Neural Program Induction for KBQA Without Gold Programs or Query Annotations

Ghulam Ahmed Ansari[*1], **Amrita Saha**[*1], **Vishwajeet Kumar**[2], **Mohan Bhambhani**[1], **Karthik Sankaranarayanan**[1] and **Soumen Chakrabarti**[2]
[1]IBM Research, India
[2]Indian Institute of Technology Bombay, India
{ansarigh, amrsaha4, mbhamb09, kartsank}@in.ibm.com
{vishwajeetkumar86, soumen.chakrabarti}@gmail.com

Abstract

Neural Program Induction (NPI) is a paradigm for decomposing high-level tasks such as complex question-answering over knowledge bases (KBQA) into executable programs by employing neural models. Typically, this involves two key phases: i) inferring input program variables from the high-level task description, and ii) generating the correct program sequence involving these variables. Here we focus on NPI for Complex KBQA with only the final answer as supervision, and not gold programs. This raises major challenges; namely i) noisy query annotation in the absence of any supervision can lead to catastrophic forgetting while learning, ii) reward becomes extremely sparse owing to the noise. To deal with these, we propose a noise-resilient NPI model, Stable Sparse Reward based Programmer (SSRP) that evades noise-induced instability through continual retrospection and its comparison with current learning behavior. On complex KBQA datasets, SSRP performs at par with hand-crafted rule-based models when provided with gold program input, and in the noisy settings outperforms state-of-the-art models by a significant margin even with a noisier query annotator.

1 Introduction

Recently, the neural program induction (NPI) paradigm has gained significant interest for approaching complex tasks through programmatic decomposition, i.e., generating a sequence of atomic operations on program variables, which upon execution yield the answer. This provides a practical approach to solving complex tasks while providing better interpretability than one-shot inferences using deep networks. Consequently, NPI techniques have been employed for variety of tasks such as Addition, Sorting [Reed and De Freitas, 2015], GridWorld navigation [Bunel *et al.*, 2018], Tabular QA [Neelakantan *et al.*, 2016; He *et al.*, 2018], Math word problems [Bosnjak *et al.*, 2017] or KBQA [Liang *et al.*, 2017]. However, two critical assumptions have been commonly made by them to keep the problem tractable:

1) Either the input program variables to the NPI process is known or trivial to infer, or the inference is outsourced to other pretrained models. For example, in learning addition or sorting, or for navigating the GridWorld, the input program variables are directly provided. For tabular QA, the question annotation is still manageable as the query-words need to be linked to table cell and column values of a quite small (typically $10\times$ 5 sized) table.

2) Either gold programs or program skeletons are used to train the NPI model, or training is constrained to relatively simple programs. But realistically, for various complex applications, these assumptions may not hold, or may be too expensive to ensure. E.g., complex KBQA requires as input to the NPI, an annotation of the query with KB entities (E), relations (R) and types (T), a.k.a. *ERT linking* using huge KBs containing millions of entities. This annotation is particularly non-trivial and noisy in an unsupervised setting. Further, the complex nature of the questions require generation of long, multi-line programs, for which providing gold program supervision is expensive. Neural Symbolic Machine (NSM) by [Liang *et al.*, 2017] is the only complex KBQA system that can be trained with only final answer as supervision and noisy query annotation as input. But it is not a true NPI model, as it decodes the program token-by-token and is unable to incorporate high-level structure and programmatic styles when decoding.

The primary contribution of our work is a noise-resilient NPI model that is distant-supervised by the final answer alone. It can handle noise in the program input so extreme, that only for 10–20% of the training or test questions the exact gold answer can be reached. We next elaborate on the steep challenges faced in this problem setting.

Effect of Noise on Program Space

The accuracy of the initial step of inferring program variables has severe repercussions on the program induction process. Even with the availability of gold input-variables, the search space of programs blows up to an exponential size. In the noisy data setting, the problem becomes compounded, since instead of the gold input program variables there are multiple possible candidates for each of the program input. E.g., to generate a 3-line program in the noisy setting with 7 operators and 5 possible candidate values for each input program variable, the program space explodes to 6.7×10^8 programs.

