Neural Enquirer: Learning to Query Tables

Pengcheng Yin[†*] Zhengdong Lu[‡] Hang Li[‡] Ben Kao[†]
[†]Dept. of Computer Science
The University of Hong Kong
{pcyin, kao}@cs.hku.hk
[‡]Noah's Ark Lab, Huawei Technologies
{Lu.Zhengdong, HangLi.HL}@huawei.com

Abstract

We proposed Neural Enquirer as a neural network architecture to execute a SQL-like query on a knowledge-base (KB) for answers. Basically, Neural Enquirer finds the distributed representation of a query and then executes it on knowledge-base tables to obtain the answer as one of the values in the tables. Unlike similar efforts in end-to-end training of semantic parser [11, 9], Neural Enquirer is fully "neuralized": it not only gives distributional representation of the query and the knowledge-base, but also realizes the execution of compositional queries as a series of differentiable operations, with intermediate results (consisting of annotations of the tables at different levels) saved on multiple layers of memory. Neural Enquirer can be trained with gradient descent, with which not only the parameters of the controlling components and semantic parsing component, but also the embeddings of the tables and query words can be learned from scratch. The training can be done in an end-to-end fashion, but it can take stronger guidance, e.g., the step-by-step supervision for complicated queries, and benefit from it. Neural Enquirer is one step towards building neural network systems which seek to understand language by executing it on real-world. Our experiments show that Neural Enquirer can learn to execute fairly complicated queries on tables with rich structures.

1 Introduction

In models for natural language dialogue and question answering, there is ubiquitous need for querying a knowledge-base [13, 11]. The traditional pipeline is to put the query through a semantic parser to obtain some "executable" representations, typically logic forms, and then apply this representation to a knowledge-base for the answer. Both the semantic parsing and the query execution part can get quite messy for complicated queries like "`what city hosts the Olympic Game after Beijing?`", and need carefully devised systems with hand-crafted features or rules. Partially to overcome this difficulty, there has been effort [11] to "backpropagate" the result of query execution to revise the semantic representation of the query, which actually falls into the thread of work on learning from grounding [5]. One drawback of these semantic parsing models is rather symbolic with rule-based features, leaving only a handful of tunable parameters to cater to the supervision signal from the execution result.

On the other hand, neural network-based models are previously successful mostly on tasks with direct and strong supervision in natural language processing or related domain, with examples including machine translation and syntactic parsing. The recent work

[*]Work done when the first author worked as an intern at Noah's Ark Lab, Huawei Technologies.

