Complex Program Induction for Querying Knowledge Bases in the Absence of Gold Programs

Amrita Saha[1] **Ghulam Ahmed Ansari**[1] **Abhishek Laddha**[*1]
Karthik Sankaranarayanan[1] **Soumen Chakrabarti**[2]

[1]IBM Research India, [2]Indian Institute of Technology Bombay
amrsaha4@in.ibm.com, ansarigh@in.ibm.com, laddhaabhishek11@gmail.com,
kartsank@in.ibm.com, soumen@cse.iitb.ac.in

Abstract

Recent years have seen increasingly complex question-answering on knowledge bases (KBQA) involving logical, quantitative, and comparative reasoning over KB subgraphs. Neural Program Induction (NPI) is a pragmatic approach toward modularizing the reasoning process by translating a complex natural language query into a multi-step executable program. While NPI has been commonly trained with the "gold" program or its sketch, for realistic KBQA applications such gold programs are expensive to obtain. There, practically only natural language queries and the corresponding answers can be provided for training. The resulting combinatorial explosion in program space, along with extremely sparse rewards, makes NPI for KBQA ambitious and challenging. We present Complex Imperative Program Induction from Terminal Rewards (CIPITR), an advanced neural programmer that mitigates reward sparsity with auxiliary rewards, and restricts the program space to semantically correct programs using high-level constraints, KB schema, and inferred answer type. CIPITR solves complex KBQA considerably more accurately than key-value memory networks and neural symbolic machines (NSM). For moderately complex queries requiring 2- to 5-step programs, CIPITR scores at least 3× higher F1 than the competing systems. On one of the hardest class of programs (comparative reasoning) with 5–10 steps, CIPITR outperforms NSM by a factor of 89 and memory networks by 9 times.[1]

1 Introduction

Structured knowledge bases (KB) like Wikidata and Freebase can support answering questions (KBQA) over a diverse spectrum of structural complexity. This includes queries with single-hop (*Obama's birthplace*) (Yao, 2015; Berant et al., 2013), or multi-hop (*who voiced Meg in Family Guy*) (Bast and Haußmann, 2015; Yih et al., 2015; Xu et al., 2016; Guu et al., 2015; McCallum et al., 2017; Das et al., 2017), or complex queries such as "*how many countries have more rivers and lakes than Brazil*?" (Saha et al., 2018). Complex queries require a proper assembly of selected operators from a library of graph, set, logical, and arithmetic operations into a complex procedure, and is the subject of this paper.

Relatively simple query classes, in particular, in which answers are KB entities, can be served with feed-forward (Yih et al., 2015) and seq2seq (McCallum et al., 2017; Das et al., 2017) networks. However, such systems show copying or rote learning behavior when Boolean or open numeric domains are involved. More complex queries need to be evaluated as an acyclic expression graph over nodes representing KB access, set, logical, and arithmetic operators (Andreas et al., 2016a). A practical alternative to inferring a stateless expression graph is to generate an imperative sequential program to solve the query. Each step of the program selects an atomic operator and a set of previously defined variables as arguments and writes the result to scratch memory, which can then be used in subsequent steps. Such imperative programs are preferable to opaque, monolithic networks for their interpretability and generalization to diverse domains. Another

[*]Now at Hike Messenger

[1]The NSM baseline in this work is a re-implemented version, as the original code was not available.

Transactions of the Association for Computational Linguistics, vol. 7, pp. 185–200, 2019. Action Editor: Scott Wen-tau Yih.
Submission batch: 8/2018; Revision batch: 11/2018; Final submission: 1/2019; Published 4/2019.
© 2019 Association for Computational Linguistics. Distributed under a CC-BY 4.0 license.

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/tacl_a_00262 by guest on 29 March 2024

motivation behind opting for the program induction paradigm for solving complex tasks, such as complex question answering, is modularizing the end-to-end complex reasoning process. With this approach it is now possible to first train separate modules for each of the atomic operations involved and then train a program induction model that learns to use these separately trained models and invoke the sub-modules in the correct fashion to solve the task. These sub-modules can even be task-agnostic generic models that can be pretrained with much more extensive training data, while the program induction model learns from examples pertaining to the specific task. This paradigm of program induction has been used for decades, with rule induction and probabilistic program induction techniques in Lake et al. (2015) and by constructing algorithms utilizing formal theorem-proving techniques in Waldinger and Lee (1969). These traditional approaches (e.g., Muggleton and Raedt, 1994) incorporated domain specific knowledge about programming languages instead of applying learning techniques. More recently, to promote generalizability and reduce dependecy on domain specific knowledge, neural approaches have been applied to problems like addition, sorting, and word algebra problems (Reed and de Freitas, 2016; Bosnjak et al., 2017) as well as for manipulating a physical environment (Bunel et al., 2018).

Program Induction has also seen initial promise in translating simple natural language queries into programs executable in one or two hops over a KB to obtain answers (Liang et al., 2017). In contrast, many of the complex queries from Saha et al. (2018), such as the one in Figure 1, require up to 10-step programs involving multiple relations and several arithmetic and logical operations. Sample operations include *gen_set*: collecting $\{t : (h, r, t) \in \text{KB}\}$, computing *set_union*, counting set sizes (*set_count*), comparing numbers or sets, and so forth. These operations need to be executed in the correct order, with correct parameters, sharing information via intermediate results to arrive at the correct answer. Note also that the actual gold program is not available for supervision and therefore the large space of possible translation actions at each step, coupled with a large number of steps needed to get any payoff, makes the reward very sparse. This renders complex KBQA in the absence of gold programs extremely challenging.

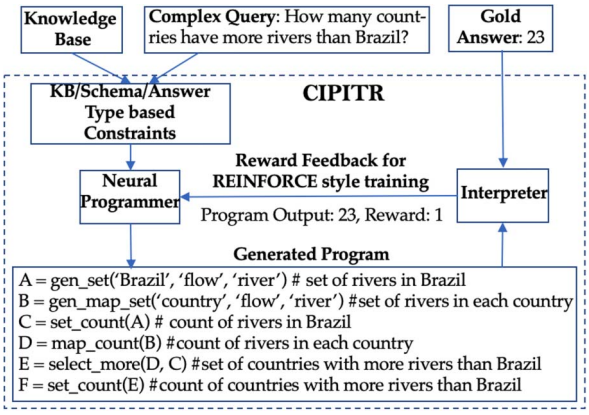


Figure 1: The CIPITR framework reads a natural language query and writes a program as a sequence of actions, guided at every step by constraints posed by the KB and the answer-type. Because the space of actions is discrete, REINFORCE is used to learn the action selection by computing the reward from the output answer obtained by executing the program and the target answer, which is the only source of supervision.

Main Contributions

- We present "Complex Imperative Program Induction from Terminal Rewards" (CIPITR),[2] an advanced Neural Program Induction (NPI) system that is able to answer complex logical, quantitative, and comparative queries by inducing programs of length up to 7, using 20 atomic operators and 9 variable types. This, to our knowledge, is the first NPI system to be trained with only the gold answer as (very distant) supervision for inducing such complex programs.

- CIPITR reduces the combinatorial program space to only semantically correct programs by (i) incorporating symbolic constraints guided by KB schema and inferred answer type, and (ii) adopting pragmatic programming techniques by decomposing the final goal into a hierarchy of sub-goals, thereby mitigating the sparse reward problem by considering additional auxiliary rewards in a generic, task-independent way.

