Neural Enquirer: Learning to Query Tables in Natural Language

Pengcheng Yin
Language Technologies Institute
pcyin@cs.cmu.edu

Zhengdong Lu
Noah's Ark Lab, Huawei Technologies
Lu.Zhengdong@huawei.com

Hang Li
Noah's Ark Lab, Huawei Technologies
HangLi.HL@huawei.com

Ben Kao
The University of Hong Kong
kao@cs.hku.hk

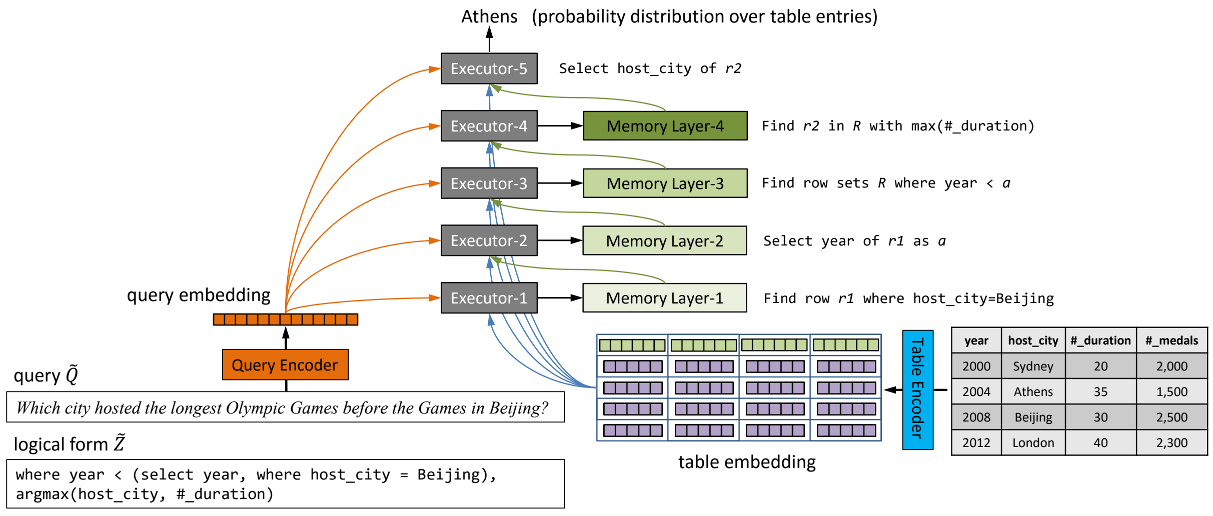
Abstract

We propose NEURAL ENQUIRER *— a neural network architecture for answering natural language (NL) questions based on a knowledge base (KB) table. Unlike existing work on end-to-end training of semantic parsers [13, 12],* NEURAL ENQUIRER *is fully "neuralized": it finds distributed representations of queries and KB tables, and executes queries through a series of neural network components called "executors". Executors model query operations and compute intermediate execution results in the form of table annotations at different levels.* NEURAL ENQUIRER *can be trained with gradient descent, with which the representations of queries and the KB table are jointly optimized with the query execution logic. The training can be done in an end-to-end fashion, and it can also be carried out with stronger guidance, e.g., step-by-step supervision for complex queries.* NEURAL ENQUIRER *is one step towards building neural network systems that can understand natural language in real-world tasks. As a proof-of-concept, we conduct experiments on a synthetic QA task, and demonstrate that the model can learn to execute reasonably complex NL queries on small-scale KB tables.*

1 Introduction

Natural language dialogue and question answering often involve querying a knowledge base [14, 3]. The traditional approach involves two steps: First, a given query $\tilde{Q}$ is semantically parsed into an "executable" representation, which is often expressed in certain logical form $\tilde{Z}$ (e.g., SQL-like queries). Second, the representation is executed against a knowledge base from which an answer is obtained. For queries that involve complex semantic constraints and logic (e.g., "*Which city hosted the longest Olympic Games before the Games in Beijing?*"), semantic parsing and query execution become extremely complex. For example, carefully hand-crafted features and rules are needed to correctly parse a complex query into its logical form (see example shown in the lower-left corner of Figure 1). This complexity often results in poor accuracy of the system. To partially overcome this difficulty, recent works [7, 10, 13] attempt to "backpropagate" query execution results to revise the semantic representation of a query, which is an example of learning from grounding [6, 9]. This approach, however, is greatly hindered by the fact that traditional semantic parsing mostly involves rule-based features and symbolic manipulation, and is subject to intractable search space incurred by the great flexibility of natural language.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering



Which city hosted the longest Olympic Games before the Games in Beijing?

```
where year < (select year, where host_city = Beijing),
argmax(host_city, #_duration)
```

year	host_city	#_duration	#_medals
2000	Sydney	20	2,000
2004	Athens	35	1,500
2008	Beijing	30	2,500
2012	London	40	2,300

Figure 1: An overview of NEURAL ENQUIRER with five executors

Neural network-based models have enjoyed much successes in natural language processing, particularly in machine translation and syntactic parsing. These successes are attributable to direct and strong supervision. The recent work on learning to execute simple program codes with LSTM [17] pioneers in the direction on learning to parse structured objects through executing it in a purely neural way, while the more recent work on Neural Turing Machines (NTMs) [8] introduces more modeling flexibility by equipping the LSTM with external memory and various means for interacting with it.

