dcomp}(0,1,>)$,..., $\mathsf{dcomp}(8,9,<)$; $\mathsf{dcomp}(1,0,>)$, ..., $\mathsf{dcomp}(9,8,>)$; $\mathsf{trans}(>,>,>)$, $\mathsf{trans}(<,<,<)$; $\mathsf{digit}(0, 0)$,..., $\mathsf{digit}(9, 9)$, *etc.*

where $T1$,$T2$ *are digit images;* $S_1, S_2, S^* \in \{0,..,9\}$*;* $S_3, S_4, S_5 \in \{<,>\}$*. There are three neural predicates are constructed for this KB, namely* $\mathcal{N}^{D,G}_{\mathsf{digit}}$, $\mathcal{N}^{AE}_{\mathsf{dcomp}}$*, and* $\mathcal{N}^{AE}_{\mathsf{trans}}$.

In Figure 3, we show the reasoning graph of CNLP for the query $\mathsf{icomp}(T_1 = 2, T_2 =?, S_3 =>)$ *using the Comparison KB. Starting with the first rule, the first voting step selects* $\mathsf{digit}(T_1 = 2, S_1)$ *and trigger* $\mathcal{N}^{D,G}_{\mathsf{digit}}$ *to infer* $S_1 = 2$*. The next step votes* dcomp *with assigned variables* $S_1 = 2$ *and* $S_3 =>$*. This is a symbolic predicate, thus* $\mathcal{N}^{AE}_{\mathsf{dcomp}}$ *is used to infer* $S_2 = 1$*. Additionally,* dcomp *is also the head of the second rule in the knowledge base. Therefore, another round of voting is carried out for recursive reasoning with this rule which results in* $S_2 = 0$*. At the final step, using the inferred values of* S_2*, the CNP* $\mathcal{N}^{D,G}_{\mathsf{digit}}$ *delivers the answers* $T_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$.



Duplicate recursion
Another recursion
Vote
Merge
INFER(dcomp, {S1 = 2, S4 = >}, KB, MC)
✓dcomp(S* = 1, S2 =?, S5 = >)
INFER(dcomp, {S* = 1, S5 = >}, KB, MC)
✓dcomp(S1 = 2, S* =?, S4 = >)
dcomp(S* =?, S2 =?, S5 = >)
INFER(trans, {S3 = >}, KB, MC)
✓trans(S4 =?, S5 =?, S3 = >)
INFER(dcomp, {S1 = 2, S3 = >}, KB, MC)
digit(T2 =?, S2 =?)
✓dcomp(S1 = 2, S2 =?, S3 = >)
✓digit(T2 =?, S2 = [0 1])
INFER(digit, {S2 = [0 1]}, KB, MC)
INFER(digit, {T1 = 2}, KB, MC)
✓digit(T1 = 2, S1)
dcomp(S1 =?, S2 =?, S3 = >)
Answer
Query icomp(T1 = 2, T2 =?, S3 = >) → INFER(icomp, {T1 = 2, S3 = >}, KB, MC)

Figure 3: Reasoning steps for $\mathsf{icomp}(T_1 = 2, T_2 =?, S_3 =>)$.

4 Experiments

4.1 Comparison KB

In this experiment, we use the Comparision KB in Example 3 to demonstrate the learning and reasoning capabilities of CNLP with recursive rules. 10000 digit images and their labels are extracted from the MNIST dataset[1] as the facts for the digit predicate. We train $\mathcal{N}^{D,G}_{\mathsf{digit}}$ on those facts and apply the CNLP to answers three questions: (a) $\mathsf{icomp}(*, *, S_3)$: compare two numbers, for example $\mathsf{icomp}(5, 8, S_3)$; (b) $\mathsf{icomp}(T_1, 0, >)$: Which numbers are bigger than 0? and (c) $\mathsf{icomp}(T_1, 9, <)$: Which numbers are smaller than 9?. We omit the variable names in the queries for ease of presentation.