We evaluate CIPITR on the following two challenging tasks: (i) complex KBQA posed by the recently-published CSQA data set (Saha et al., 2018) and (ii) multi-hop KBQA in one of the more

[2]The code and reinforcement learning environment of CIPITR is made public in `https://github.com/CIPITR/CIPITR`.

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/tacl_a_00262 by guest on 29 March 2024

popularly used KBQA data sets WebQuestionsSP (Yih et al., 2016). WebQuestionsSP involves complex multi-hop inferencing, sometimes with additional constraints, as we will describe later. However, CSQA poses a much greater challenge, with its more diverse classes of complex queries and almost 20-times larger scale. On a data set such as CSQA, contemporary models like neural symbolic machines (NSM) fail to handle exponential growth of the program search space caused by a large number of operator choices at every step of a lengthy program. Key-value memory networks (KVMnet) (Miller et al., 2016) are also unable to perform the necessary complex multi-step inference. CIPITR outperforms them both by a significant margin while avoiding exploration of unwanted program space or memorization of low-entropy answer distributions. On even moderately complex programs of length 2–5, CIPITR scored at least $3\times$ higher F1 than both. On one of the hardest class of programs of around 5–10 steps (i.e., comparative reasoning), CIPITR outperformed NSM by a factor of 89 and KVMnet by a factor of 9. Further, we empirically observe that among all the competing models, CIPITR shows the best generalization across diverse program classes.

2 Related Work

Whereas most of the earlier efforts to handle complex KBQA did not involve writable memory, some recent systems (Miller et al., 2016; Neelakantan et al., 2015, 2016; Andreas et al., 2016b; Dong and Lapata, 2016) used end-to-end differentiable neural networks. One of the state-of-the-art neural models for KBQA, the key-value memory network KVMnet (Miller et al., 2016) learns to answer questions by attending on the relevant KB subgraph stored in its memory. Neelakantan et al. (2016) and Pasupat and Liang (2015) support simple queries over tables, for example, of the form "find the sum of a specified column" or "list elements in a column more than a given value." The query is read by a recurrent neural network (RNN), and then, in each translation step, the column and operator are selected using the query representation and history of operators and columns selected in the past. Andreas et al. (2016b) use a "stateless" model where neural network based subroutines are assembled using syntactic parsing.

Recently, Reed and de Freitas (2016) took an early influential step with the NPI compositional framework that learns to decompose high level tasks like addition and sorting into program steps (carry, comparison) aided by persistent memory. It is trained by high-level task input and output as well as all the program steps. Li et al. (2016) and Bosnjak et al. (2017) took another important step forward by replacing NPI's expensive strong supervision with supervision of the program-sketch. This form of supervision at every intermediate step still keeps the problem simple, by arresting the program space to a tractable size. Although such data are easy to generate for simpler problems such as arithmetic and sorting, it is expensive for KBQA. Liang et al. (2017) proposed the NSM framework in absence of the gold program, which translates the KB query to a structured program token-by-token. While being a natural approach for program induction, NSM has several inherent limitations preventing generalization towards longer programs that are critical for complex KBQA. Subsequently, it was evaluated only on WebQuestionsSP (Yih et al., 2016), that requires relatively simpler programs. We consider NSM as the primary and KVMnet as an additional baseline and show that CIPITR significantly outperforms both, especially on the more complex query types.

3 Complex KBQA Problem Set-up

3.1 CSQA Data Set

The CSQA data set (Saha et al., 2018) contains 1.15M natural language questions and its corresponding gold answer from WikiData Knowledge Base. Figure 1 shows a sample query from the data set along with its true program-decomposed form, the latter *not provided by CSQA*. CSQA is particularly suited to study the Complex Program Induction (CPI) challenge over other KBQA data sets because:

- It contains large-scale training data of question-answer pairs across diverse classes of complex queries, each requiring different inference tools over large KB sub-graphs.
- Poor state-of-the-art performance of memory networks on it motivates the need for sweeping changes to the NPI's learning strategy.

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/tacl_a_00262 by guest on 29 March 2024

- The massive size of the KB involved (13 million entities and 50 million tuples) poses a scalability challenge for prior NPI techniques.

- Availability of KB metadata helps standardize comparisons across techniques (explained subsequently).

We adapt CSQA in two ways for the CPI problem.

Removal of extended conversations: To be consistent with the NSM work on KBQA, we discard QA pairs that depend on the previous dialogue context. This is possible as every query is annotated with information on whether it is self-contained or depends on the previous context. Relevant statistics of the resulting data set are presented in Table 3.

Use of gold entity, type, and relation annotations to standardize comparisons: Our focus being on the reasoning aspect of the KBQA problem, we use the gold annotations of canonical KB entities, types, and relations available in the data set along with the the queries, in order to remove a prominent source of confusion in comparing KBQA systems (i.e., all systems take as inputs the natural language query, with spans identified with KB IDs of entities, types, relations, and integers). Although annotation accuracy affects a complete KBQA system, our focus here is on complex, multi-step program generation with only final answer as the distant supervision, and not entity/type/relation linking.

3.2 WebQuestionsSP Data Set

In Figure 2 we illustrate one of the most complex questions from the the WebQuestionsSP data set and its semantic parsed version provided by human annotator. Questions in the WebQuestionsSP data set are answerable from the Freebase KB and tyically require up to 2-hop inference chains, sometimes with additional requirements of satisfying specific constraints. These constraints can be temporal (e.g., *governing_position_held_from*) or non-temporal (e.g., *government_office_position_or_title*). The human-annotated semantic parse of the questions provide the exact structure of the subgraph and the inference process on it to reach the final answer. As in this work, we are focusing on inducing programs where the gold entity relation annotations are known; for this data set as well, we use the human-annotations to collect all

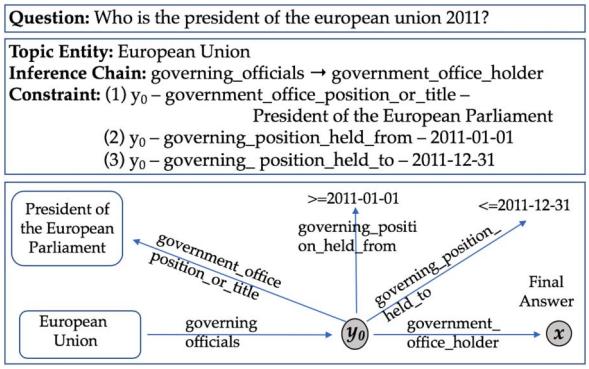


Figure 2: Semantic parsed form of a sample Question from WebQuestionsSP, along with the depiction of the reasoning over the subgraph to reach the answer.

the entities and relations in the oracle subgraph associated with the query. The NPI model has to understand the role of these gold program inputs in question-answering and learn to induce a program to reflect the same inferencing.

4 Complex Imperative Program Induction from Terminal Rewards

4.1 Notation

This subsection introduces the different notations commonly used by our model.