Inspired by the above-mentioned research, we aim to design a neural network system that learns to understand queries and execute them on a knowledge base table from examples of queries and answers. We propose NEURAL ENQUIRER, a fully neuralized, end-to-end differentiable system that jointly models semantic parsing and query execution. NEURAL ENQUIRER encodes queries and KB tables into distributed representations, and executes compositional queries against the KB through a series of differentiable executors. The model is trained using query-answer pairs, where the distributed representations of queries and the KB are optimized together with the query execution logic in an end-to-end fashion. As the first step along this line of research, we evaluate NEURAL ENQUIRER using a synthetic question-answering task as a proof-of-concept, and demonstrate that our proposed model is capable of learning to execute complex compositional natural language questions on small-scale KB tables.

2 Model

Following [13], we study the problem of question answering on a single KB table. Specifically, given an NL query Q and a KB table $\mathcal{T}$, NEURAL ENQUIRER executes Q against $\mathcal{T}$ and outputs a ranked list of answers. The execution is done by first using *Encoders* to encode the query and table into distributed representations, which are then sent to a cascaded pipeline of *Executors* to derive the answer. Figure 1 gives an illustrative example (with five executors). It consists of the following components:

Query Encoder (Section 2.1), which abstracts the semantics of an NL query and encodes it into a query embedding.

Table Encoder (Section 2.2), which derives a table embedding by encoding entries in the table into distributed vectors.

Executor (Section 2.3), which executes the query against the table and outputs *annotations* that encode intermediate execution results. Annotations are stored in the memory of each layer to be accessed by the executor of the next layer. Since complex compositional queries can be answered in multiple steps of computation, each executor models a specific type of operation conditioned on the query. Figure 1 illustrates the operation each executor is assumed to perform in answering the example query $\tilde{Q}$. Different from classical semantic parsing approaches which require a predefined set of all possible logical operations, NEURAL ENQUIRER *learns* the logic of executors via end-to-end training using query-answer pairs. By stacking several executors, our model is able to answer complex queries that involve multiple steps of computation.

2.1 Query Encoder

Query Encoder converts an NL query Q into a query embedding $\mathbf{q} \in \mathbb{R}^{d_Q}$. Let $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T\}$ be the embeddings of words in Q, where $\mathbf{x}_t \in \mathbb{R}^{d_W}$ is from an embedding matrix $\mathbf{L}$. We employ a bidirectional Gated Recurrent Unit (GRU) [2] to summarize the sequence $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T\}$ in forward and reverse orders. $\mathbf{q}$ is formed by concatenating the last hidden states in the two directions.

It is worth noting that Query Encoder can find the representation of a rather general class of symbol sequences, agnostic to the actual representation of the query (e.g., natural language, SQL, etc). The model is able to learn the semantics of input queries through end-to-end training, making it a generic model for query understanding and query execution.

2.2 Table Encoder

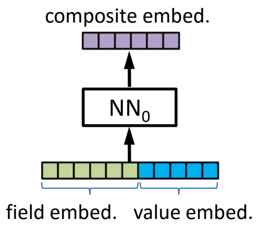


Table Encoder converts a KB table $\mathcal{T}$ into a distributed representation, which is used as an input to executors. Suppose $\mathcal{T}$ has M rows and N columns. In our model, the n-th column is associated with a *field name* (e.g., `host_city`). Each cell value is a word (e.g., *Beijing*) in the vocabulary. We use w_{mn} to denote the cell value in row m column n, and $\mathbf{w}_{mn}$ to denote its embedding. Let $\mathbf{f}_n$ be the embedding of the field name for column n. For each entry (cell) w_{mn}, Table Encoder computes a $\langle$field, value$\rangle$ composite embedding $\mathbf{e}_{mn} \in \mathbb{R}^{d_{\mathcal{E}}}$ by fusing $\mathbf{f}_n$ and $\mathbf{w}_{mn}$ using a single-layer Neural Network:

$$\mathbf{e}_{mn} = \mathsf{NN}_0(\mathbf{f}_n, \mathbf{w}_{mn}) = \tanh(\mathbf{W} \cdot [\mathbf{f}_n; \mathbf{w}_{mn}] + \mathbf{b}),$$

where $[\cdot;\cdot]$ denotes vector concatenation. The output of Table Encoder is an $M \times N \times d_{\mathcal{E}}$ tensor that consists of $M \times N$ embeddings, each of length $d_{\mathcal{E}}$.

We remark that our Table Encoder is different from classical knowledge embedding models (e.g., TransE [5]). While traditional methods learn the embeddings of entities (cell values) and relations (field names) in an unsupervised fashion via minimizing certain reconstruction errors, embeddings in Table Encoder are optimized via supervised learning in end-to-end QA tasks.

2.3 Executor

NEURAL ENQUIRER executes an input query on a KB table through layers of execution. Each layer consists of an executor that, after learning, performs certain operation (e.g., `select`, `max`) relevant to the input query. An executor outputs intermediate execution results, referred to as *annotations*, which are saved in the external memory of the executor. A query is executed sequentially through a stack of executors. Such a cascaded architecture enables the model to answer complex, compositional queries. An example is given in Figure 1 in which descriptions of the operation each executor is assumed to perform for the query $\tilde{Q}$ are shown. We will demonstrate in Section 4 that the model is capable of learning the operation logic of each executor via end-to-end training.