For the evaluation of the first question, we train a CNLP and apply it to answer 5000 test queries which achieves 97.62% accuracy. The same CNLP is employed to answer the second question and accurately generates the images of numbers whose values are bigger than 0, for example {1, 2, 3, 4, 5, 6, 7, 8, 9}. Similarly, the CNLP also responds correctly to third question $\mathsf{icomp}(T_1, 9, <)$, for example {0, 1, 2, 3, 4, 5, 6, 7, 8}.

[1]http://yann.lecun.com/exdb/mnist/

4.2 Addition

We extend the addition task in [Manhaeve *et al.*, 2018], based on the MNIST dataset, to demonstrate the advantages of robust learning and efficient reasoning of CNLP. The addition function consists of two lists of digit images representing two numbers and a symbolic variable for the sum of the numbers. In this experiment, we not only evaluate the proposed approach on the reasoning task $addition(*, *, ?)$, but also on $addition(?, *, *)$, $addition(?, *, ?)$, and $addition(?, ?, *)$ to showcase the capability of general reasoning with CNLP. We use the notation $*$ for an arbitrary input and ? for an output we want to infer in a query. The rules in the knowledge base are:

$$\mathsf{add}_1(T_1, T_2, S_3) \leftarrow \mathsf{digit}(T_1, S_1) \wedge \mathsf{digit}(T_2, S_2) \wedge \mathsf{sum}_1(S_1, S_2, S_3)$$
$$\mathsf{add}_2([T_{11}, T_{12}], [T_{21}, T_{22}], S_4) \leftarrow \mathsf{add}_1(T_{11}, T_{21}, S_{31}) \wedge \mathsf{add}_1(T_{12}, T_{22}, S_{32}) \wedge \mathsf{sum}_2(S_{31}, S_{32}, S_4)$$
$$\mathsf{add}_3([T_{11}, T_{12}, T_{13}], [T_{21}, T_{22}, T_{23}], S_5) \leftarrow \mathsf{add}_1(T_{11}, T_{12}, S_3) \wedge \mathsf{add}_2([T_{12}, T_{13}], [T_{22}, T_{23}], S_4) \wedge \mathsf{sum}_3(S_3, S_4, S_5)$$

The knowledge base in this experiment (addition KB) also has a set of grounding facts of the digit predicate which we use to train a CNP $\mathcal{N}^{D,G}_{\mathsf{digit}}$. We construct $\mathcal{N}^{AE}_{\mathsf{sum}_1}$, $\mathcal{N}^{AE}_{\mathsf{sum}_2}$, $\mathcal{N}^{AE}_{\mathsf{sum}_3}$ from the grounding facts of sum_1, sum_2, and sum_3 respectively. T_1, T_2, T_{11}, T_{12}, T_{13}, T_{21}, T_{22}, T_{23} are digit images; S_1, $S_2 \in [0, 9]$ are single-digit numbers; S_3, S_{31}, $S_{32} \in [00, 18]$ are double-digit numbers; $S_4 \in [000, 198]$ is triple-digit number; and $S_5 \in [0000, 1998]$. We construct a CNLP incrementally, starting with $\mathcal{N}^{D,G}_{\mathsf{digit}}$ and $\mathcal{N}^{AE}_{\mathsf{sum}_1}$ for the first neural rule (add_1) in the addition KB. The second neural rule (add_2) is built upon the first neural rule and $\mathcal{N}^{AE}_{\mathsf{sum}_2}$. Finally, the third neural rule (add_3) is composed of the two former neural rules and $\mathcal{N}^{AE}_{\mathsf{sum}_3}$.

Addition(*,*,?)

We evaluate CNLP and DeepProbLog on the addition task with single-digit numbers, double-digit numbers, and triple-digit numbers. All models were trained by Adam optimizer with the batch size 64. Different from CNLP which uses batch computation, DeepProbLog accumulates the gradients from each training query in a batch to update its parameters. The results show that CNLP takes less time to converge than DeepProbLog in all three cases single-digit, double-digit, and triple-digit. We show the learning curve of the triple-digit case in Figure 4. We also compare the efficiency of inference between CNLP and DeepProbLog. We run 1000 queries for the addition(*,*,?) task using a computer with a quad-core 3.6 GHz CPU and 16 GB of RAM. With such type of query, CNLP can utilise its batch reasoning capability and, therefore, is much more efficient than DeepProbLog. The latter relies on ProbLog inference where the grounding step may cause a bottleneck. The running time of both models on the addition of single-digit numbers, double-digit numbers, and triple-digit numbers are shown in Table 1.