Nine variable-types: (distinct from KB types)

- KB artifacts: ent(entity), rel(relation), $type$
- Base data types: int, $bool$, $None$ (empty argument type used for padding)
- Composite data types: set (i.e., set of KB entities) or map_set and map_int (i.e., a mapping function from an entity to a set of KB entities or an integer)

Twenty Operators:

- $gen_set(ent, rel, type) \rightarrow set$
- $verify(ent, rel, ent) \rightarrow bool$
- $gen_map_set(type, rel, type) \rightarrow map_set$
- $map_count(map_set) \rightarrow map_int$
- $set_\{union/ints/diff\}(set, set) \rightarrow set$
- $map_\{union/ints/diff\}(map_set, map_set) \rightarrow map_set$
- $set_count(set) \rightarrow int$

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/tacl_a_00262 by guest on 29 March 2024

- $select_\{atleast/atmost/more/less/equal/approx\}(map_int, int) \rightarrow set$
- $select_\{max/min\}(map_int) \rightarrow ent$
- $no_op()$ (i.e., no action taken)

Symbols and Hyperparameters: (typical values)

- num_op: Number of operators (20)
- num_var_types: Number of variable types (9)
- max_var: Maximum number of variables accommodated in memory for each type (3)
- m: Maximum number of arguments for an operator ($None$ padding for fewer arguments) (3)
- d^{key} & d^{val}: Dimension of the key and value embeddings ($d^{key} \ll d^{val}$) (100, 300)
- n_p & n_v: Number of operators and argument variables sampled per operator each time (4, 10)
- f with subscript: some feed-forward network

Embedding Matrices: The model is trained with a vocabulary of operators and variable-types. In order to sample operators, two matrices $M^{op_key} \in \mathbb{R}^{num_op \times d^{key}}$ and $M^{op_val} \in \mathbb{R}^{num_op \times d^{val}}$ are needed for encoding the operator's key and value embedding. The key embedding is used for looking up and retrieving an entry from the operator vocabulary and the corresponding value embedding encodes the operator information. The variable type has only the value embedding $M^{vtype_val} \in \mathbb{R}^{num_op \times d^{val}}$ as no lookup is needed on it.

Operator Prototype Matrices: These matrices store the argument variable type information for the m arguments of every operator in $M^{op_arg} \in \{0, 1, \ldots, num_var_types\}^{num_op \times m}$ and the output variable type created by it in $M^{op_out} \in \{0, 1, \ldots, num_var_types\}^{num_op}$.

Memory Matrices: This is the query-specific scratch memory for storing new program variables as they get created by CIPITR. For each variable type, we have separate key and value embedding matrices $M^{var_key} \in \mathbb{R}^{num_var_type \times max_var \times d^{key}}$ and $M^{var_val} \in \mathbb{R}^{num_var_type \times max_var \times d^{val}}$, respectively for looking up a variable in memory and accessing the information in it. In addition, we also have a variable attention matrix $M^{var_att} \in \mathbb{R}^{num_var_type \times max_var}$ which stores the attention vector over the variables declared of each type.
CIPITR consists of three components:

The preprocessor takes the input query and the KB and performs the task of entity, relation, and type linking which acts as input to the program induction. It also pre-populates the variable memory matrices with any entity, relation, type, or integer variable directly extracted from the query.

The programmer model takes as input the natural language question, the KB, and the pre-populated variable memory tables to generate a program (i.e., a sequence of operators invoked with past instantiated variables as their arguments and generating new variables in memory).

The interpreter executes the generated program with the help of the KB and scratch memory and outputs the system answer.

During training, the predicted answer is compared with the gold to obtain a reward, which is sent back to CIPITR to update its model parameters through a REINFORCE (Williams, 1992) objective. In the current version of CIPITR, the preprocessor consults an oracle to link entities, types and relations in the query to the KB. This is to isolate the programming performance of CIPITR from the effect of imperfect linkage. Extending earlier studies (Karimi et al., 2012; Khalid et al., 2008) to investigate robustness of CIPITR to linkage errors may be of future interest.

4.2 Basic Memory Operations in CIPITR

We describe some of the foundational modules invoked by the rest of CIPITR.

Memory Lookup: The memory lookup looks up scratch memory with a given probe, say x (of arbitrary dimension), and retrieves the memory entry having closest key embedding to x. It first passes x through a feed-forward layer to transform its dimension to key embedding dimension x_key. Then, by computing softmax over the matrix multiplication of M^{x_key} and x^{key}, the distribution over the memory variables for lookup is obtained.

$$x^{key} = f(x), x^{dist} = softmax(M^{x_key} x^{key})$$

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/tacl_a_00262 by guest on 29 March 2024

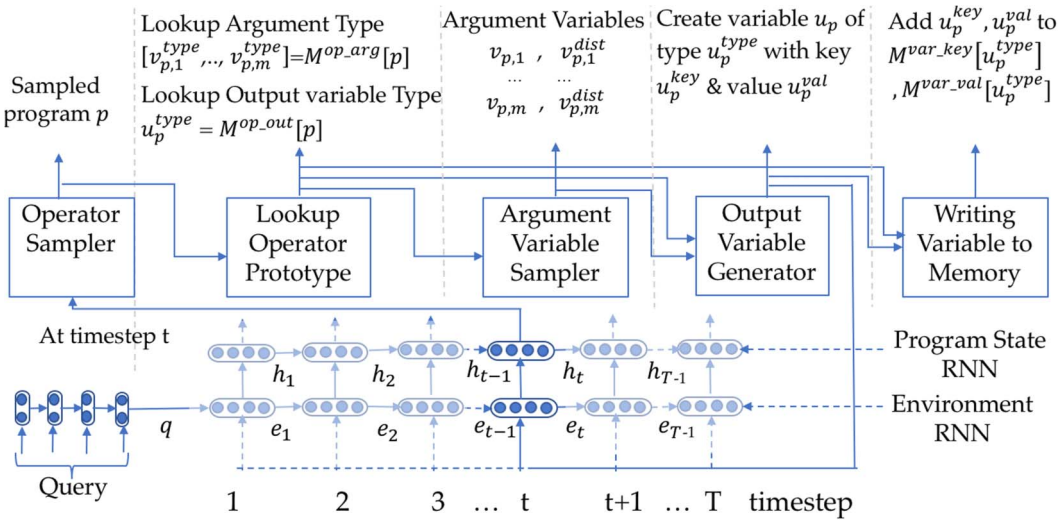


Figure 3: CIPITR Control Flow (with n_p=1 & n_v=1 for simplicity) depicting the order in which different modules (see Section 4) are invoked. A corresponding example execution trace of the CIPITR algorithm is given in Figure 4.

Feasibility Sampling: To restrict the search space to meaningful programs, CIPITR incorporates both high-level generic or task-specific constraints when sampling any action. The generic constraints can help it adopt more pragmatic programming styles like not repeating lines of code or avoiding syntactical errors. The task specific constraints ensure that the generated program is consistent as per the KB schema or on execution gives an answer of the desired variable type. To sample from the feasible subset using these constraints, the input sampling distribution, x^{dist}, is element-wise transformed by a feasibility vector x^{feas} followed by a L1-normalization. Along with the transformed distribution, the top-k entries $x^{sampled}$ is also returned.

Algorithm 1 Feasibility Sampling

Input:

- $x^{dist} \in \mathbb{R}^N$ (where N is the size of the population set over which lookup needs to be done)
- $x^{feas} \in \{0,1\}^N$ (boolean feasibility vector)
- k (top-k sampled)

Procedure: FeasSampling (x^{dist}, x^{feas}, k)
$x^{dist} = x^{dist} \odot x^{feas}$ (elementwise multiply)
$x^{dist} =$ L1-Normalized(x^{dist})
$x^{sampled} = k$-argmax(x^{dist})
Output: $x^{dist}, x^{sampled}$

Writing a new variable to memory: This operation takes a newly generated variable, say x, of type x^{type} and adds its key and value embedding to the row corresponding to x^{type} in the memory matrices. Further, it updates the attention vector for x^{type} to provide maximum weight to the newest variable generated, thus, emulating a stack like behavior.