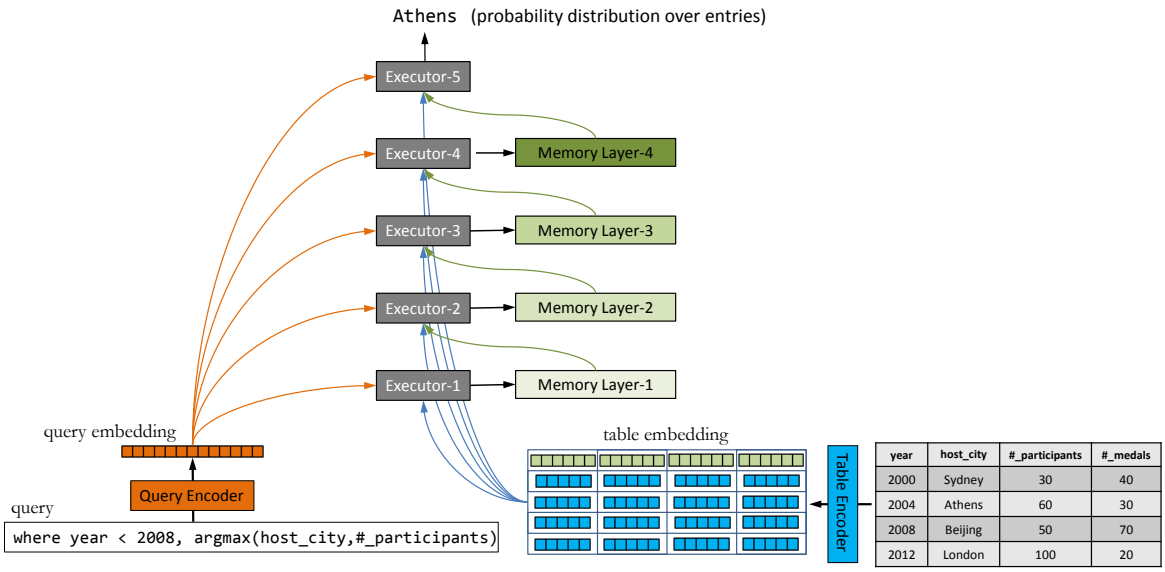


Figure 1: An overview of NEURAL ENQUIRER with five executors

on learning to execute simple Python code with LSTM [15] pioneers in the direction on learning to parse structured objects through executing it in a purely neural way, while the later work on Nerual Turing Machine (NTM) [6] introduces more modeling flexibility by equipping the LSTM with external memory and various means of interacting with it.

Our work, inspired by above-mentioned threads of research, aims to design a neural network system that can learn to understand the query and execute it on a knowledge-base from examples of queries and answers. As the first step, we limit ourselves to SQL-like queries as a canonical and informative form. In other words, for query "`what city hosts the Olympic Game after Beijing?`", we instead use "`where host_city = Beijing, select year as A, where year > A, argmin(host_city, year)`". However, since NEURAL ENQUIRER parses, represents, and executes queries with purely neural network-based models, it is insensitive to the actual format of the query, and can be extended to natural language queries or query representations given by other models with learning ability.

2 Overview of Neural Enquirer

Given a query Q and a KB table $\mathcal{T}$, NEURAL ENQUIRER executes the query against the table and outputs a ranked list of query 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) of various types of components involved:

Query Encoder (Section 3.1), which encodes the query into a distributed representation that carries the semantic information of the original query. The encoded query embedding will be sent to various executors to compute its execution result.

Table Encoder (Section 3.2), which encodes entries in the table into distributed vectors. Table Encoder outputs an embedding vector for each table entry, which retains the two-dimensional structure of the table.

Executor (Section 3.3), which executes the query against the table and outputs *anno-*

tations as intermediate execution results. Annotations are stored in the memory of each layer to facilitate retrieval by subsequent layer of executor. Instead of giving annotations, the last executor answers the query by computing the probability of each table entry being the query answer. Our basic assumption is that complex, compositional queries can be answered through multiple layers of computation, and each executor models an intermediate computational step. By stacking executors, NEURAL ENQUIRER is capable of answering complex queries involving multi-step computations.

3 Model

In this section we give a more detailed exposition of different types of components in the NEURAL ENQUIRER model.

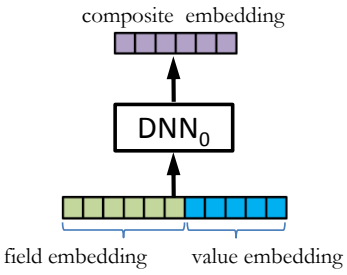
3.1 Query Encoder

Given a query Q composed of a sequence of words $\{w_1, w_2, \ldots, w_T\}$, Query Encoder parses Q into a $d_{\mathcal{Q}}$-dimensional vectorial representation $\mathbf{q}$: $Q \xrightarrow{\texttt{encode}} \mathbf{q} \in \mathbb{R}^{d_{\mathcal{Q}}}$. In our implementation of NEURAL ENQUIRER, we employ a Gated Recurrent Network (GRU) for this mission[1]. More specifically, a GRU takes as input the sequence of embeddings of words, $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T\}$, where $\mathbf{x}_t = \mathbf{L}[w_t]$, $\mathbf{x}_t \in \mathbb{R}^{d_{\mathcal{W}}}$ and $\mathbf{L}$ is the embedding matrix. We use the last hidden state of the GRU as the vectorial representation of the query. See Appendix A for details.

It is worth noting that our Query Encoder can find the representation of rather general class of symbol sequences, agnostic to the actual representation of queries (e.g., Natural Language, SQL-like, etc). NEURAL ENQUIRER is capable of learning the execution logic expressed in the input query through end-to-end training, making it a generic model for query execution.