∗ These authors contributed equally to this work

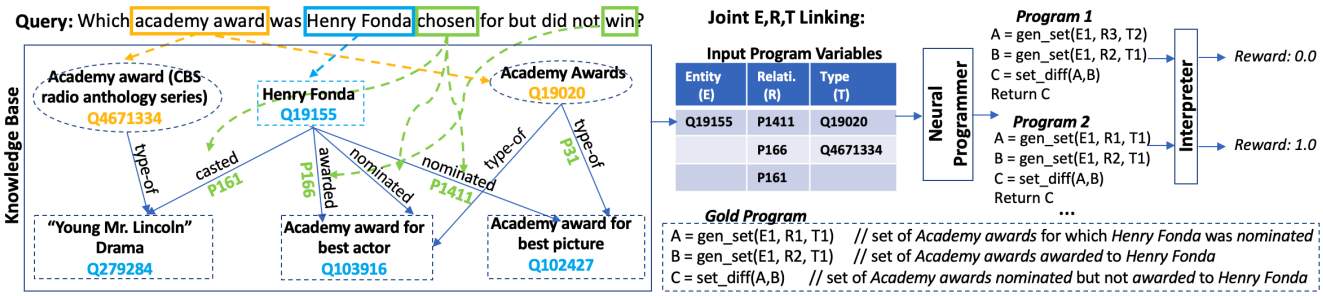


Figure 1: Complex KBQA by identifying the input program variables through joint ERT linking, then learning to induce programs from it, which on execution leads to the answer with associated reward. (Note that the *Gold Program* is not available during training or test)

Effect of Noise on Reward Sparsity

NPI models typically learn through reinforce-style updates [Williams, 1992], owing to the discrete nature of the feedback (or rewards) obtained by evaluating the induced programs. But inducing programs without gold programs for training suffers from an extremely sparse reward space, i.e., only a negligible number of programs in the entire exponential program space can yield rewards. Out of the enormous number of possible programs, only one or very few having a correct operator sequence invoked on the correct set of input and intermediate program variables get any reward. This renders the NPI problem extremely hard in the noisy setting and also requires the query annotation to be highly accurate.

Effect of Extreme Reward Sparsity on Learning

This nature of extreme sparsity in the reward space poses learning challenges by making the model unstable and prone to local optima problems. Further, the noise in the input has a more debilitating effect, by increasing the variance in the system. It can sometimes even fatally cripple the learner by confounding the model as it can receive counter-intuitive rewards even upon exploring the correct operator sequence.

Figure 1 illustrates the two phases of the complex KBQA task for a question *Which academy award was Henry Fonda chosen for but did not win?* which requires multi-step inference using logical reasoning over the KB. The first step of ERT linking requires a joint inference to resolve *Henry Fonda* as the American actor and identify phrases like *academy award* to be a potential KB-type or words like *win* or *chosen* to be linked to KB-relations *nominated* or *awarded* or *casted*. However, this joint reasoning, based on semantic similarity and KB connectivity, may not be always sufficient to make the correct inference. The situation escalates in the absence of any gold ERT linking supervision. E.g., in Figure 1, *chosen* has been linked to both *casted:P161* and *nominated:P1411* as potential candidates and *Academy Award* to a radio series and the film award. The output of the first phase is the input program variables memory consisting of *all* potential ERT candidates, which is fed into the NPI model, along with the gold answer. The NPI model then explores different programs that are logically consistent and compliant with the task description (i.e. KB, query, answer-type) and obtains a positive reward if it is able to reach the correct answer on execution. For e.g., the true program for the query in Figure 1 is the 2nd one and not the 1st, despite having the same operator sequence.

Complex KBQA Datasets

1) *WebQuestionsSP* [Yih *et al.*, 2016] provided a complex KBQA dataset having around 5K questions answerable from Freebase. While the natural language form of the questions look simple, for e.g. *Which team does David Beckham play for?*, it often requires up to 2-hop inference chains, sometimes with additional constraints.

2) *Complex Sequential QA (CSQA)* [Saha *et al.*, 2018] provided a dataset with 1.5M complex conversational QA pairs, requiring logical and quantitative reasoning over WikiData. For simplicity, we use the publicly available subset CQA-12K (comparable in size to WebQuestionsSP) consisting of 12K QA pairs from each of the seven question categories, where the questions do not depend on the previous context. We use the smaller dataset because the initial step of unsupervised query annotation is very time-consuming. The diverse categories of complex questions and the massive scale KB makes CQA-12K particularly suited to study Complex KBQA.