Algorithm 2 Write a new variable to memory

Input:

- x^{key}, x^{val} the key and value embedding of x
- x^{type} is a scalar denoting type of variable x

Procedure: WriteVarToMem$(x^{key}, x^{val}, x^{type})$
i is the 1^{st} empty slot in the row $M^{x_key}[x^{type}, :]$
$M^{var_key}[x^{type}, i] = x^{key}$
$M^{var_val}[x^{type}, i] = x^{val}$
$M^{var_att}[x^{type}, :] =$ L1-Normalized$(M^{var_att}[x^{type}, :] + \textit{One-Hot}(i))$

4.3 CIPITR Architecture

In Figure 3, we sketch the CIPITR components; in this section we describe them in the order they appear in the model.

Query Encoder: The query is first parsed into a sequence of KB-entities and non-KB words. KB entities e are embedded with the concatenated vector $[\text{TransE}(e), \mathbf{0}]$ using Bordes et al. (2013), and non-KB words ω with $[\mathbf{0}, \text{GloVe}(\omega)]$. The final query representation is obtained from a GRU encoder as q.

NPI Core: The query representation q is fed at the initial timestep to an environment encoding

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/tacl_a_00262 by guest on 29 March 2024

RNN, which gives out the environment state e_t at every timestep. This, along with the value embedding u^{val}_{t-1} of the last output variable generated by the NPI engine, is fed at every timestep into another RNN that finally outputs the program state h_t. h_t is then fed into the successive modules of the program induction engine as described below. The 'OutVarGen' algorithm describes how to obtain u^{val}_{t-1}.

Procedure: NPI Core$(e_{t-1}, h_{t-1}, u^{val}_{t-1})$
$e_t = GRU(e_{t-1}, u^{val}_{t-1})$
$h_t = GRU(e_t, u^{val}_{t-1}, h_{t-1})$
Output: e_t, h_t

Operator Sampler: It takes the program state h_t, a Boolean vector p^{feas}_t denoting operator feasibility, and the number of operators to sample n_p. It passes h_t through the Lookup operation followed by Feasibility Sampling to obtain the top-n_p operations (P_t).

Argument Variable Sampler: For each sampled operator p, it takes: (i) program state h_t, (ii) the list of variable types V^{type}_p of the m arguments obtained by looking up the operator prototype matrix M^{op_arg}, and (iii) a Boolean vector V^{feas}_p that indicates the valid variable configurations for the m-tuple arguments of the operator p. For each of the m arguments, a feed-forward network f_{vtype} first transforms the program state h_t to a vector in $\mathbb{R}^{max_var}$. It is then element-wise multiplied with the current attention state over the variables in memory of that type. This provides the program-state-specific attention over variables $v^{att}_{p,j}$ which is then passed through the Lookup function to obtain the distribution over the variables in memory. Next, feasibility sampling is applied over the joint distribution of its argument variables, comprised of the m individual distributions. This provides the top-n_v tuples of m-variable instantiations V_p.

Output Variable Generator: The new variable u_p of type $u^{type}_p = M^{op_out}[p]$ is generated by the procedure **OutVarGen** by invoking a sampled operator p with m variables $v_{p,1} \cdots v_{p,m}$ of type $v^{type}_{p,1} \cdots v^{type}_{p,m}$ as arguments. This also requires generating the key and value embedding, which are both obtained by applying different feed-forward layers over the concatenated representation of the value embedding of the operator $M^{op_val}[p]$, argument types $(M^{vtype_val}[v^{type}_{p,1}] \cdots M^{vtype_val}[v^{type}_{p,m}])$ and the instantiated variables $(M^{var_val}[v^{type}_{p,1}, v_{p,1}] \ \cdots \ M^{var_val}[v^{type}_{p,m}, v_{p,m}])$. The newly generated variable is then written to memory using Algorithm **WriteVarToMem**.

Procedure: ArgVarSampler$(h_t, V^{type}_p, V^{feas}_p, n_v)$
for $j \in 1, 2, \cdots, m$ **do**
$v^{att}_{p,j} = softmax(M^{var_att}[V^{type}_{p,j}]) \odot f_{vtype}(h_t)$
$v^{dist}_{p,j} =$ **Lookup**$(v^{att}_{p,j}, f_{var}, M^{var_key}[V^{type}_{p,j}])$
$V^{dist}_p = v^{dist}_{p,0} \times v^{dist}_{p,1} \cdots \times v^{dist}_{p,m}$, ▷ Joint Distribution
$V^{dist}_p, V_p =$ **FeasSampling**$(V^{dist}_p, V^{feas}_p, n_v)$
Output: V_p

End-to-End CIPITR training: CIPITR takes a natural language query and generates an output program in a number of steps. A program is composed of actions, which are operators applied over variables (as in Figure 3). In each step, it selects an operator and a set of previously defined variables as its arguments, and writes the operator output to a dynamic memory, to be subsequently used for further search of next actions. To reduce exposure bias (Ranzato et al., 2015), CIPITR uses a beam search to obtain multiple candidate programs to provide feedback to the model from a single training instance. Algorithm 3 shows the pseudocode of the program induction algorithm (with beam size b as 1 for simplicity), which goes over T time steps, each time sampling n_p feasible operators conditional to the program state. Then, for each of the n_p operators, it samples n_v feasible

Algorithm 3 CIPITR pseudo-code (beam size=1)
Query Encoding: $q = GRU(Query)$
Initialization: $e_1, h_1 = f(q), A = [\,]$
for $t \in 1, \cdots, T$ **do**
$p^{feas}_t =$ **FeasibleOp**$()$
$P_t =$ **OperatorSampler**(h_t, p^{feas}_t, n_p)
$C = \{\}$
for $p \in P_t$ **do**
$V^{type}_p = [v^{type}_{p,1}, \cdots, v^{type}_{p,m}] = M^{op_arg}[p]$
$V^{feas}_p =$ **FeasibleVar**(p)
$V_p =$ **ArgVarSampler**$(h_t, V^{type}_p, V^{feas}_p, n_v)$
for $V \in V_p$ **do**
$C = C \bigcup (p, V, V^{type}_p)$
$(p, V, V^{type}_p) = \arg\max(C)$
$u^{key}_p, u^{val}_p, u^{type}_p =$ **OutVarGen**(p, V^{type}_p, V)
WriteVarToMem$(u^{key}_p, u^{val}_p, u^{type}_p)$
$e_{t+1}, h_{t+1} =$ **NPICore**(e_t, h_t)
A.append$((p, V))$
Output: A

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/tacl_a_00262 by guest on 29 March 2024

variable instantiations, resulting in a total of $n_p * n_v$ candidates out of which b most-likely actions are sampled for the b beams and the corresponding newly generated variables written into memory. This way the algorithm progresses to finally output b candidate programs, each of which will feed the model back with some reward. Finally, in order to learn from the discrete action samples, the REINFORCE objective (Williams, 1992) is used. Because of lack of space, we do not provide the equation for REINFORCE, but our objective formulation remains very similar to that in Liang et al. (2017). We next describe several learning challenges that arise in the context of this overall architecture.