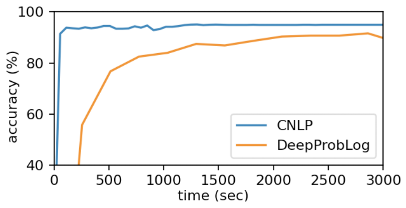


Figure 4: Triple digit

Figure 5: Learning curves of CNLP and DeepProbLog for addition(*,*,?) task with triple digits.

	Single	Double	Triple
CNLP	0.062	0.131	0.374
DeepProbLog	10.504	12.291	15.557

Table 1: Inference time (in seconds) for 1000 queries

Addition(?,?,*)

To show the capability of general reasoning, we evaluate CNLP in a generative task to produce images of two digits given their sum. For comparison, we use DCGAN [Radford *et al.*, 2016] and ACGAN [Odena *et al.*, 2017] as the baselines. We double the training data for DCGAN and ACGAN by duplicating every sample and switch the images. As CNLP leverages logical reasoning it decomposes the complexity of the task to smaller tasks, i.e. generating 28×28 images from 10 labels. This is much easier than generating larger images (28×56) from 19 labels, as in DCGAN and ACGAN. Table 2 shows that CNLP achieves better performance than DCGAN and ACGAN, as it has a higher inception score and lower cross-entropy. In Table 3, we present a random set of samples generated by CNLP, DCGAN, and ACGAN for $\mathsf{add}_1(T_1, T_2, 10)$. We found that most of the samples from DCGAN and ACGAN are the pairs of digit images for (7,3) and (3,7), while samples from CNLP are more diverse. Another advantage of CNLP over these two generative models is that it can generalise from the addition of single-digit numbers to multi-digit numbers without retraining. Examples of triple-digit can be found in the 3rd column of Table 4.

Addition(?,*,*) and Addition(?,*,?)

For completeness, we also use the same CNLP from the above tasks for these tasks. In Table 4, we demonstrate three patches of samples obtained from the addition(?,*,*) and addition(?,*,?) tasks (1st and 2nd column), and also from the addition(?,?,*) task (3rd column), in the case of triple-digit numbers. As we can see, a single CNLP can deal with different types of queries and variables.

4.3 Semantic Image Interpretation

Finally, we apply CNLP to the semantic image interpretation (SII) task which describes the semantic structure of an image through the relations of pairs of objects in the image. In this experiment, we use the images of indoor objects from PASCAL-PART dataset [Chen *et al.*, 2014]. Each object is represented by a rank-1 tensor consisting of geometric and semantic features extracted from its bounding box

	IS	Cross-Entropy
ACGAN	12.040 ± 1.021	1.42
DCGAN	09.137 ± 0.830	3.65
CNLP	$\mathbf{16.534 \pm 1.277}$	$\mathbf{0.159 \pm 0.105}$

Table 2: Performance of CNLP, DCGAN, and ACGAN for $\mathsf{add}_1(T_1, T_2, 10)$.

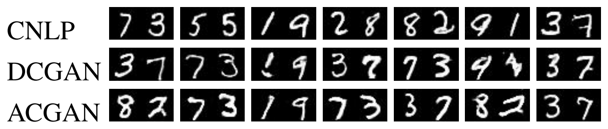


Table 3: Sample answers for the $\mathsf{add}_1(T_1, T_2, 10)$ query.