As illustrated in Figure 2, an executor at Layer-ℓ (denoted as $\mathsf{Executor}$-ℓ) consists of two major neural network components: a *Reader* and an *Annotator*. The executor processes a table row-by-row. The Reader reads

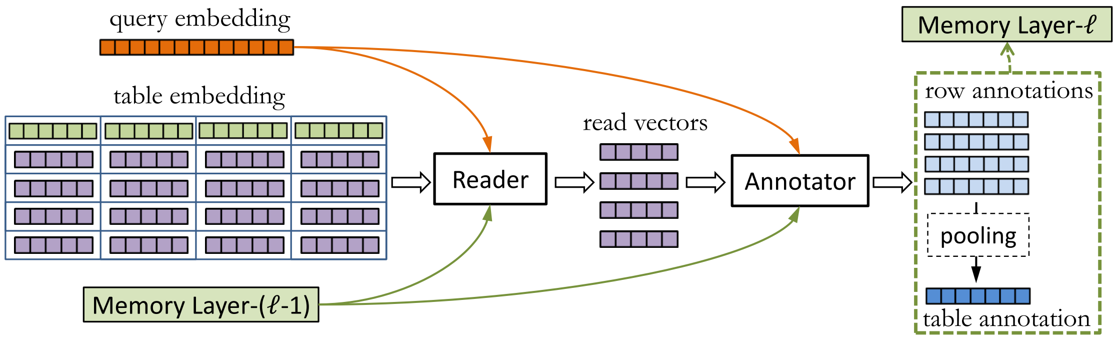


Figure 2: Overview of an Executor-ℓ

in data from each row m in the form of a *read vector* $\mathbf{r}_m^\ell$, which is then sent to the Annotator to perform the actual execution. The output of the Annotator is a *row annotation* $\mathbf{a}_m^\ell$, which captures the row-wise local computation result. Once all row annotations are obtained, Executor-ℓ generates a *table annotation* $\mathbf{g}^\ell$ to summarize the global computation result on the whole table by pooling all row annotations. All the row and table annotations are saved in the memory of Layer-ℓ: $\mathcal{M}^\ell = \{\mathbf{a}_1^\ell, \mathbf{a}_2^\ell, \ldots, \mathbf{a}_M^\ell, \mathbf{g}^\ell\}$. Intuitively, row annotations handle operations that require only row-wise, local information (e.g., `select`, `where`), while table annotations model superlative operations (e.g., `max`, `min`) by aggregating table-wise, global execution results. A combination of row and table annotations enables the model to perform a wide variety of query operations in real world scenarios.

2.3.1 Reader

As illustrated in Figure 3, for the m-th row with N $\langle$field, value$\rangle$ composite embeddings $\mathcal{R}_m = \{\mathbf{e}_{m1}, \mathbf{e}_{m2}, \ldots, \mathbf{e}_{mN}\}$, the Reader fetches a read vector $\mathbf{r}_m^\ell$ from $\mathcal{R}_m$ via an attentive reading operation:

$$\mathbf{r}_m^\ell = f_{\mathrm{R}}^\ell(\mathcal{R}_m, \mathcal{F}_\mathcal{T}, \mathbf{q}, \mathcal{M}^{\ell-1}) = \sum_{n=1}^{N} \tilde{\omega}(\mathbf{f}_n, \mathbf{q}, \mathbf{g}^{\ell-1})\mathbf{e}_{mn}$$

where $\mathcal{M}^{\ell-1}$ denotes the content of memory Layer-$(\ell-1)$, and $\mathcal{F}_\mathcal{T} = \{\mathbf{f}_1, \mathbf{f}_2, \ldots, \mathbf{f}_N\}$ is the set of field name embeddings. $\tilde{\omega}(\cdot)$ is the normalized attention weights given by:

$$\tilde{\omega}(\mathbf{f}_n, \mathbf{q}, \mathbf{g}^{\ell-1}) = \frac{\exp(\omega(\mathbf{f}_n, \mathbf{q}, \mathbf{g}^{\ell-1}))}{\sum_{n'=1}^{N} \exp(\omega(\mathbf{f}_{n'}, \mathbf{q}, \mathbf{g}^{\ell-1}))} \tag{10}$$

where $\omega(\cdot)$ is modeled as a Deep Neural Network (denoted as $\mathsf{DNN}_1^{(\ell)}$). Since each executor models a specific type of computation, it should only attend to a subset of entries that are pertinent to its execution. This is modeled by the Reader. Our approach is related to the content-based addressing of Neural Turing Machines [8] and the attention mechanism in neural machine translation models [2].

2.3.2 Annotator

The Annotator of Executor-ℓ computes row and table annotations based on read vectors fetched by the Reader. The results are stored in the ℓ-th memory layer $\mathcal{M}^\ell$ accessible to Executor-$(\ell$+$1)$. The last executor is the only exception, which outputs the final answer.