2 Related Work

2.1 Joint Entity, Relation, Type (ERT) Linking

While entity, relation and type linking has been around for decades, either the task is focused on (i) linking corpus mentions to KB, leveraging distant supervision signals from large scale corpus, for e.g. [Wu *et al.*, 2018; Wang *et al.*, 2018; Titov and Le, 2018; Radhakrishnan *et al.*, 2018] or (ii) linking short query text to KB artifacts by using gold linking data which is often expensive to obtain [Dubey *et al.*, 2018; Yu *et al.*, 2017; Yih *et al.*, 2015; Sorokin and Gurevych, 2017]. We are, however, interested in an unsupervised joint ERT linking on a query without leveraging any corpus.

2.2 Neural Program Induction

First generation semantic parsing systems, e.g., [Yih *et al.*, 2016], for modularizing complex tasks like complex KBQA using hand-crafted rules, have been popular for decades. Of the more recent neural methods of task decomposition into a program, the most relevant is Neural Symbolic Machines (NSM) [Liang *et al.*, 2017]. They learn to translate the query to a program like logical form executable on the KB, with only answer as supervision. Their choice of KBQA task and answer-supervised program induction setting makes NSM the

closest comparable work to ours. The two main distinctive factors of our work with respect to NSM are: i) NSM was applied to simpler questions and has several limitations which makes generalization to more complex programs hard, and ii) the input program variable creation was outsourced to an in-house annotator [Iyyer *et al.*, 2017] trained on gold entity, relation and type linking data and having near-oracle accuracy (94% on entity linking). This greatly alleviates the noise-related issues arising in the program induction itself.

In contrast, the core challenge in our work arises from the absence of gold ERT linking data and the downstream NPI model having to handle realistic levels of labeling noise shown by ERT linkers on complex questions. This level of noisy-input in complex program induction for QA on a large scale KB, has not been addressed before.

3 Model Overview

Following are the main components of the model:
1) The *ERT linker* is an unsupervised query annotator that links the query spans with KB entities, relations and types and pre-populates them as input variables to the NPI model.
2) The *programmer* generates programs using the natural language query, KB, and pre-populated variables in memory as input. A program is a sequence of operators invoked with past instantiated variables as their arguments and by generating new variables which are written into memory.
3) The *interpreter* executes the generated program using the KB and outputs an answer, which upon comparison with the gold answer yields a reward. During training this reward is sent back to the programmer to update its parameters through a REINFORCE objective [Williams, 1992].

4 Entity Relation Type (ERT) Linking

Step 1: Independent E,R,T Linking generates separate candidate-lists for each of the KB entity, relation and type in the query. This step annotates all the longest possible query n-grams or shallow-parsed chunks that have a GloVe cosine score $>$0.3 with any KB entity, relation or type.
Step 2: Joint E,R,T linking in an unsupervised setting ranks the extracted entities, relations and types using the sum of the following scores:

- Semantic or surface-match score from Step 1.
- KB-connectivity score of a candidate entity, relation or type for a given query-span based on its hop-distance with candidate entities or relations or types associated with other spans or the entity-relation-type or type-relation-type connection with other relations or types.

5 Stable Sparse Reward Based Programmer

In this section, we describe in detail the proposed NPI model which we call Stable Sparse Reward based Programmer (SSRP). The candidate entities, relations and KB-types obtained from the previous step and the query are provided as input program variables to SSRP which then induces the program. SSRP consists of the following two components: (i) the core program induction algorithm whose sole supervision is from rewards obtained by evaluating induced programs with respect to the final answer, which we call Sparse Reward based Programmer (SRP) and is described in Section 5.1, and (ii) a generic noise-resilient wrapper over SRP which counteracts the noise encountered in the complex KBQA setting and stabilizes SSRP, which is described in Section 5.2.

5.1 Sparse Reward Based Programmer

We first describe the building blocks of SRP.
Seven Variable-Types

- KB artifacts: ent, rel, $type$
- Base data types: int, $bool$, $None$ (for padding)
- Composite data types: set i.e. Set of KB entities