as in LTN [Donadello *et al.*, 2017]. The task requires the prediction of the types of objects and whether an object is a part of another. We use the predicate $\mathsf{type}(T_1, S_1)$ for the type-of-object relation, i.e. object T_1 has the type S_1. Normally, the part-of predicate would be $\mathsf{partof}(T_1, T_2)$ which is for "T_1 is part of T_2". However, to make it work with the argument-retrieval style of CNLP we convert the predicate to $\mathsf{partof}(T_1, T_2, S_3)$, where S_3 is the truth indicator of the predicate. The rule for this task is: $\mathsf{partof}(T_1, T_2, S_3) \leftarrow \mathsf{type}(T_1, S_1) \land \mathsf{type}(T_2, S_2) \land \mathsf{part}(S_1, S_2, S_3)$

The types of the variables in this KB are as follows. $T_1, T_2 \in \mathbb{R}^{64}$; $S_1, S_2 \in C = \{$bottle, body, cap, potted-plant, plant, pot, tvmonitor, screen, chair, sofa, diningtable$\}$; and $S_3 \in \{0, 1\}$. The CNLP for this task consists of a CNP $\mathcal{N}^{D}_{\mathsf{type}}$ and a SNP $\mathcal{N}^{AE}_{\mathsf{part}}$. We train $\mathcal{N}^{D}_{\mathsf{type}}$ from 8557 examples and construct $\mathcal{N}^{AE}_{\mathsf{part}}$ from 121 facts, for example, $\mathsf{part}(\text{bottle}, \text{bottle}, False)$ and $\mathsf{part}(\text{cap}, \text{bottle}, True)$ etc.

We compare the CNLP with other approaches, including Faster-RCNN (for type prediction), the inclusion ratio baseline RBPOF (for part-of prediction), and LTN [Serafini and d'Avila Garcez, 2016; Donadello *et al.*, 2017]. The test set consists of 2150 type samples and part-of 16674 samples. For a fair comparison, we also use the RMSProp optimiser for training as in [Donadello *et al.*, 2017]. The AUC scores of the approaches are detailed in Table 5 which indicates the better performance of CNLP in both type prediction and part-of prediction.

5 Conclusions and Future Work

We proposed CNLP, a compositional neural logic programming framework, to integrate symbolic and sub-symbolic representations for reasoning with different types of variables. The framework consists of multiple neural predicates chained together to model a knowledge base. The reasoning is conducted through a voting mechanism that supports recursion, and also batch inference where applicable. In the experiment, we demonstrated the ability of CNLP to work on both generative and discriminative tasks. We also show that CNLP is more efficient than DeepProbLog in the addition task and more effective than LTN in the image interpretation task. As future work, we will apply CNLP to a large database to evaluate the advantage of the compositional approach at scale.

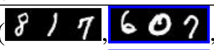
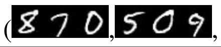
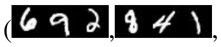
$\mathsf{add}_3(?,*,*)$	$\mathsf{add}_3(?,*,?)$	$\mathsf{add}_3(?,?,*)$
(,1495)	(,1348)	(,1495)
(,1495)	(,1407)	(,1495)
(,1379)	(,1424)	(,1379)
(,1379)	(,648)	(,1379)
(,1533)	(,843)	(,1533)
(,1533)	(,1057)	(,1533)

Table 4: Sample answers of CNLP for the $\mathsf{add}_3(?,*,*)$, $\mathsf{add}_3(?,*,?)$, and $\mathsf{add}_3(?,?,*)$ queries.

	Object type (AUC)	Part-of (AUC)
LTN_prior	0.800	0.598
LTN_expl	0.692	0.492
FRCNN	0.756	-
RBPOF	-	0.172
CNLP	$\mathbf{0.816} \pm 0.004$	$\mathbf{0.644} \pm 0.015$

Table 5: Area Under the Curve (AUC) on SII task.

A VBFC Algorithm

The algorithm is sketched in Algorithm 1 as an inference function, namely INFER(P, $\mathbf{X}_{assigned}$, KB, MC). The function takes in a predicate P and its assigned variables $\mathbf{X}_{assigned}$ from a query and answers the values for the unassigned variables. VBFC starts with searching through the knowledge base (KB) for predicates that match the query. If there exists a symbolic neural predicate or a compositional neural predicate for P, the neural predicate would be used to infer unassigned variables (INFER:22,24, a.k.a line 22 and line 24 in the Algorithm 1). If P is a head predicate, CNLP works through the predicates in the body of its rule (INFER:4). Each time, a predicate is voted to infer.