[Row annotations] Capturing row-wise execution result, the annotation $\mathbf{a}_m^\ell$ for row m in Executor-ℓ is given by

$$\mathbf{a}_m^\ell = f_{\mathrm{A}}^\ell(\mathbf{r}_m^\ell, \mathbf{q}, \mathcal{M}^{\ell-1}) = \mathsf{DNN}_2^{(\ell)}([\mathbf{r}_m^\ell; \mathbf{q}; \mathbf{a}_m^{\ell-1}; \mathbf{g}^{\ell-1}]). \tag{11}$$

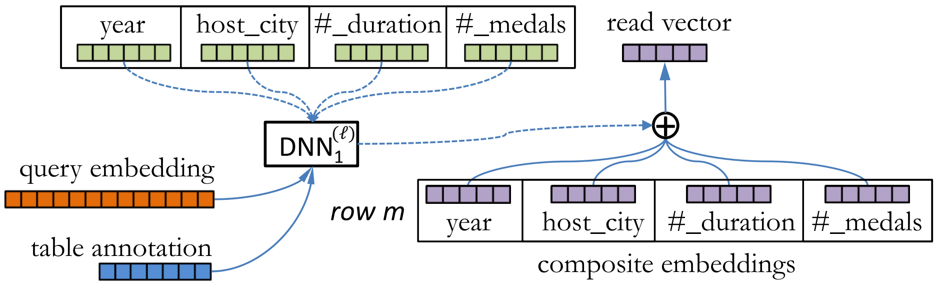


Figure 3: Illustration of the Reader in Executor-ℓ

$\mathsf{DNN}_2^{(\ell)}$ fuses the corresponding read vector $\mathbf{r}_m^\ell$, the results saved in the previous memory layer (row and table annotations $\mathbf{a}_m^{\ell-1}$, $\mathbf{g}^{\ell-1}$), and the query embedding $\mathbf{q}$. Intuitively, row annotation $\mathbf{a}_m^{\ell-1}$ and table annotation $\mathbf{g}^{\ell-1}$ summarize the local and global status of execution up to Layer-($\ell-1$), respectively. $\mathsf{DNN}_2^{(\ell)}$ then performs the actual query execution by combing these annotations with the read vector and query embedding, and outputs a row annotation $\mathbf{a}_m^\ell$ that encodes the local execution result on row m.
[Table annotations] Capturing global execution state, a table annotation summarizes all row annotations via a global max pooling operation:

$$\mathbf{g}^\ell = f_{\text{MAXPOOL}}(\mathbf{a}_1^\ell, \mathbf{a}_2^\ell, \ldots, \mathbf{a}_M^\ell) = [g_1, g_2, \ldots, g_{d_\mathcal{G}}]^\top \tag{12}$$

where $g_k = \max(\{\mathbf{a}_1^\ell(k), \mathbf{a}_2^\ell(k), \ldots, \mathbf{a}_M^\ell(k)\})$ is the maximum value among the k-th elements of all row annotations.

2.3.3 Last Layer Executor

Instead of computing annotations based on read vectors, the last executor in NEURAL ENQUIRER directly outputs the probability of an entry w_{mn} in table $\mathcal{T}$ being the answer a:

$$p(a = w_{mn}|Q, \mathcal{T}) = \frac{\exp(f_{\text{ANS}}^\ell(\mathbf{e}_{mn}, \mathbf{q}, \mathbf{a}_m^{\ell-1}, \mathbf{g}^{\ell-1}))}{\sum_{m'=1,n'=1}^{M,N} \exp(f_{\text{ANS}}^\ell(\mathbf{e}_{m'n'}, \mathbf{q}, \mathbf{a}_{m'}^{\ell-1}, \mathbf{g}^{\ell-1}))} \tag{13}$$

where $f_{\text{ANS}}^\ell(\cdot)$ is modeled as a DNN ($\mathsf{DNN}_3^{(\ell)}$). Note that the last executor, which is devoted to returning answers, could still carry out execution in $\mathsf{DNN}_3^{(\ell)}$.

3 Learning

NEURAL ENQUIRER can be trained in an *end-to-end* (N2N) fashion on QA tasks. During training, both the representations of queries and tables, as well as the execution logic captured by the weights of executors are learned. Given a set of $N_\mathcal{D}$ query-table-answer triples $\mathcal{D} = \{(Q^{(i)}, \mathcal{T}^{(i)}, y^{(i)})\}$, we learn the model parameters by maximizing the log-likelihood of gold-standard answers:

$$\mathcal{L}_{\text{N2N}}(\mathcal{D}) = \sum_{i=1}^{N_\mathcal{D}} \log p(a = y^{(i)}|Q^{(i)}, \mathcal{T}^{(i)}) \tag{14}$$

In end-to-end training, each executor discovers its operation logic from training data in a purely data-driven fashion, which could be difficult for complex queries requiring four or five sequential operations.

year	host_city	#_participants	#_medals	#_duration	#_audience	host_country	GDP	country_size	population
2008	Beijing	4,200	2,500	30	67,000	China	2,300	960	130

Figure 4: An example table in the synthetic QA task (only one row shown)

Query Type	Example Queries (Q) with Annotated SQL-like Logical Forms (Z)
SELECT_WHERE	▷ Q: *How many people participated in the game in Beijing?* Z: `select #_participants, where host_city = Beijing` ▷ Q: *In which country was the game hosted in 2012?* Z: `select host_country, where year = 2012`
SUPERLATIVE	▷ Q: *When was the lastest game hosted?* Z: `argmax(host_city, year)` ▷ Q: *How big is the country which hosted the shortest game?* Z: `argmin(country_size, #_duration)`
WHERE_SUPERLATIVE	▷ Q: *How long is the game with the most medals that has fewer than 3,000 participants?* Z: `where #_participants < 3,000, argmax(#_duration, #_medals)` ▷ Q: *How many medals are in the first game after 2008?* Z: `where #_year > 2008, argmin(#_medals, #_year)`
NEST	▷ Q: *Which country hosted the longest game before the game in Athens?* Z: `where year<(select year,where host_city=Athens),argmax(host_country,#_duration)` ▷ Q: *How many people watched the earliest game that lasts for more days than the game in 1956?* Z: `where #_duration<(select #_duration,where year=1956),argmin(#_audience,#_year)`