Seven Operators

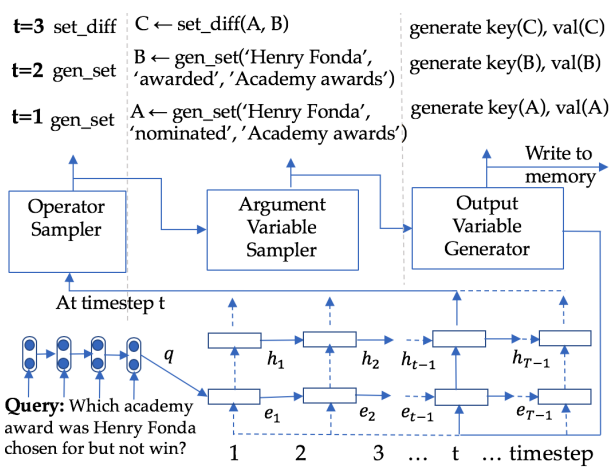
- $gen\_set(ent, rel, type) \rightarrow set$
- $verify(ent, rel, ent) \rightarrow bool$
- $set\_\{union/intersec/diff\}(set, set) \rightarrow set$
- $set\_count(set) \rightarrow int$
- $terminate()$

The core part of the NPI is a recurrent model which can access the following at each step: i) embedding matrices encoding a vocabulary of operators and variable types, ii) operator prototype matrices M^{op_arg} and M^{op_out} for storing the argument and output variable type information respectively for each operator, and iii) variable memory matrix which is a query-specific scratch (dynamic) memory for storing new program variables as they get created. For each variable type it learns separate key and value embedding matrices respectively, for looking up a variable in memory and accessing its information. Additionally it also has a matrix to store the attention over the variables declared of each type.

SRP Pipeline and Training

The algorithm in Fig. 2 outlines the pseudocode of the SRP invoking the following steps. The figure on its left, illustrates the same process, and in the following description we refer to both.

- SRP encodes the input query into a fixed dimensional representation q using a GRU network.
- The other essential input to SRP is the scratch memory for storing program variables, which is initialized with the input variables (KB entities, relations and types), when the programmer starts the induction process.
- The action taken by SRP at time-step t is markovially conditioned on the environment state e_t and the hidden representation of the current program state h_t, each of which are encoded by recurrent networks initialized with the query encoding. Given the environment and program state, the NPI model then generates an output program which is a sequence of actions, each of which involves an operator invoked over previously defined variables in memory. This process creates a new variable, at each time-step which is added to the memory to be used later.
- At each time-step, conditional to program state h_t, the model first samples a set of n_p operators using **OperatorSampler**. For e.g. gen_set at first step in Fig 2
- Then for each of the sampled operators, for e.g. p, SRP understands the feasibility of all possible variable instantiations for its m arguments by invoking **FeasibleVariable**. Feasibility of a variable instantiation for a given operator p is a boolean vector (V_p^{feas} in the pseu-



Query Encoding: $q = GRU(Query)$
Initialization: $e_1, h_1 = Linear(q), A = [\,]$
for $t \in 1, \cdots, T$ **do**
  P_t =**OperatorSampler**(h_t, n_p), $C = \{\}$
  for $p \in P_t$ **do**
    $V_p^{type} = [v_{p,1}^{type}, \cdots, v_{p,m}^{type}] = M^{op\_arg}[p]$
    V_p^{feas} =**FeasibleVariable**(p)
    V_p =**ArgVarSampler**$(h_t, V_p^{type}, V_p^{feas}, n_v)$
    for $V \in V_p$ **do**
      $C = C \bigcup (p, V, V_p^{type})$
  $(p, V, V_p^{type}) = \arg\max(C)$
  u_p^{key}, u_p^{val} =**OutVarGen**$(p, V_p^{type}, V), u_p^{type} = M^{op\_out}[p]$
  WriteVarToMem$(u_p^{key}, u_p^{val}, u_p^{type})$
  $e_{t+1} = GRU(e_t, u_p^{val}), h_{t+1} = GRU(e_t, u_p^{val}, h_t)$
  A.append$((p, V))$
Output: A ▷ Generated action sequence

Figure 2: SRP model architecture and pseudocode: Starting with the query encoding, at each time-step the model samples an action based on the environment e_t and program state h_t. Each action involves sampling the operator, and previously defined variables as its arguments and creating the output variable which is then added to the scratch memory. Figure outlines the pipeline with solid line for the t'th timestep.

docode), governed by the KB, to ensure that no variable is created that is inconsistent with the KB. For e.g. *gen_set('Henry Fonda', 'nominated','Academy award')* (where *Academy award* refers to the CBS series) is not valid as per the KB. Other programmatic paradigms like ensuring non-repetition of lines of code are also used to determine feasibility.