We define a voting function VOTE($\mathcal{P}$, ϕ), where $\mathcal{P}$ is the list of inferable predicates in a rule's body and ϕ is the assignment of values to variables, known as local working memory. First, any in-body predicates with a single unassigned variable will be voted, as we can always use their corresponding neural predicates to infer the unassigned variable straightforwardly. If none of them exists, we select the in-body symbolic predicates for voting, because their SNPs are certainly inferable. When there are no in-body symbolic predicates left, then we will select intermediate and recursive predicates. Among the predicates selected for voting, preference will be given to those with the least unassigned variables, and then, to those are non-recursive.

In VBFC, the voting idea extends the search strategy of backward chaining to work with argument-retrieval queries on both symbolic and tensor variables. During the reasoning, voting is performed iteratively, each time a predicate is selected to infer the unassigned variables (INFER:9). The results are then added to the working memory for the next round (INFER:17). The process repeats until no predicates are left to vote (INFER:10). After all predicates successfully infer the unassigned variables, we extract the inferred values and merge them to the answer (INFER:20).

Algorithm 1 Voting Backward-Forward Chaining (sketch)

```
1:  function INFER(P, $\mathbf{X}_{assigned}$,KB,MC)
2:      $\mathbf{X}^*_{unassigned} = \emptyset$            ▷ List of unassigned variables
3:      if $\{\mathsf{P}, \mathbf{X}_{assigned}\} \notin$ MC then
4:          for each rule $R \in$ KB whose head predicate is P do
5:              $\mathcal{P}$ is a set of predicates in $R$'s body
6:              $\mathcal{P}' = \{\}$ is a set of uninferable predicates
7:              Initialise a local working memory $\phi = \{\mathbf{X}_{assigned}\}$
8:              while $True$ do
9:                  $\mathsf{P}^{voted}, \mathbf{X}^{voted}_{assigned}$ = VOTE($\mathcal{P} \setminus \mathcal{P}'$,$\phi$)
10:                 if $\mathsf{P}^{voted} == NULL$ then
11:                     break
12:                 $\mathbf{X}^{voted}_{unassigned}$
13:                 =INFER($\mathsf{P}^{voted}$, $\mathbf{X}^{voted}_{assigned}$,KB,MC)
14:                 if $\mathbf{X}^{voted}_{unassigned} == \emptyset$ then
15:                     Add $\mathsf{P}^{voted}$ to $\mathcal{P}'$
16:                 else
17:                     Add $\mathbf{X}^{voted}_{unassigned}$ to $\phi$
18:                     $\mathcal{P}' = \{\}$
19:                     Remove $\mathsf{P}^{voted}$ from of $\mathcal{P}$
20:             if $\mathcal{P}' == \{\}$ then
21:                 Merge $\phi.\mathbf{X}_{unassigned}$ into $\mathbf{X}^*_{unassigned}$
22:     if $\mathcal{N}^{AE}_{\mathsf{P}}$ exists then
23:         $\mathbf{X}_{assigned} \overset{\mathcal{N}^{AE}_{\mathsf{P}}}{\mapsto} \mathbf{X}_{unassigned}$
24:     else if $\mathcal{N}^{D,G}_{\mathsf{P}}$ exists and $(|\mathbf{X}_{unassigned}| == 1)$ then
25:         $\mathbf{X}_{assigned} \overset{\mathcal{N}^{D,G}_{\mathsf{P}}}{\mapsto} \mathbf{X}_{unassigned}$
26:     else:
27:         $\mathbf{X}_{unassigned} = \emptyset$               ▷ Unable to infer
28:     Merge $\mathbf{X}_{unassigned}$ into $\mathbf{X}^*_{unassigned}$
29:     Add $\{\mathsf{P}, \mathbf{X}^*_{assigned}\}$ to MC
30:     return $\mathbf{X}^*_{unassigned}$
```

References

[Bottou, 2014] Léon Bottou. From machine learning to machine reasoning. *Mach. Learn.*, 94(2):133–149, February 2014.