Table 1: Example queries in our synthetic QA task

This can be alleviated by softly guiding the learning process via controlling the attention weights $\tilde{w}(\cdot)$ in Eq. (10). By enforcing $\tilde{w}(\cdot)$ to bias towards a field pertaining to a specific operation, we can "coerce" the executor to figure out the logic of this operation relative to the field. For example, for Executor-1 in Figure 1, by biasing the attention weight of the `host_city` field towards 1.0, only the value of `host_city` will be fetched and sent to the Annotator. In this way we can "force" the executor to learn the `where` operation to find the row whose `host_city` is *Beijing*. This method will be referred to as *step-by-step* (SbS) training. Formally, this is done by introducing additional supervision signal to Eq. (14):

$$\mathcal{L}_{\text{SbS}}(\mathcal{D}) = \sum_{i=1}^{N_{\mathcal{D}}} \Big(\log p(a = y^{(i)} | Q^{(i)}, \mathcal{T}^{(i)}) + \alpha \sum_{\ell=1}^{L-1} \log \tilde{w}(\mathbf{f}^{\star}_{i,\ell}, \cdot, \cdot) \Big) \tag{15}$$

where α is a tuning weight, and L is the number of executors. $\mathbf{f}^{\star}_{i,\ell}$ is the embedding of the field known *a priori* to be used by Executor-ℓ in answering the i-th example.

4 Experiments

In this section we evaluate NEURAL ENQUIRER on synthetic QA tasks with NL queries of varying compositional depths.

4.1 Synthetic QA Task

We present a synthetic QA task with a large number of QA examples at various levels of complexity to evaluate the performance of NEURAL ENQUIRER. Starting with "artificial" tasks accelerates the development of novel deep models [15], and has gained increasing popularity in recent research on modeling symbolic computation using DNNs [8, 17].

Our synthetic dataset consists of query-table-answer triples $\{(Q^{(i)}, \mathcal{T}^{(i)}, y^{(i)})\}$. To generate a triple, we first randomly sample a table $\mathcal{T}^{(i)}$ of size 10×10 from a synthetic schema of Olympic Games. The cell values of $\mathcal{T}^{(i)}$ are drawn from a vocabulary of 120 location names and 120 numbers. Figure 4 gives an example table. Next, we

	MIXED-25K				MIXED-100K		
	SEMPRE	N2N	SbS	N2N-OOV	N2N	SbS	N2N-OOV
SELECT_WHERE	93.8%	96.2%	99.7%	90.3%	99.3%	100.0%	97.6%
SUPERLATIVE	97.8%	98.9%	99.5%	98.2%	99.9%	100.0%	99.7%
WHERE_SUPERLATIVE	34.8%	80.4%	94.3%	79.1%	98.5%	99.8%	98.0%
NEST	34.4%	60.5%	92.1%	57.7%	64.7%	99.7%	63.9%
Overall Accuracy	65.2%	84.0%	96.4%	81.3%	90.6%	99.9%	89.8%

Table 2: Accuracies on MIXED datasets

sample a query $Q^{(i)}$ generated using NL templates, and obtain its gold-standard answer $y^{(i)}$ on $\mathcal{T}^{(i)}$. Our task consists of four types of NL queries, with examples given in Table 1. We also give the logical form template for each type of query. The templates define the semantics and compositionality of queries. We generate queries at various compositional depths, ranging from simple SELECT_WHERE queries to more complex NEST ones. This makes the dataset have similar complexity as a real-world one, except for the relatively small vocabulary. The queries are flexible enough to involve complex matching between NL phrases and logical constituents, which makes query understanding nontrivial: (1) the same field is described by different NL phrases (e.g., *"How big is the country ..."* and *"What is the size of the country ..."* for the `country_size` field); (2) different fields may be referred to by the same NL pattern (e.g, *"$\underline{\text{in}}$ China"* for `host_country` and *"$\underline{\text{in}}$ Beijing"* for `host_city`); (3) simple NL constituents may be grounded to complex logical operations (e.g., *"$\underline{\text{after}}$ the Games in Beijing"* implies comparing between the values of `year` fields).

To simulate the read-world scenario where queries of various types are issued to the model, we construct two MIXED datasets, with $25K$ and $100K$ training examples respectively, where four types of queries are sampled with the ratio $1:1:1:2$. Both datasets share the same testing set of $20K$ examples, $5K$ for each type of query. We enforce that no tables and queries are shared between training/testing sets.