5 Mitigating Large Program Space and Sparse Reward

Handling complex queries by expanding the operator set and generating longer programs blows up the program space to a huge size of $(num_op * (max_var)^m)^T$. This, in absence of gold programs, poses serious training challenges for the programmer. Additionally, whereas the relatively simple NSM architecture could explore a large beam size (50–100), the complex architecture of CIPITR entailed by the CPI problem could only afford to operate with a smaller beam size (≤ 20), which further exacerbates the sparsity of the reward space. For example, for integer answers, only a single point in the integer space returns a positive reward, without any notion of partial reward. Such a delayed—indeed, terminal—reward causes high variance, instability, and local minima issues. A problem as complex as ours requires not only generic constraints for producing semantically correct programs, but also incorporation of prior knowledge, if the model permits. We now describe how to guide CIPITR more efficiently through such a challenging environment using both generic and task-specific constraints.

Phase change network: For complex real-word problems, the reinforcement learning community has proposed various task-abstractions (Parr and Russell, 1998; Dietterich, 2000; Bakker and Schmidhuber, 2004; Barto and Mahadevan, 2003; Sutton et al., 1999) to address the curse of dimensionality in exponential action spaces. HAMs, proposed by Parr and Russell (1998), is one such important form of abstraction aimed at restricting

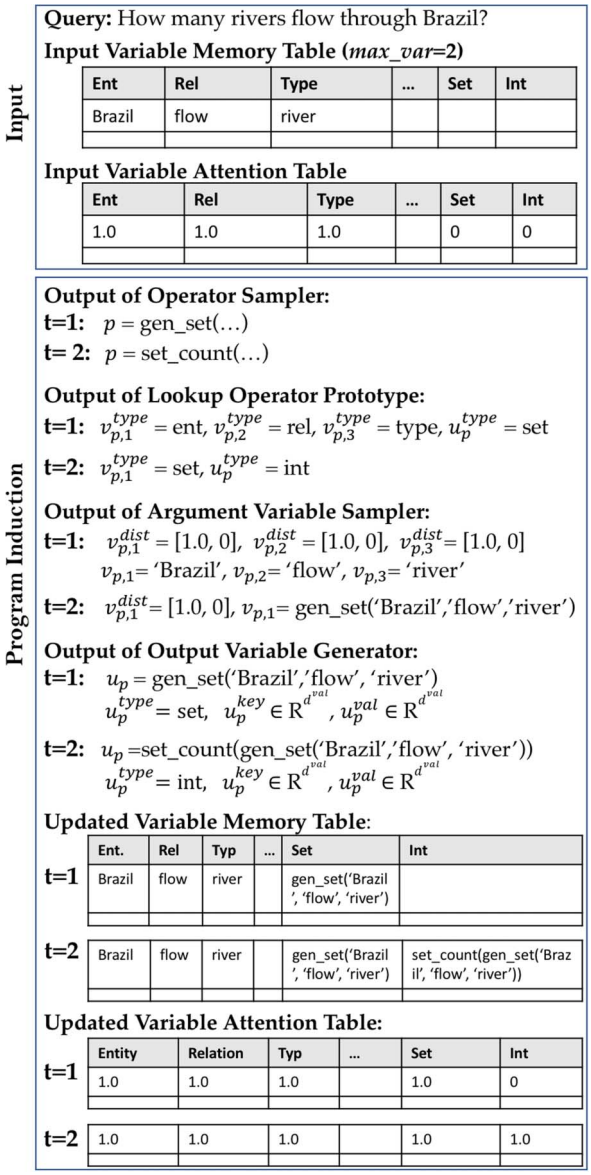


Figure 4: An example of a CIPITR execution trace depicting the internals of memory and action sampling to generate the program: $(A = gen_set(Brazil, flow, river), B = set_count(A))$.

the realizable action sequences. Inspired by HAMs, we decompose the program synthesis into phases having restricted action spaces. The first phase (*retrieval phase*) constitutes gathering the information from the preprocessed input variables only (i.e., KB entities, relations, types, integers). This restricts the feasible operator set to gen_set, gen_map_set, and $verify$. In the second phase (*algorithm phase*) the model is allowed to operate on all the generated variables in order to reach the answer. The programmer learns whether to switch from the first phase to the second at any timestep t, based on parameter ϕ_t ($\phi_t = 1$ indicating

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/tacl_a_00262 by guest on 29 March 2024

change of phase, where $\phi_0 = 0$) which is obtained as $\phi_t = \mathbb{1}\{\max(sigmoid(f(h_t)), \phi_{t-1}) \geq \phi_{thresh}\}$ if $t < T/2$, else 1 (T being total timesteps and ϕ_{thresh} is set to 0.8 in our experiments). The motivation behind this is similar to the multi-staged techniques that have been adopted in order to make QA tasks more tractable, as in Yih et al. (2015) and Iyyer et al. (2017). In contrast, here we further allow the model to learn when to switch from one stage to the next. Note that this is a generic characteristic, as for every task, this kind of phase division is possible.

Generating semantically correct programs: Other than the generic syntactical and semantic rules, the NPI paradigm also allows us to leverage prior knowledge and to incorporate task-specific symbolic constraints in the program representation learning in an end-to-end differentiable way.

- **Enforcing KB consistency:** Operators used in the retrieval phase (described above) must honor the KB-imposed constraints, so as not to initialize variables that are inconsistent with respect to the KB. For example, a set variable assigned from *gen_set* is considered valid only when the *ent*, *rel*, *type* arguments to *gen_set* are consistent with the KB.

- **Biasing the last operator using answer type predictor:** Answer type prediction is a standard preprocessing step in question answering (Li and Roth, 2002). For this we use a rule-based predictor that has 98% accuracy. The predicted answer type helps in directing the program search toward the correct answer type by biasing the sampling towards feasible operators that can produce the desired answer type.

- **Auxiliary reward strategy:** Jaccard scores of the executed program's output and the gold answer set is used as reward. An invalid program gets a reward of -1. Further, to mitigate the sparsity of the extrinsic rewards, an additional auxiliary feedback is designed to reward the model on generating an answer of the predicted answer-type. A linear decay makes the effect of auxiliary reward vanish eventually. Such a curriculum learning mechanism, while being particularly useful for the more complex queries, is still quite generic as it does not require any additional task-specific prior knowledge.

Beam Management and Action Sampling

- **Pruning beams by target answer type:** Penalize beams that terminate with an answer type not matching the predicted answer type.

- **Length-based normalization of beam scores:** To counteract the characteristic of beam search favoring shorter beams as more probable and to ensure the scoring is fair to the longer beams, we normalize the beam scores with respect to their length.

- **Penalizing beams for no_op operators:** Another way of biasing the beams toward generating longer sequences, is by penalizing for the number of times a beam takes no_op as the action. Specifically, we reduce the beam score by a hyperparameter-controlled logarithmic factor of the number of no_op actions taken till now.

- **Stochastic beam exploration with entropy annealing:** To avoid early local minima where the model severely biases towards specific actions, we added techniques like (i) a stochastic version of beam search to sample operators in an ϵ-greedy fashion (ii) dropout, and (iii) entropy-based regularization of action distribution.

Sampling only feasible actions: Sampling a feasible action requires first sampling a feasible operator and then its feasible variable arguments:

- The operator must be allowed in the current phase of the model's program induction.

- *Valid Variable instantiation*: A feasible operator should be having at least one valid instantiation of its formal arguments with non-empty variable values that are also consistent with the KB.

- *Action Repetition*: An action (i.e., an operator invoked with a specific argument instantiation) should not be repeated at any time step.

- Some operators disallow some arguments; for example, union or intersection of a set with itself.