- For the operator p, **ArgVarSampler** then samples n_v (hyperparameter) feasible variable instantiations by attending over all variables of the corresponding types stored in memory. The variable type of the arguments V_p^{type} is obtained by looking up the operator prototype matrix M^{op_arg}. For e.g. in the fig. the sampled action is *gen_set('Henry Fonda, 'nominated', 'Academy awards')* at t=1 (n_v=1 for simplicity). The pseudocode shows the sampled variable instantiations V_p.
- After applying the sampled operator over the sampled variable instantiations, the **OutVarGen** module generates the key and value embedding of the new variable u_p^{key} and u_p^{val}, which are functions of the embedding of the sampled operator p and operand variables V.
- The **WriteVarToMem** module writes the key and value embedding of the newly generated variable to the dynamic scratch memory and correspondingly updates the attention over the memory variables of that type.
- The actions are sampled till either the *terminate* is sampled or for the maximum allowable timesteps, thus generating the program to be evaluated by the interpreter.

In order to get feedback from multiple candidate programs using single training instance, SRP employs a beam search i.e. at each step, out of the total number of candidates generated as above, K (typically 20) most-likely actions are sampled for the K beams and the corresponding newly generated variables written into memory. Thus the algorithm progresses till T steps to finally output K candidate programs each of which feed the model back with some reward. To learn from the discrete rewards, the REINFORCE objective is used.

Mitigating Sparsity in SRP

The following steps make SRP's exploration of the program space more efficient and mitigate the sparsity problem

- Decomposing the program generation into two predetermined phases, first one involving only operations over input program variables, and second phase operating on the variables created by the first phase.
- Generating semantically correct programs by (a) Incorporating programmatic paradigms like disallowing repeating or useless actions (e.g intersection of a set with itself) (b) Biasing the model towards generating answers of the desired variable type using auxiliary rewards.
- Penalizing for terminating in the wrong variable type as answer or generating shorter programs.
- Entropy and dropout based regularization.

5.2 Noise-stabilizing Wrapper of SSRP

We now discuss the multiple challenges in the program induction process caused by the presence of noise in the input program variable. In the noise-free setting itself, the program space is exponential with the number of gold input program variables. The noisy input data blows up the program variable space furthermore, with the manifold combinations of the candidate inputs. Additionally, in the absence of gold programs, the noisy setting raises another serious issue of exacerbating the reward space sparsity, where, even with gold input data, only a handful of programs in the exponential space could have yielded a positive reward. The program explosion compounded with extreme reward sparsity can easily render the learning unstable, even more so in reinforce-style algorithms by increasing variance. For instance, by following the same *good* operator sequence the model will get a positive reward, if it had used the correct input variable candidates but no reward on selecting the wrong input data, which is bound to confound the model.

To counteract this phenomenon, we propose SSRP, having a noise-resiliency wrapper over SRP that introduces a con-

Algorithm 1 SSRP Algorithm

θ,θ^{ref} denotes parameters of the Current/Reference Programmer

for $n \in 1, \cdots, N$ **do** ▷ loop over mini-batch
  $\overline{P}^{cur} : [P_0^{cur} \cdots P_K^{cur}] \leftarrow$ CurrentProgrammer(q, θ)
  $\overline{P}^{ref} : [P_0^{ref} \cdots P_K^{ref}] \leftarrow$ ReferenceProgrammer(q, θ^{ref})
  for $\forall i \in 1 \cdots K$ and $\forall j \in 1 \cdots K$ **do**
    $c_{ij} = 1$
    $d_{ij} \leftarrow$ Distance(P_i^{cur}, P_j^{ref})
    $r_i^{cur} \leftarrow$ Reward_Func(P_i^{cur}, q)
    $r_j^{ref} \leftarrow$ Reward_Func(P_j^{ref}, q)
    if $r_i^{cur} \leq 0$ **then** ▷ need safe back-prop
      if $r_i^{cur} - r_j^{ref} + \epsilon > 0$ **then**
        $\delta_{ij} \leftarrow r_i^{cur} - r_j^{ref}$
      else
        $\delta_{ij} \leftarrow 0$
      $c_{ij} \leftarrow e^{-\alpha d_{ij}} \delta_{ij}$ ▷ α is a hyperparameter
  $c_i \leftarrow \max_j(c_{ij})$
  if $n \mod Update\_Freq == 0$ **then**
    $\theta^{ref} \leftarrow \theta$