4.2 Setup

[Tuning] We adopt a model with five executors. The lengths of hidden states for GRU and DNNs are 150, 50. The numbers of layers for $\text{DNN}_1^{(\ell)}$, $\text{DNN}_2^{(\ell)}$ and $\text{DNN}_3^{(\ell)}$ are 2, 3, 3. The length of word embeddings and annotations is 20. α is 0.2. We train the model using ADADELTA [18] on a Tesla K40 GPU. The training converges fast within 2 hours.
[Metric] We evaluate in terms of accuracy, defined as the fraction of correctly answered queries.
[Models] We compare the results of the following settings:

- **Sempre** [13] is a state-of-the-art semantic parser and serves as the baseline;
- **N2N**, our model trained using *end-to-end* setting (Sec 4.3);
- **SbS**, our model trained using *step-by-step* setting (Sec 4.4);
- **N2N-OOV**, a variant of the N2N model to deal with out-of-vocabulary words (Sec 4.5)

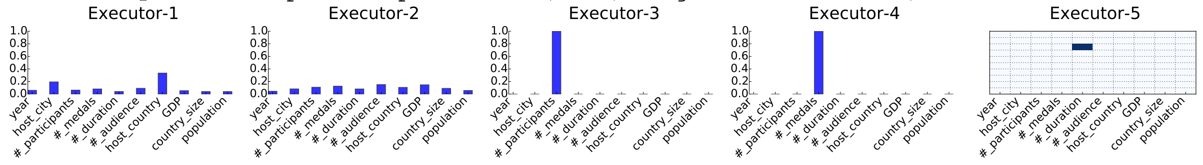
4.3 End-to-End Evaluation

Table 2 summarizes the results of SEMPRE and NEURAL ENQUIRER under different settings. We show both the individual performance for each query type and the overall accuracy. We evaluate SEMPRE only on MIXED-25K because of its long training time even on this small dataset (about 3 days).

In this section we discuss the results under end-to-end (N2N) training setting. On MIXED-25K, the relatively low performance of SEMPRE indicates that our QA task, although synthetic, is highly nontrivial. Surprisingly, NEURAL ENQUIRER outperforms SEMPRE on all query types, with a marginal gain on *simple* queries

Q_1: *How long was the Games with the most medals that had fewer than 3,000 participants?*

Z_1: `where #_participants < 3,000, argmax(#_duration, #_medals)`



Q_2: *Which country hosted the longest Games before the Games in Athens?*

Z_2: `where year < (select year, where host_city = Athens), argmax(host_country, #_duration)`

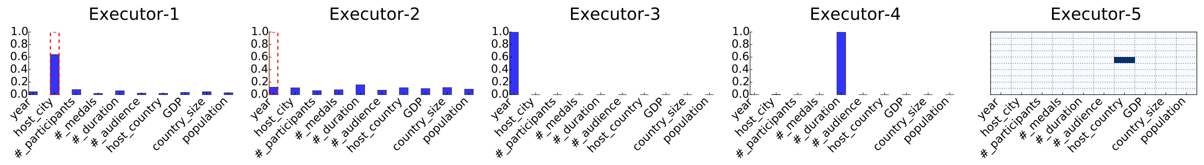


Figure 5: Weights visualization of queries Q_1 and Q_2

(SELECT_WHERE, SUPERLATIVE), and significant improvement on *complex* queries (WHERE_SUPERLATIVE, NEST). We posit that the low performance of SEMPRE on complex queries is likely due to the intractable search space incurred by the flexibility of its float parsing algorithm. On MIXED-100K, our model registers an overall accuracy of 90.6%. These results show that in our QA task, NEURAL ENQUIRER is very effective in answering compositional NL queries, especially those with complex semantics compared with the state-of-the-art system.

To further understand why our model is capable of answering compositional queries, we study the attention weights of Readers (Eq. 10) for intermediate executors, and the answer probability (Eq. 13) the last executor outputs for each table entry. These statistics are obtained on MIXED-100K. We sample two queries (Q_1 and Q_2) in the testing set that our model answers correctly and visualize their corresponding values in Figure 5. To better understand the query execution process, we also give the logical forms (Z_1 and Z_2) of the two queries. Note that the logical forms are just for reference purpose and unknown by the model. We find that each executor actually *learns* its execution logic in N2N training, which is in accordance with our assumption. The model executes Q_1 in three steps, with each of the last three executors performs a specific type of operation. For each row, Executor-3 takes the value of the `#_participants` field as input, while Executor-4 attends to the `#_medals` field. Finally, Executor-5 outputs a high probability for the `#_duration` field in the 3-rd row. The attention weights for Executor-1 and Executor-2 appear to be meaningless because Q_1 requires only three steps of execution, and the model learns to defer the meaningful execution to the last three executors. Comparing with the logical form Z_1 of Q_1, we can deduce that Executor-3 "executes" the `where` clause in Z_1 to find row sets R satisfying the condition, and Executor-4 performs the first part of `argmax` to find the row $r \in R$ with the maximum value of `#_medals`, while Executor-5 outputs the value of `#_duration` in r.

Compared with the relatively simple Q_1, Q_2 is more complicated. According to Z_2, Q_2 involves an additional nest sub-query to be solved by two extra executors, and requires a total of five steps of execution. The last three executors function similarly as in answering Q_1, yet the execution logic for the first two executors (devoted to solving the sub-query) is a bit obscure, since their attention weights are scattered instead of being perfectly centered on the ideal fields as highlighted in red dashed rectangles. We posit that this is because during the end-to-end training, the supervision signal propagated from the top layer has decayed along the long path down to the first two executors, which causes vanishing gradients.