3.2 Table Encoder

Table Encoder converts a knowledge-base table $\mathcal{T}$ into its distributional representation as input to NEURAL ENQUIRER. Suppose the table has M rows and N columns, where each column comes with a *field name* (e.g., `host_city`), and the value of each table entry is a word (e.g., `Beijing`) in our vocabulary, Table Encoder first finds the embedding for field names and values of table, and then it computes the (field, value) composite embedding for each of the $M \times N$ entries in the table. More specifically, for the entry in the m-th row and n-th column with a value of w_{mn}, Table Encoder computes a $d_{\mathcal{E}}$-dimensional embedding vector $\mathbf{e}_{mn}$ by fusing the embedding of the entry value with the embedding of its corresponding field name as follows:



$$\mathbf{e}_{mn} = \mathsf{DNN}_0([\mathbf{L}[w_{mn}]; \mathbf{f}_n]) = \tanh(\mathbf{W} \cdot [\mathbf{L}[w_{mn}]; \mathbf{f}_n] + \mathbf{b})$$

where $\mathbf{f}_n$ is the embedding of the field name (of the n-th column). $\mathbf{W}$ and $\mathbf{b}$ denote the weight matrices, and $[\cdot;\cdot]$ the concatenation of vectors. The output of Table Encoder is a tensor of shape $M \times N \times d_{\mathcal{E}}$, consisting of $M \times N$ embeddings of length $d_{\mathcal{E}}$ for all entries.

[1]Other choices of sentence encoder such as LSTM or even convolutional neural networks are possible too

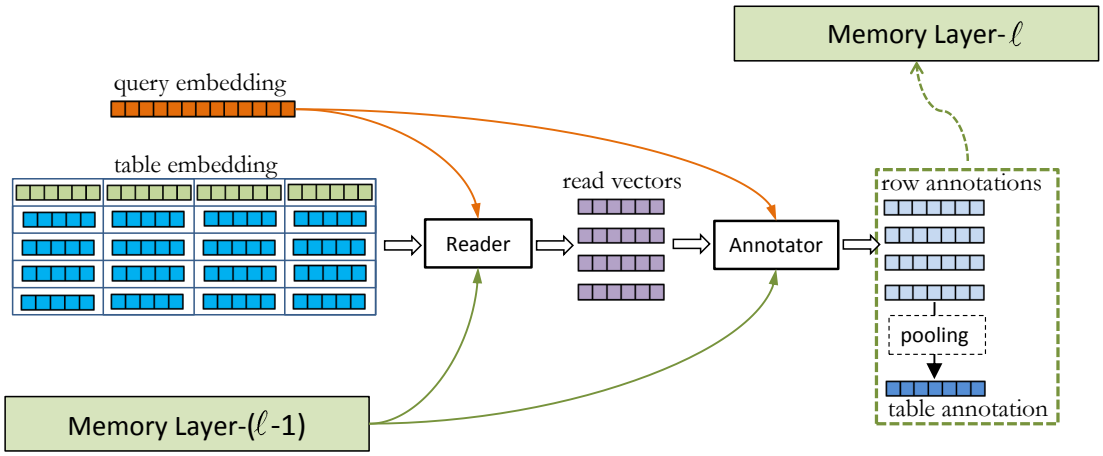


Figure 2: Overview of an Executor-ℓ

Our Table Encoder functions differently from classical knowledge embedding models (e.g., TransE [4]), where embeddings of entities (entry values) and relations (field names) are learned in a unsupervised fashion via minimizing certain reconstruction errors. Embeddings in NEURAL ENQUIRER are optimized via supervised learning towards end-to-end QA tasks. Additionally, as will shown in the experiments, those embeddings function in a way as indices, which not necessarily encode the exact semantic meaning of their corresponding words.

3.3 Executor

NEURAL ENQUIRER executes an input query on a KB table through layers of execution. Each layer of executor captures a certain type of operation (e.g., `select`, `where`, `max`, etc.) and returns some intermediate results, referred to as *annotations*, saved in an external memory of the same layer. A query is executed step-by-step through a sequence of stacked executors. Such a cascaded architecture enables NEURAL ENQUIRER to answer complex, compositional queries.