Output: $\overline{c} : [c_0 \cdots c_i \cdots c_K]$

cept of a Reference Programmer. The Reference Programmer is a snapshot (periodically refreshed) of an older version of the SRP model that is currently undergoing training, which we will call Current Programmer. In order to avoid unlearning, for a given training instance, the model retrospectively takes a decision about extent of backpropagation, based on two factors: i) *distance* between its current programs $P_0^{cur} \cdots P_K^{cur}$($K$ being beam size) and the programs $P_0^{ref} \cdots P_K^{ref}$ seen by the Reference Programmer, and ii) difference between the rewards obtained by the reference and the current model. The intuition here being that when the current reward is non-positive but higher than the reference reward, the confidence in backpropagation ($\overline{c} = [c_0 \cdots c_K]$) increases proportional to the reward difference but exponentially decreases with the program distance. Here, jaccard distance between the operator sequence is used as program distance. α is a cautiousness hyperparameter, for which, higher the value, more conservative the model is in updating itself, when the current model's exploration diverges from its past behavior. Also, as the reward difference decreases, the need for the model to update itself diminishes, as it has not improved much beyond its old version. This sort of retrospective controlling of gradient, by multiplying with the confidence vector $\overline{c}$ helps stabilize the model when dealing with crippling noise in the data.

Comparison with Advantage Actor-Critic methods. We also empirically compare our SSRP[1] with an SRP version trained with the more stable RL algorithms like Advantage Actor Critic($A2C$)[Konda and Tsitsiklis, 2003] that can handle noise by reducing variability in the system. For our $A2C$ baseline, we use the standard rewards (and no auxiliary rewards, refer end of Sec 5.1) provided by the environment for optimization. In this version, an additional network in SRP estimates the advantage function used in the objective.

[1]code and supplementary material available at https://github.com/CIPITR/SSRP

6 Experiments

In this section we describe separate experimental setup having gold or noisy program input to extensively evaluate the proposed model SSRP with NSM [Liang *et al.*, 2017] the closest known baseline, as well as a hand-crafted rule-based model. We also compare SSRP with SRP to study the impact of employing the noise-resiliency wrapper. We evaluate on two datasets: i) the popularly used WebQuestionsSP which requires upto 2-hop inferences over KB, sometimes with additional constraint satisfaction requirements, and ii) the recently introduced CSQA for complex KBQA. For experimental simplicity, we selected the publicly available subset CQA-12K of the original CSQA dataset, which contains 12K QA pairs from each question category and is comparable in size to WebQuestionsSP. Out of the 12K QA pairs, 10K pairs are used for training and 1K each for development and test. Since addressing the NPI in the absence of gold programs and on noisy input is a hard problem, we focus our evaluation on three query categories, requiring simple (single-step programs), logical (typically requiring 3 steps, for e.g. in Fig 1), and quantitative reasoning (requiring 4-5 steps - for e.g, *How many rivers originate in China and flow through Tibet?*).

Question Type	Rule-Based	SRP	SSRP
Inference-chain-len=1, no constraint	89.03	89.09	83.81
Inference-chain-len=1 with constraint	46.52	79.94	48.12
Inference-chain-len=2, no constraint	100.0	88.69	77.41
Inference-chain-len=2, with nontemporal constraint	71.31	63.07	40.65
Inference-chain-len<=2, with temporal constraint	83.83	48.86	57.42
All	82.59	82.85	72.61

Table 1: WebQuestionsSP Test F1 Scores(%) of rule-based model and proposed SRP on gold program input and SSRP on noisy program input. Competing model NSM, on noisy program input is reported to have 69% on overall Test

6.1 Results on WebQuestionsSP

With gold input data. We first train a SRP model with the gold program input and compare this model's performance with a rule-based model based human annotated semantic parsed form of the query. The latter knows the inference chain of relations and the exact constraints that need to be additionally applied to reach the answer. This inference rule that was manually derived, can be written out in a program form, which on execution will give the final answer. Table 1 shows that SRP performs at-par with the manually crafted parser even without supervision of gold programs.

With noisy input. Here we follow the training setup of NSM to train our SSRP model and test it on the noisy program input. One notable difference here is that while NSM uses a proprietary in-house entity linker reported to have a 94% recall, we can only use the best possible publicly available linker, [Xu *et al.*, 2016] having precision and recall respectively of (96.7%, 86%) for entity linking and (53.2%,

Model	Answerable				Full			
	Simple (83)	Logical (228)	QCount (96)	Combined (407)	Simple (1000)	Logical (1000)	QCount (1000)	Combined (3000)
SSRP	76.38	48.13	47.01	**44.7**	**14.54**	2.98	7.03	**4.6**
SRP	54.76	33.13	**51.25**	33.6	10.42	2.20	**7.33**	3.7
A2C	**82.5**	**52.4**	35.4	31.7	4.982	**3.3**	5.2	2.4
NSM	31.49	7.35	2.76	19.35	6.11	0.44	0.9	2.0

Table 2: F1 score (in %) of baseline NSM and A2C and the proposed models SRP and SSRP in the noisy setting. Boldfaced numbers indicate the best model.