4.4 With Additional Step-by-Step Supervision

To alleviate the vanishing gradient problem when training on complex queries like Q_2, we train the model using step-by-step (SbS) setting (Eq. 15), where we encourage each intermediate executor to attend to the field that is known *a priori* to be relevant to its execution logic. Results are shown in Table 2 (column SbS). With stronger supervision signal, the model significantly outperforms the N2N setting, and achieves perfect accuracy on MIXED-100K. This shows that NEURAL ENQUIRER is capable of leveraging the additional supervision signal given to intermediate layers in SbS training. Let us revisit the query Q_2 in SbS setting. In contrast to the result in N2N setting (Figure 5) where the attention weights for the first two executors are obscure, now the weights are perfectly skewed towards each relevant field with a value of 1.0, which corresponds with the highlighted ideal weights.

4.5 Dealing with Out-Of-Vocabulary Words

One of the major challenges for applying neural network models to NLP applications is to deal with out-of-vocabulary (OOV) words (e.g., new entities for QA). Surprisingly, we find that a simple variant of NEURAL ENQUIRER is able to handle unseen entities almost without loss of accuracy.

Specifically, we divide words in the vocabulary into *entity* words and *operation* words. Embeddings of entity words (e.g., *Beijing*) function like indices to facilitate the matching between the entities in queries and tables during query execution, and therefore are not updated once randomly initialized; while those of operation words, i.e., all non-entity words (e.g., numbers, *longest*, *before*, etc), carry semantic meanings relevant to execution and will be optimized in training. Therefore, after randomly initializing the embedding matrix $\mathbf{L}$, we only update the embeddings of operation words in training, while keeping those of entity words unchanged. To evaluate the model we modify the queries in the testing set to replace all entity words (i.e., all country and city names) with those unseen in training. Results obtained using N2N training, reported in Table 2 (column N2N-OOV), show that the model yields performance comparable with non-OOV settings.

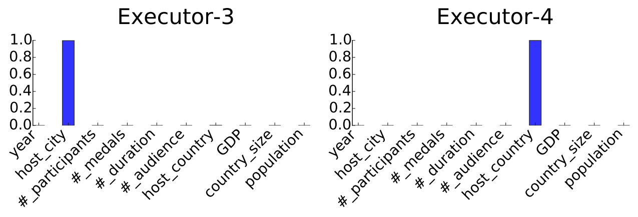


Figure 6: Weights visualization of query Q_3

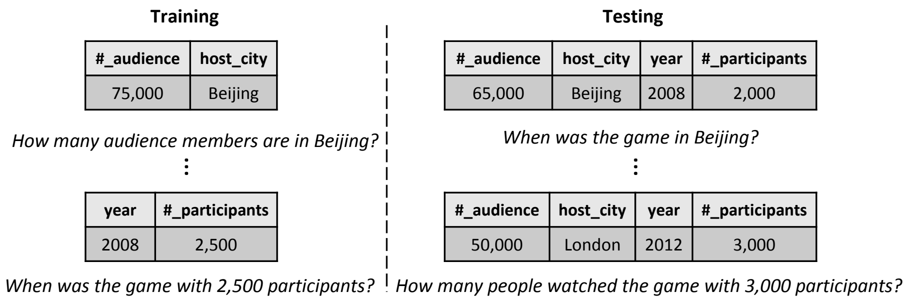
An interesting question is how the model resolves the types of OOV entity words (i.e., cities *vs.* countries) in ambiguous queries, e.g., Q_3: *"How many people watched the Games in <u>Macau</u>?"*, since the random embeddings of entity words (e.g, *Macau*) cannot link them to their corresponding fields. The model executes Q_3 using the last three executors, with the last executor attending to the `#_audience` field as expected. Interestingly, however, the model attends to the `host_city` field in Executor-3, and then `host_country` in Executor-4 (see Figure 6), indicating the model learns to scan all possible fields to figure out the correct field of an OOV entity.

4.6 Querying Expanded Knowledge Source

We simulate a test case to evaluate the model's ability to generalize to an expanded knowledge source. We train a model on tables whose field sets are either $\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_5$, where $\mathcal{F}_i$ is a subset of the entire field set $\mathcal{F}_\mathcal{T}$ and $|\mathcal{F}_i| = 5$. We then test the model on tables with all fields $\mathcal{F}_\mathcal{T}$ and queries whose fields span multiple

Query Type	SELECT_WHERE	SUPERLATIVE	WHERE_SUPERLATIVE	Overall
Accuracy	68.2%	84.8%	80.2%	77.7%

Table 3: Accuracies for querying expanded knowledge source


Training
Testing
How many audience members are in Beijing?
When was the game in Beijing?
When was the game with 2,500 participants?
How many people watched the game with 3,000 participants?

#_audience	host_city
75,000	Beijing

year	#_participants
2008	2,500

#_audience	host_city	year	#_participants
65,000	Beijing	2008	2,000

#_audience	host_city	year	#_participants
50,000	London	2012	3,000

Figure 7: Expanded knowledge source querying simulation

subsets $\mathcal{F}_i$. Figure 7 illustrates the setting. All test queries exhibit field combinations unseen in training. This simulates the difficulty the model often encounters when scaling to large knowledge sources, which usually poses a great challenge on model's generalization ability. We evaluate the N2N model on a dataset of the first three types of relatively simple queries. The sizes of training/testing splits are 75,000 and 30,000, with equal numbers for each query type. Table 3 lists the results. The model maintains a reasonable performance even when the compositionality of test queries is previously unseen, showing the model's generalization ability in tackling unseen query patterns through the composition of familiar ones, and hence the potential to scale to larger and unseen knowledge sources.