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/tacl_a_00262 by guest on 29 March 2024

	Simple	Logical	Verify	Quanti	Quant Count	Comp	Comp Count	WebQSP All
Timesteps	2	4	5	7	7	7	7	3 to 5
Entropy-Loss Wt.	$5e^{-4}$	$5e^{-4}$	$5e^{-6}$	$5e^{-3}$	$5e^{-3}$	$5e^{-2}$	$5e^{-2}$	$5e^{-3}$
Feasible Program after iterations	1000	2000	500	1300	1300	1500	1500	50
Beam Pruning after iterations	100	100	100	1300	1300	1300	1000	100
Auxillary Reward till iterations	0	0	0	800	800	800	800	200
Learning Rate	$1e^{-5}$	$1e^{-5}$	$1e^{-5}$	$1e^{-5}$	$1e^{-5}$	$1e^{-5}$	$1e^{-5}$	$1e^{-4}$

Table 1: Critical hyperparameters.

6 Experiments

We compare CIPITR against baselines (Miller et al., 2016; Liang et al., 2017) on complex KBQA and further identify the contributions of the ideas presented in Section 5 via ablation studies. For this work, we limit our effort on KBQA to the setting where the query is annotated with the gold KB-artifacts, which standardizes the input to the program induction for the competing models.

6.1 Hyperparameters Settings

We trained our model using the Adam Optimizer and tuned all hyperparameters on the validation set. Some parameters are selectively turned on/ off after few training iterations, which is itself a hyperparameter (see Table 1). We combined reward/ loss such as entropy annealing and auxiliary rewards using different weights detailed in Table 1. The key, value embedding dimensions are set to 100, 300.

6.2 WebQuestionsSP Data Set

We first evaluate our model on the more popularly used WebQuestionsSP data set.

6.2.1 Rule-Based Model on WebQuestionsSP

Though quite a few recent works on KBQA have evaluated their model on WebQuestionsSP, the reported performance is always in a setting where the gold entities/relations are not known. They either internally handle the entity and relation-linking problem or outsource it to some external or in-house model, which itself might have been trained with additional data. Additionally, the entity/relation linker outputs used by these models are also not made public, making it difficult to set up a fair ground for evaluating the program induction model, especially because we are interested in the program induction given the program inputs and handling the entity/relation linking is beyond the scope of this work. To avoid these issues, we use the human-annotated entity/relation linking data available along with the questions as input to the program induction model. Consequently the performance reported here is not comparable to the previous works evaluated on this data set, as the query annotation is obtained here from an oracle linker.

Further, to gauge the proficiency of the proposed program induction model, we construct a rule-based model which is aware of the human annotated semantic parsed form of the query—that is, the inference chain of relations and the exact constraints that need to be additionally applied to reach the answer. The pseudocode below elaborates how the rule based model works on the human-annotated parse of the given query, taking as input the central entity, the inference chain, and associated constraints and their type. This

Procedure: RuleBasedModel(parse, KB)
$ent_1 \leftarrow parse[\text{`}TopicEntityMid\text{'}]$
$rel_1 \leftarrow parse[\text{`}InferentialChain\text{'}][0]$
$ans \leftarrow \{x \mid (ent_1, rel_1, x) \in KB\}$
for $c \in parse[\text{`}Constraints\text{'}]$
  $c_rel \leftarrow c[\text{`}NodePredicate\text{'}]$
  $c_op \leftarrow c[\text{`}Operator\text{'}]$
  $c_arg \leftarrow c[\text{`}Argument\text{'}]$
  if $c[\text{`}ArgumentType\text{'}] == \text{`}Entity\text{'}$
    $ans \leftarrow ans \cap \{x \mid (c_arg, c_rel, x) \in KB\}$
  else
    $ans \leftarrow \bigcup_{x \in ans} \{x \mid (x, c_rel, y) \in KB, c_arg \gtreqless_{c_op} y\}$
if $len(parse[\text{`}InferentialChain\text{'}]) > 1$
  $rel_2 \leftarrow parse[\text{`}InferentialChain\text{'}][1]$
  $ans \leftarrow \bigcup_{x \in ans} \{y \mid (x, rel_2, y) \in KB\}$
Output: ans

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/tacl_a_00262 by guest on 29 March 2024

Question Type	Rule Based	CIPITR
Inference-chain-len-1, no constraint	87.34	89.09
Inference-chain-len-1 with constraint	93.64	79.94
Inference-chain-len-2, no constraint	82.85	88.69
Inference-chain-len-2, with nontemporal constraint	61.26	63.07
Inference-chain-len-2, with temporal constraint	35.63	48.86
All	81.19	82.85

Table 2: F1 scores(%) of CIPITR and rule-based model(as in Sec.6.2.1) on WebQuestionsSP test set having 1,639 queries.

inference rule, manually derived, can be written out in a program form, which on execution will give the final answer. On the other hand, the task of CIPITR is to actually learn the program by looking at training examples of the query and corresponding answer. Both the models need to induce the program using the gold entity/relation data. Subsequently, the rule-based model is indeed a very strong competitor as it is generated by annotators having detailed knowledge about the KB.

6.2.2 Results on WebQuestionsSP

A comparative performance analysis of the proposed CIPITR model, the rule-based model and the SparQL executor is tabulated in Table 2. The main take-away from these results is that CIPITR is indeed able to learn the rules behind the multi-step inference process simply from the distance supervision provided by the question-answer pairs and even perform slightly better in some of the query classes.

6.3 CSQA Data Set

We now showcase the performance of the proposed models and related baselines on the CSQA data set.

6.3.1 Baselines on CSQA

KVMnet with decoder (2016), which performed best on CSQA data set (Saha et al., 2018) (as discussed in Section 2), learns to attend on a KB subgraph in memory and decode the attention over memory-entries as their likelihood of being in the answer. Further, it can also decode a vocabulary of non-KB words like integers or booleans. However, because of the inherent architectural constraints, it is not possible to incorporate most of the symbolic constraints presented in Section 5 in this model, other than KB-guided consistency and biasing towards answer-type. More importantly, recently the usage of these models have been criticized for numerical and boolean question answering as these deep networks can easily memorize answers without "understanding" the logic behind the queries simply because of the skew in the answer distribution. In our case this effect is more pronounced as CSQA evinces a curious skew in integer answers to "count" queries. Fifty-six percent of training and 52% of test count-queries have single digit answers. Ninety percent of training and 81% of test count-queries have answers less than 200. Though this makes it unfair to compare NPI models (that are oblivious to the answer vocabulary) with KVMnet on such queries, we still train a KVMnet version on a balanced resample of CSQA, where, for only the count queries, the answer distribution over integers has been made uniform.

NSM (2017) uses a key-variable memory and decodes the program as a sequence of operators and memory variables. As the NSM code was not available, we implemented it and further incorporated most of the six techniques presented in Table 4. However, constraints like action repetition, biasing last operator selection, and phase change cannot be incorporated in NSM while keeping the model generic, as it decodes the program token by token.