	SRP	NSM
Simple	**96.52**	78.38
Logical	**87.72**	35.40
QCount	**51.33**	12.38

Table 3: F1-score (%) of NSM and SRP using gold ERT linking data as program input.

ERT	Train	Test
E	(19,79)	(17,40)
R	(15,47)	(16,44)
T	(16,61)	(24,51)

Table 4: Unsupervised ERT linker's (Precision, Recall) (in %) on CQA-12K

85%) for relation linking. Despite having substantially poorer recall in entity linking, SSRP surpasses NSM's performance reported in [Liang *et al.*, 2017], as shown in Table 1.

6.2 Results on CQA-12K

With gold input data. We first analyze the proposed SRP model along with the closest baseline NSM on each question category: simple, logical, and quantitative. To appropriately reflect their level of difficulty, we train and evaluate separate models on each question category. Further, in order to isolate the program induction performance from the noise resiliency, we first evaluate the models on input program variables obtained using oracle ERT linking. As reflected in Table 3, SRP outperforms NSM by a large margin of 20-50% F1 over the three question categories.

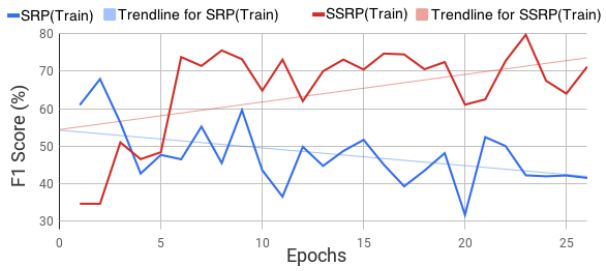


Figure 3: Training trend of SSRP vs SRP in noisy setting. SSRP is fairly immune to unlearning that clearly affects SRP.

With noisy input. We next show the effect of the noisy input ERT data for the baseline NSM and the proposed NPI model, with the noise-resilient wrapper i.e. SSRP and without it i.e. SRP in Table 2. We also compare the noise resilience of SSRP with a variant of SRP which is trained with A2C objective. The performance of the unsupervised ERT linker, provided in Table 4 reflects the level of noise that the model has to handle during training. Also, missing gold KB artifacts in the candidate lists render some of the test questions unanswerable. So we evaluate the competing models on two test-sets: i) **Full** 1K test set publicly available in CQA-12K dataset, and ii) the subset of the 1K test set that is **answerable** after the noisy query annotation. Performance on the answerable subset is a useful measure to understand the effects of noise. A noise-resilient model should retain good performance on the answerable set. Further, apart from training separate models for each question type, to understand the generalizability of the models, we also train a single model on data pooled from all categories.

6.3 Discussion

We make a few key observations on the results:

- On using gold input to the NPI, the significant margin of upto 2× improvement in F1 showcased by SRP over NSM indicate that the proposed method can indeed handle complex multi-step inferencing pragmatically and explore the search space more efficiently.
- The significant margin of performance between SSRP and SRP evinces that reference network for self-assessment in SSRP considerably helps in alleviating catastrophic forgetting. SSRP's performance is remarkable, as the average noise level is high, with only 10–20% of the train and test set being exactly answerable.
- Table 2 shows that *A2C* performs better than SSRP and SRP on the simpler classes (Simple and Logical). However, on the more complex category (Quantitative Count) or when generalizing to multiple categories with a single model, both SRP and SSRP beat A2C. This shows that the auxiliary rewards incorporated in SRP and SSRP facilitate better exploration in extremely sparse settings.
- From the training trend in Fig 3, it is evident that while SRP succumbs to unlearning, in the noisy setting, the training performance of SSRP steadily improves.
- Also, in contrast to NSM, both SSRP and SRP adapt fairly well to the exploded action space, because of the noisy input candidates, as can be seen in the scores obtained by the respective models. Only on the combined data, NSM performs somewhat better than SRP, presumably because it can afford to widen the exploration with double the beam-size owing to its model simplicity.
- While NSM decodes token-by-token requiring multi-step decoding for even simple queries, SRP and SSRP's atomic actions are each line-of-code. This altogether avoids generating logically inconsistent programs or incorporate high-level programmatic paradigms. Whereas, NSM only filters out "bad" programs post generatation, thus still wasting most explorations.