5 Related Work

This work is related to semantic parsing, which aims to parse NL queries into logical forms executable on KBs [19, 1]. Recent studies take a semi-supervised learning approach, and adopt the results of query execution as indirect supervision to train a parser [3, 4, 16, 13, 11]. Semantic parsers learned in this way can scale to large open domain KBs, but are inadequate for understanding complex queries because of the intractable search space incurred by the flexibility of parsing algorithms. Our work follows this approach in using query answers as indirect supervision, but jointly performs semantic parsing and query execution in distributional spaces, where the distributed representations of logical forms are implicitly learned in end-to-end QA tasks.

Our work is also related to the recent research of modeling symbolic computation using neural networks, pioneered by the development of Neural Turing Machines (NTMs) [8] and the work of learning to execute (LTE) simple Python programs using LSTM [17]. It is related to both lines of research in using external memories like NTMs and learning by executing like LTE. As a highlight and difference, our work employs multiple layers of deep memories, with the neural network operations highly customized towards querying KB tables.

Perhaps the most related work is the recently proposed NEURAL PROGRAMMER [12], which studies the same task of executing queries on tables using DNNs. While in NEURAL PROGRAMMER, the query planning is modeled using DNNs to determine which operation to execute at each step, the symbolic operations are predefined by users. In contrast our model is fully neuralized: it models both the query planning and query execution using DNNs, which are jointly optimized via end-to-end training. Our model learns symbolic operations using a data-driven approach. We also present results on NL queries and demonstrate that a fully neural system is capable of executing compositional logic operations up to a certain level of complexity.

6 Conclusion

We propose NEURAL ENQUIRER, a fully neural, end-to-end differentiable network that learns to execute compositional natural language queries on knowledge base tables. We present results on a set of synthetic QA tasks to demonstrate the ability of NEURAL ENQUIRER to answer fairly complicated compositional queries across multiple tables. In the future we plan to advance this work in the following directions. First we will apply NEURAL ENQUIRER to natural language questions and natural language answers, where both the input query and the output supervision are noisier and less informative. Second, we are going to scale to real world QA task as in [13], for which we have to deal with a large vocabulary and novel predicates. Third, we are going to work on the computational efficiency issue in query execution by heavily borrowing the symbolic operation.

References

[1] Y. Artzi, K. Lee, and L. Zettlemoyer. Broad-coverage CCG semantic parsing with AMR. In *EMNLP*, 1699–1710, 2015.

[2] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. In *ICLR*, 2015.

[3] J. Berant, A. Chou, R. Frostig, and P. Liang. Semantic parsing on freebase from question-answer pairs. In *EMNLP*, 1533–1544, 2013.

[4] J. Berant and P. Liang. Semantic parsing via paraphrasing. In *ACL (1)*, 1415–1425, 2014.

[5] A. Bordes, N. Usunier, A. Garca-Durn, J. Weston, and O. Yakhnenko. Translating embeddings for modeling multi-relational data. In *NIPS*, 2787–2795, 2013.

[6] D. L. Chen and R. J. Mooney. Learning to sportscast: A test of grounded language acquisition. In *ICML*, 128–135, 2008.

[7] J. Clarke, D. Goldwasser, M.-W. Chang, and D. Roth. Driving semantic parsing from the world's response. In *CoNLL*, 18–27, 2010.

[8] A. Graves, G. Wayne, and I. Danihelka. Neural turing machines. *CoRR*, abs/1410.5401, 2014.

[9] J. Kim and R. J. Mooney. Unsupervised pcfg induction for grounded language learning with highly ambiguous supervision. In *EMNLP-CoNLL*, 433–444, 2012.

[10] P. Liang, M. I. Jordan, and D. Klein. Learning dependency-based compositional semantics. In *ACL (1)*, 590–599, 2011.

[11] D. K. Misra, K. Tao, P. Liang, and A. Saxena. Environment-driven lexicon induction for high-level instructions. In *ACL (1)*, 992–1002, 2015.

[12] A. Neelakantan, Q. V. Le, and I. Sutskever. Neural programmer: Inducing latent programs with gradient descent. *CoRR*, abs/1511.04834, 2015.

[13] P. Pasupat and P. Liang. Compositional semantic parsing on semi-structured tables. In *ACL (1)*, 1470–1480, 2015.

[14] T.-H. Wen, M. Gasic, N. Mrksic, P. hao Su, D. Vandyke, and S. J. Young. Semantically conditioned lstm-based natural language generation for spoken dialogue systems. In *EMNLP*, 1711–1721, 2015.

[15] J. Weston, A. Bordes, S. Chopra, and T. Mikolov. Towards ai-complete question answering: A set of prerequisite toy tasks. *CoRR*, abs/1502.05698, 2015.

[16] W. Yih, M. Chang, X. He, and J. Gao. Semantic parsing via staged query graph generation: Question answering with knowledge base. In *ACL (1)*, 1321–1331, 2015.

[17] W. Zaremba and I. Sutskever. Learning to execute. *CoRR*, abs/1410.4615, 2014.

[18] M. D. Zeiler. ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701, 2012.

[19] L. S. Zettlemoyer and M. Collins. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *UAI*, 658–666, 2005.