6.3.2 Results on CSQA

In Table 3 we compare the F1 scores obtained by our system, CIPITR, against the KVMnet and NSM baselines. For NSM and CIPITR, we train seven models with different hyperparameters tuned on each of the seven question types. For the train and valid splits, a rule-based query type classifier with 97% accuracy was used to bucket queries into the classes listed in Table 3. For each of these three systems, we also train and evaluate

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/tacl_a_00262 by guest on 29 March 2024

↓ Run name \ Question type →	Simple	Logical	Verify	Quanti.	Quant Count	Compar.	Comp Count	All
Training Size Stats.	462K	93K	43K	99K	122K	41K	42K	904K
Test Size Stats.	81K	18K	9K	9K	18K	7K	7K	150K
KVMnet	41.40	37.56	27.28	0.89	17.80	1.63	**9.60**	26.67
NSM, best at top beam	78.38	35.40	28.70	4.31	12.38	0.17	0.00	10.63
NSM best over top 2 beams	80.12	41.23	35.67	4.65	15.34	0.21	0.00	11.02
NSM, best over top 5 beams	86.46	64.70	50.80	6.98	29.18	0.48	0.00	12.07
NSM, best over top 10 beams	96.78	69.86	60.18	10.69	30.71	2.09	0.00	14.36
CIPITR, best at top beam	**96.52**	**87.72**	**89.43**	**23.91**	**51.33**	**15.12**	0.33	**58.92**
CIPITR, best over top 2 beams	96.55	87.78	90.48	25.85	51.72	19.85	0.41	62.52
CIPITR, best over top 5 beams	97.18	87.96	90.97	27.19	52.01	29.45	1.01	69.25
CIPITR, best over top 10 beams	97.18	88.92	90.98	28.92	52.71	32.98	1.54	73.71

Table 3: F1 score (%) of KVMnet and NSM, and CIPITR. Bold numbers indicate the best among KVMnet and top beam score of NSM and CIPITR.

one single model over all question types. KVMnet does not have any beam search, the NSM model uses a beam size of 50, and CIPITR uses only 20 beams for exploring the program space.

Our manual inspection of these seven query categories show that *simple* and *verify* are simplest in nature requiring 1-line programs while *logical* is moderately difficult, with around 3 lines of code. The query categories next in order of complexity are *quantitative* and *quantitative count*, needing a sequence of 2–5 operations. The hardest types are *comparative* and *comparative count*, which translate to an average of 5–10 lined programs.

Analysis: The experiments show that on the simple to moderately difficult (i.e., first three) query classes, CIPITR's performance at the top beam is up to 3 times better than both the baselines. The superiority of CIPITR over NSM is showcased better on the more complex classes where it outperforms the latter by 5–10 times, with the biggest impact (by a factor of 89 times) being on the "comparative" questions. Also, the $5\times$ better performance of CIPITR over NSM over *All* category evinces the better generalizability of the abstract high-level program decomposition approach of the former.

On the other hand, training the KVMnet model on the balanced data helps showcase the real performance of the model, where CIPITR outperforms KVMnet significantly on most of the harder query classes. The only exception is the hardest class (Comp, Count with numerical answers) where the abrupt "best performance" of KVMnet can be attributed to its rote learning abilities simply because of its knowledge of the answer vocabulary, which the program induction models are oblivious to, as they never see the actual answer.

Feature	F1 (%)
Action Repetition	1.68
Phase Change	2.41
Valid Variable Instantiation	3.08
Biasing last operator	7.52
Auxiliary Reward	9.85
Beam Pruning	10.34

Table 4: Ablation testing on comparative questions: Top beam's F1 score obtained by omitting each of the above features from CIPITR, originally having F1 of 15.123%.

Lastly, in our experimental configurations, whereas CIPITR and NSM's parameter-size is almost comparable, KVMnet's is approximately $6\times$ larger.

Ablation Study: To quantitatively analyze the utility of the features mentioned in Section 5, we experiment with various ablations in Table 4 by turning off each feature, one at a time. We show the effect on the hardest question category ("comparative") on which our proposed model achieved reasonable performance. We see in the table that each of the 6 techniques helped the model significantly. Some of them boosted F1 by 1.5–4 times, while others proved to be instrumental to obtained large improvements in F1 score of over 6–9 times.

To summarize, CIPITR has the following advantages, inducing programs more efficiently

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/tacl_a_00262 by guest on 29 March 2024

Ques Type	Input		CIPITR Program	NSM program
	Query	(E,R,T)		
Simple	Who is associated with Robert Emmett O'Malley ?	**E0:** Robert Emmett O'Malley **R0:** associated with **T0:** person	A = gen_set(E0, R0, T0)	A = gen_set(E0, R0, T0)
Verify	Is Sergio Mattarella the chief of state of Italy ?	**E0:** Sergio Mattarella, **E1:** Italy, **R0:** chief of state	A = verify(E0, R0, E1)	A = verify(E0, R0, E1)
Logical	Which cities were Animal Kingdom filmed on or share border with Pedralba ?	**E0:** Animal Kingdom, **E1:** Pedralba, **R0:** filmed_on, **R1:** Share_border, **T0:** cities	A = gen_set(E0, R0, T0); B = gen_set(E1, R1, T0); C = set_union(A,B)	A = gen_set(E1, R1, T0); B = gen_set(E1, R1, None); C = set_union(A,B)
Quant Count	How many nucleic acid sequences encodes Calreticulin or Neurotensin/neuromedin-N ?	**E0:** Calreticulin, **E1:** Neurotensin/neuromedin-N, **R0:** encoded_by, **T0:** nucleic acid	A = gen_set(E0, R0, T0); B = gen_set(E1, R0, T0); C = set_union(A,B); D = set_count(C)	A = gen_set(E0, R0, T0); B = set_count(A); C = set_union(A,B)
Quant	What municipal councils are the legislative bodies for max US administrative territories ?	**T0:** muncipal council, **T1:** US administrative territories **R0:** legislative_body of	A = gen_map_set(T0, R0, T1); B = map_count(A); C = select_max(B)	A = gen_map_set(T0, R0, T1); B = gen_map_set(T0, R0, T1); C = map_count(A)
Comp	Which works did less number of people do the dubbing for than Herculesy el rey de Tesalia ?	**E0:** Herculesy el rey de Tesalia, **R0:** dubbed_by, **T0:** works, **T1:** people	A = gen_map_set(T0, R0, T1); B = gen_set(E0, R0, T1); C = set_count(B); D= map_count(A); E = select_less(D,C)	A = gen_set(E0, R0, T1); B= gen_set(E0, R0, T1); C= gen_map_set(T0, R0, T1); D= set_diff(A,B)

Table 5: Qualitative analysis of programs for different type of question for CIPITR and NSM.

and pragmatically, as illustrated by the sample outputs in Table 5:

- **Generating syntactically correct programs**: Because of the token-by-token decoding of the program, NSM cannot restrict its search to only syntactically correct programs, but rather only resorts to a post-filtering step during training. However, at test time, it could still generate programs with wrong syntax, as shown in Table 5. For example, for the Logical question, it invokes a gen_set with a wrong argument type *None* and for the Quantitative count question, it invokes the set_union operator on a non-set argument. On the other hand, CIPITR, by design, can never generate a syntactically incorrect program because at every step it implicitly samples only feasible actions.

- **Generating semantically correct programs**: CIPITR is capable of incorporating different generic programming styles as well as problem-specific constraints, restricting its search space to only semantically correct programs. As shown in Table 5, CIPITR is able to generate at least meaningful programs having the desired answer-type or without repeating lines of code. On the other hand the NSM-generated programs are often semantically wrong, for instance, both in the Quantitative and Quantitative Count based questions, the type of the answer is itself wrong, rendering the program meaningless. This arises once again, owing to the token-by-token decoding of the program by NSM which makes it hard to incorporate high level rules to guide or constrain the search.

- **Efficient search-space exploration**: Owing to the different strategies used to explore the program space more intelligently, CIPITR scales better to a wide variety of complex queries by using less than half of NSM's beam size. We experimentally established that for programs of length 7 these various techniques reduced the average program space from 1.33×10^{19} to 2,998 programs.