7 Conclusion

In this work, we proposed a noise resilient NPI model SSRP that tackles complex KBQA significantly better than state-of-the art models, with the two notable distinctions from other existing NPI models, i) it learns program induction in absence of supervision from gold programs, and ii) both during training and evaluation, it has to handle noise in the query annotation, so severe that it renders 80-90% of the questions unanswerable. In future, we plan to study more realistic applications like complex visual QA or algebraic problem solving.

References

[Bosnjak *et al.*, 2017] Matko Bosnjak, Tim Rocktäschel, Jason Naradowsky, and Sebastian Riedel. Programming with a differentiable forth interpreter. In *ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, pages 547–556, 2017.

[Bunel *et al.*, 2018] Rudy Bunel, Matthew J. Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. *CoRR*, abs/1805.04276, 2018.

[Dubey *et al.*, 2018] Mohnish Dubey, Debayan Banerjee, Debanjan Chaudhuri, and Jens Lehmann. EARL: joint entity and relation linking for question answering over knowledge graphs. *CoRR*, abs/1801.03825, 2018.

[He *et al.*, 2018] Xuanli He, Gholamreza Haffari, and Mohammad Norouzi. Sequence to sequence mixture model for diverse machine translation. In *Proceedings of the 22nd Conference on Computational Natural Language Learning, CoNLL 2018, Brussels, Belgium, October 31 - November 1, 2018*, pages 583–592, 2018.

[Iyyer *et al.*, 2017] Mohit Iyyer, Wen-tau Yih, and Ming-Wei Chang. Search-based neural structured learning for sequential question answering. In *ACL*, volume 1, 2017.

[Konda and Tsitsiklis, 2003] Vijay R. Konda and John N. Tsitsiklis. On actor-critic algorithms. *SIAM J. Control Optim.*, 42(4):1143–1166, April 2003.

[Liang *et al.*, 2017] Chen Liang, Jonathan Berant, Quoc Le, Kenneth D. Forbus, and Ni Lao. Neural symbolic machines: Learning semantic parsers on freebase with weak supervision. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, pages 23–33, 2017.

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

[Radhakrishnan *et al.*, 2018] Priya Radhakrishnan, Partha Talukdar, and Vasudeva Varma. Elden: Improved entity linking using densified knowledge graphs. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, volume 1, pages 1844–1853, 2018.

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

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

[Sorokin and Gurevych, 2017] Daniil Sorokin and Iryna Gurevych. Context-aware representations for knowledge base relation extraction. In *Proceedings of EMNLP*, 2017.

[Titov and Le, 2018] Ivan Titov and Phong Le. Improving entity linking by modeling latent relations between mentions. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers*, 2018.

[Wang *et al.*, 2018] William Yang Wang, Weiran Xu, and Pengda Qin. Robust distant supervision relation extraction via deep reinforcement learning. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers*, pages 2137–2147, 2018.

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

[Wu *et al.*, 2018] Zeqiu Wu, Xiang Ren, Frank F. Xu, Ji Li, and Jiawei Han. Indirect supervision for relation extraction using question-answer pairs. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining, WSDM 2018, Marina Del Rey, CA, USA, February 5-9, 2018*, pages 646–654, 2018.

[Xu *et al.*, 2016] Kun Xu, Siva Reddy, Yansong Feng, Songfang Huang, and Dongyan Zhao. Question answering on freebase via relation extraction and textual evidence. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*, 2016.

[Yih *et al.*, 2015] Wen-tau Yih, Ming-Wei Chang, Xiaodong He, and Jianfeng Gao. Semantic parsing via staged query graph generation: Question answering with knowledge base. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, ACL 2015, July 26-31, 2015, Beijing, China, Volume 1: Long Papers*, pages 1321–1331, 2015.

[Yih *et al.*, 2016] Wen-tau Yih, Matthew Richardson, Chris Meek, Ming-Wei Chang, and Jina Suh. 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, 2016.

[Yu *et al.*, 2017] Mo Yu, Wenpeng Yin, Kazi Saidul Hasan, Cícero Nogueira dos Santos, Bing Xiang, and Bowen Zhou. Improved neural relation detection for knowledge base question answering. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, 2017.