[Chen *et al.*, 2014] Xianjie Chen, Roozbeh Mottaghi, Xiaobai Liu, Sanja Fidler, Raquel Urtasun, and Alan L. Yuille. Detect what you can: Detecting and representing objects using holistic models and body parts. In *2014 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2014, Columbus, OH, USA, June 23-28, 2014*, pages 1979–1986. IEEE Computer Society, 2014.

[Cohen *et al.*, 2017] William W. Cohen, Fan Yang, and Kathryn Mazaitis. Tensorlog: Deep learning meets probabilistic dbs. *CoRR*, abs/1707.05390, 2017.

[Donadello *et al.*, 2017] Ivan Donadello, Luciano Serafini, and Artur d'Avila Garcez. Logic tensor networks for semantic image interpretation. In *IJCAI-17*, pages 1596–1602, 2017.

[Larochelle and Murray, 2011] Hugo Larochelle and Iain Murray. The neural autoregressive distribution estimator. In Geoffrey Gordon, David Dunson, and Miroslav Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 29–37, Fort Lauderdale, FL, USA, 11–13 Apr 2011. PMLR.

[Manhaeve *et al.*, 2018] Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. Deepproblog: Neural probabilistic logic programming. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 3749–3759. Curran Associates, Inc., 2018.

[Mirza and Osindero, 2014] Mehdi Mirza and Simon Osindero. Conditional generative adversarial nets, 2014. cite arxiv:1411.1784.

[Odena *et al.*, 2017] Augustus Odena, Christopher Olah, and Jonathon Shlens. Conditional image synthesis with auxiliary classifier GANs. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 2642–2651, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR.

[Pierrot *et al.*, 2019] Thomas Pierrot, Guillaume Ligner, Scott E Reed, Olivier Sigaud, Nicolas Perrin, Alexandre Laterre, David Kas, Karim Beguir, and Nando de Freitas. Learning compositional neural programs with recursive tree search and planning. In H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 14673–14683. Curran Associates, Inc., 2019.

[Radford *et al.*, 2016] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[Reed and de Freitas, 2016] Scott Reed and Nando de Freitas. Neural programmer-interpreters. In *International Conference on Learning Representations (ICLR)*, 2016.

[Riveret *et al.*, 2020] Regis Riveret, Son N. Tran, and Artur d'Avila Garcez. Neuro-Symbolic Probabilistic Argumentation Machines. In *Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning*, pages 871–881, 9 2020.

[Russell and Norvig, 2003] Stuart Russell and Peter Norvig. Knowledge, reasoning, and planning. In *Artificial Intelligent: A Modern Approach*. Pearson Education, 2003.

[Serafini and d'Avila Garcez, 2016] Luciano Serafini and Artur S. d'Avila Garcez. Learning and reasoning with logic tensor networks. In *AI*IA*, pages 334–348, 2016.

[Socher *et al.*, 2013] Richard Socher, Danqi Chen, Christopher D. Manning, and Andrew Ng. Reasoning with neural tensor networks for knowledge base completion. In *NIPS*, pages 926–934. 2013.

[Towell and Shavlik, 1994] Geoffrey Towell and Jude Shavlik. Knowledge-based artificial neural networks. *Artificial Intelligence*, 70:119–165, 1994.

[Tran and d'Avila Garcez, 2018] Son N. Tran and Artur d'Avila Garcez. Deep logic networks: Inserting and extracting knowledge from deep belief networks. *IEEE Transaction on Neural Networks and Learning Systems.*, (29):246–258, 2018.

[Wang *et al.*, 2019] Wenguan Wang, Zhijie Zhang, Siyuan Qi, Jianbing Shen, Yanwei Pang, and Ling Shao. Learning compositional neural information fusion for human parsing. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 5703–5713, 2019.

[Yang *et al.*, 2017] Fan Yang, Zhilin Yang, and William W Cohen. Differentiable learning of logical rules for knowledge base reasoning. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 2319–2328. Curran Associates, Inc., 2017.