7 Conclusion

We presented CIPITR, an advanced NPI framework that significantly pushes the frontier of complex program induction in absence of gold programs. CIPITR uses auxiliary rewarding techniques to mitigate the extreme reward sparsity and incorporates generic pragmatic programming styles to constrain the combinatorial program space to only semantically correct programs. As future directions of work, CIPITR can be further improved to handle the hardest question types by making the search more strategic, and can be further generalized to a diverse set of goals when training on all question categories together.

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/tacl_a_00262 by guest on 29 March 2024

Other potential directions of research could be toward learning to discover sub-goals to further decompose the most complex classes beyond just the two-level phase transition proposed here. Additionally, further improvements are required to induce complex programs without availability of gold program input variables.

References

Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. 2016a. Learning to compose neural networks for question answering. In *NAACL HLT 2016, The 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1545–1554.

Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. 2016b. Learning to compose neural networks for question answering. In *NAACL-HLT*, pages 1545–1554.

B. Bakker and J. Schmidhuber. 2004. Hierarchical reinforcement learning based on subgoal discovery and subpolicy specialization. In *Proceedings of the 8th Conference on Intelligent Autonomous Systems IAS-8*, pages 438–445.

Andrew G. Barto and Sridhar Mahadevan. 2003. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 13(1-2):41–77.

Hannah Bast and Elmar Haußmann. 2015. More accurate question answering on freebase. In *CIKM*, pages 1431–1440.

J. Berant, A. Chou, R. Frostig, and P. Liang. 2013. Semantic parsing on Freebase from question-answer pairs. In *EMNLP Conference*, pages 1533–1544.

Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. 2013. Translating embeddings for modeling multi-relational data. In *NIPS Conference*, pages 2787–2795.

Matko Bosnjak, Tim Rocktäschel, Jason Naradowsky, and Sebastian Riedel. 2017. Programming with a differentiable forth interpreter. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017*, pages 547–556.

Rudy Bunel, Matthew J. Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. 2018. Leveraging grammar and reinforcement learning for neural program synthesis. In *International Conference on Learning Representations (ICLR)*.

Rajarshi Das, Manzil Zaheer, Siva Reddy, and Andrew McCallum. 2017. Question answering on knowledge bases and text using universal schema and memory networks. In *ACL (2)*, pages 358–365.

Peter Dayan and Geoffrey E. Hinton. 1993. Feudal reinforcement learning. In *Advances in Neural Information Processing Systems 5, [NIPS Conference]*, pages 271–278.

Thomas G. Dietterich. 2000. Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of Artificial Intelligence Research*, 13(1):227–303.

Li Dong and Mirella Lapata. 2016. Language to logical form with neural attention. In *ACL*, volume 1, pages 33–43.

Kelvin Guu, John Miller, and Percy Liang. 2015. Traversing knowledge graphs in vector space. In *EMNLP Conference*.

Mohit Iyyer, Wen-tau Yih, and Ming-Wei Chang. 2017. Search-based neural structured learning for sequential question answering. In *ACL*, volume 1, pages 1821–1831.

Sarvnaz Karimi, Justin Zobel, and Falk Scholer. 2012. Quantifying the impact of concept recognition on biomedical information retrieval. *Information Processing & Management*, 48(1): 94–106.

Mahboob Alam Khalid, Valentin Jijkoun, and Maarten De Rijke. 2008. The impact of named entity normalization on information retrieval for question answering. In *Proceedings of the IR Research, 30th European Conference on Advances in Information Retrieval*, ECIR'08, pages 705–710.

Brenden M. Lake, Ruslan Salakhutdinov, and Joshua B. Tenenbaum. 2015. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338.

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/tacl_a_00262 by guest on 29 March 2024

Chengtao Li, Daniel Tarlow, Alexander L. Gaunt, Marc Brockschmidt, and Nate Kushman. 2016. Neural program lattices. In *International Conference on Learning Representations (ICLR)*.

X. Li and D. Roth. 2002. Learning question classifiers. In *COLING*, pages 556–562.

Chen Liang, Jonathan Berant, Quoc Le, Kenneth D. Forbus, and Ni Lao. 2017. Neural symbolic machines: Learning semantic parsers on freebase with weak supervision. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, pages 23–33.

Andrew McCallum, Arvind Neelakantan, Rajarshi Das, and David Belanger. 2017. Chains of reasoning over entities, relations, and text using recurrent neural networks. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics, EACL 2017, Volume 1: Long Papers*, pages 132–141.

Alexander H. Miller, Adam Fisch, Jesse Dodge, Amir-Hossein Karimi, Antoine Bordes, and Jason Weston. 2016. Key-value memory networks for directly reading documents. In *EMNLP*, pages 1400–1409.

Stephen Muggleton and Luc De Raedt. 1994. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19/20: 629–679.

Arvind Neelakantan, Quoc V. Le, Martin Abadi, Andrew McCallum, and Dario Amodei. 2016. Learning a natural language interface with neural programmer. *arXiv preprint*, arXiv: 1611.08945.

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

Ronald Parr and Stuart Russell. 1998. Reinforcement learning with hierarchies of machines. In *Proceedings of the 1997 Conference on Advances in Neural Information Processing Systems 10*, NIPS '97, pages 1043–1049.

Panupong Pasupat and Percy Liang. 2015. Compositional semantic parsing on semi-structured tables. *arXiv preprint* arXiv:1508.00305.

Marc'Aurelio Ranzato, Sumit Chopra, Michael Auli, and Wojciech Zaremba. 2015. Sequence level training with recurrent neural networks. *CoRR*, abs/1511.06732.

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

Amrita Saha, Vardaan Pahuja, Mitesh M. Khapra, Karthik Sankaranarayanan, and Sarath Chandar. 2018. Complex sequential question answering: Towards learning to converse over linked question answer pairs with a knowledge graph. In *AAAI*.

Richard S. Sutton, Doina Precup, and Satinder Singh. 1999. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1-2):181–211.

Reut Tsarfaty, Ilia Pogrebezky, Guy Weiss, Yaarit Natan, Smadar Szekely, and David Harel. 2014. Semantic parsing using content and context: A case study from requirements elicitation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1296–1307.

Richard J. Waldinger and Richard C. T. Lee. 1969. PROW: A step toward automatic program writing. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, pages 241–252.

Ronald J Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. In *Reinforcement Learning*, Springer, pages 5–32.

Kun Xu, Siva Reddy, Yansong Feng, Songfang Huang, and Dongyan Zhao. 2016. Question answering on Freebase via relation extraction and textual evidence. *arXiv preprint*, arXiv: 1603.00957.

Xuchen Yao. 2015. Lean question answering over Freebase from scratch. In *NAACL Conference*, pages 66–70.

Scott Wen-tau Yih, Ming-Wei Chang, Xiaodong He, and Jianfeng Gao. 2015. Semantic parsing via staged query graph generation: Question

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/tacl_a_00262 by guest on 29 March 2024

answering with knowledge base. In *ACL Conference*, pages 1321–1331.

Wen-tau Yih, Matthew Richardson, Chris Meek, Ming-Wei Chang, and Jina Suh. 2016. The value of semantic parse labeling for knowledge base question answering. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, volume 2, pages 201–206.

Downloaded from http://www.mitpressjournals.org/doi/pdf/10.1162/tacl_a_00262 by guest on 29 March 2024