As illustrated in Figure 2, an executor at Layer-ℓ (denoted as Executor-ℓ) has two major neural network components: Reader and Annotator. The executor processes a table row-by-row. For the m-th row, with N (field, value) 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 $\mathcal{R}_m$, which is sent to the Annotator to compute a *row annotation* $\mathbf{a}_m^\ell \in \mathbb{R}^{d_\mathcal{A}}$:

$$\text{Read Vector:} \quad \mathbf{r}_m^\ell = f_{\text{R}}^\ell(\mathcal{R}_m, \mathcal{F}_\mathcal{T}, \mathbf{q}, \mathcal{M}^{\ell-1}) \tag{1}$$

$$\text{Row Annotation:} \quad \mathbf{a}_m^\ell = f_{\text{A}}^\ell(\mathbf{r}_m^\ell, \mathbf{q}, \mathcal{M}^{\ell-1}) \tag{2}$$

where $\mathcal{M}^{\ell-1}$ denotes the content in 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. Once all row annotations are obtained, Executor-ℓ then generates the *table annotation* through the following pooling process:

$$\text{Table Annotation:} \quad \mathbf{g}^\ell = f_{\text{POOL}}(\mathbf{a}_1^\ell, \mathbf{a}_2^\ell, \ldots, \mathbf{a}_M^\ell).$$

A row annotation captures the local execution result on each row, while a table annotation, derived from all row annotations, summarizes the global computational result on the whole table. Both row annotations $\{\mathbf{a}_1^\ell, \mathbf{a}_2^\ell, \ldots, \mathbf{a}_M^\ell\}$ and table annotation $\mathbf{g}^\ell$ are saved in memory Layer-ℓ: $\mathcal{M}^\ell = \{\mathbf{a}_1^\ell, \mathbf{a}_2^\ell, \ldots, \mathbf{a}_M^\ell, \mathbf{g}^\ell\}$.

Our design of executor is inspired by Neural Turing Machines [6], where data is fetched from an external memory using a read head, and subsequently processed by a controller,

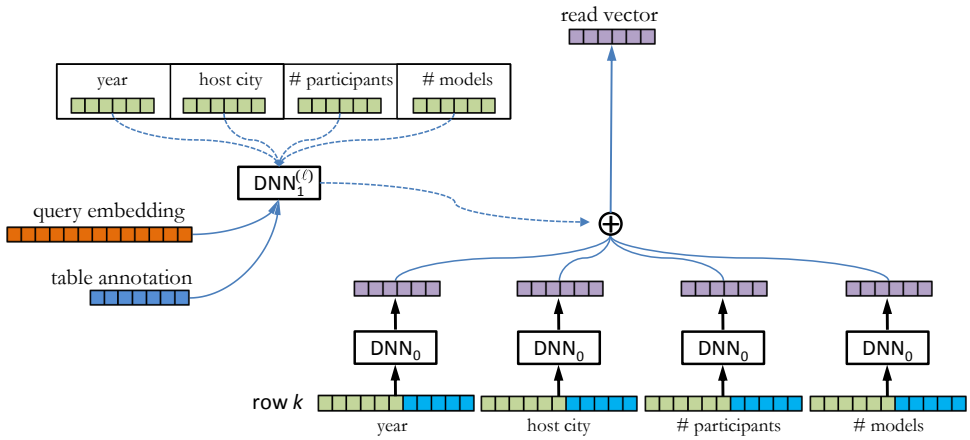


Figure 3: Illustration of the Reader for Executor-ℓ.

whose outputs are flushed back in to memories. An executor functions similarly by reading data from each row of the table, using a Reader, and then calling an Annotator to calculate intermediate computational results as annotations, which are stored in the executor's memory. We assume that row annotations are able to handle operations which require only row-wise, local information (e.g., `select`, `where`), while table annotations can model superlative operations (e.g., `max`, `min`) by aggregating table-wise, global execution results. Therefore, a combination of row and table annotations enables Neural Enquirer to capture a variety of real-world query operations.

3.3.1 Reader

As illustrated in Figure 3, an executor at Layer-ℓ reads in a vector $\mathbf{r}_m^\ell$ for each row m, defined as the weighted sum of composite embeddings for entries in this row:

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

where $\tilde{\omega}(\cdot)$ is the normalized attention weights given by:

$$\tilde{\omega}(\mathbf{f}_n, \mathbf{q}, \mathbf{g}^{\ell-1}) = \frac{\exp(\omega(\mathbf{f}_n, \mathbf{q}, \mathbf{g}^{\ell-1}))}{\sum_{n'=1}^{N} \exp(\omega(\mathbf{f}_{n'}, \mathbf{q}, \mathbf{g}^{\ell-1}))} \tag{3}$$

and $\omega(\cdot)$ is modeled as a DNN (denoted as DNN$_1^{(\ell)}$).

Note that the $\tilde{\omega}(\cdot)$ is agnostic to the values of entries in the row, i.e., in an executor all rows share the same set of weights $\tilde{\omega}(\cdot)$. Since each executor models a specific type of computation, it should only attend to a subset of entries pertain to its execution, which is modeled by the Reader. This is related to the content-based addressing of Neural Turing Machines [6] and the attention mechanism in neural machine translation models [2].

3.3.2 Annotator

In Executor-ℓ, the Annotator computes row and table annotations based on the fetched read vector $\mathbf{r}_m^\ell$ of the Reader, which are then stored in the ℓ-th memory layer $\mathcal{M}^\ell$ accessible to Executor-(ℓ+1). This process is repeated in intermediate layers, until the executor in the last layer to finally generate the answer.

Row annotations A row annotation encodes the local computational result on a specific row. As illustrated in Figure 4, a row annotation for row m in Executor-ℓ, given by

$$\mathbf{a}_m^\ell = f_{\text{A}}^\ell(\mathbf{r}_m^\ell, \mathbf{q}, \mathcal{M}^{\ell-1}) = \mathsf{DNN}_2^{(\ell)}([\mathbf{r}_m^\ell; \mathbf{q}; \mathbf{a}_m^{\ell-1}; \mathbf{g}^{\ell-1}]). \tag{4}$$

fuses the corresponding read vector $\mathbf{r}_m^\ell$, the results saved in previous memory layer (row and table annotations $\mathbf{a}_m^{\ell-1}$, $\mathbf{g}^{\ell-1}$), and the query embedding $\mathbf{q}$. Basically,

- row annotation $\mathbf{a}_m^{\ell-1}$ represents the local status of the execution before Layer-ℓ;
- table annotation $\mathbf{g}^{\ell-1}$ summarizes the global status of the execution before Layer-ℓ;
- read vector $\mathbf{r}_m^\ell$ stores the value of current attention;
- query embedding $\mathbf{q}$ encodes the overall execution agenda,

all of which are combined through $\mathsf{DNN}_2^{(\ell)}$ to form the annotation of row m in the current layer.

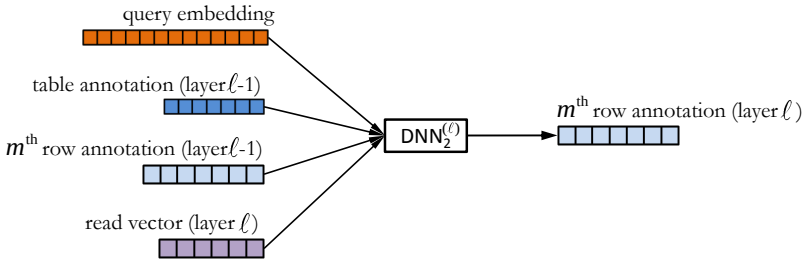


Figure 4: Illustration of Annotator for Executor-ℓ.

Table annotations Capturing the global execution state, a table annotation is summarized from all row annotations via a global pooling operation. In our implementation of Neural Enquirer we employ max pooling:

$$\mathbf{g}^\ell = f_{\text{Pool}}(\mathbf{a}_1^\ell, \mathbf{a}_2^\ell, \ldots, \mathbf{a}_M^\ell) = [g_1, g_2, \ldots, g_{d_\mathcal{G}}]^\top \tag{5}$$

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. It is possible to use other pooling operations (e.g., gated pooling), but we find max pooling yields the best results.

3.3.3 Last Layer of Executor

Instead of computing annotations based on read vectors, the last executor in Neural Enquirer directly outputs the probability of the value of each entry in $\mathcal{T}$ being the answer:

$$p(w_{mn}|Q, \mathcal{T}) = \frac{\exp(f_{\text{ANS}}^\ell(\mathbf{e}_{mn}, \mathbf{q}, \mathbf{a}_m^{\ell-1}, \mathbf{g}^{\ell-1}))}{\sum_{m'=1}^{M}\sum_{n'=1}^{N}\exp(f_{\text{ANS}}^\ell(\mathbf{e}_{m'n'}, \mathbf{q}, \mathbf{a}_{m'}^{\ell-1}, \mathbf{g}^{\ell-1}))} \tag{6}$$

where $f_{\text{ANS}}^\ell(\cdot)$ is modeled as a DNN. Note that the last executor, which is devoted to returning answers, carries out a specific kind of execution using $f_{\text{ANS}}^\ell(\cdot)$ based on the entry value, the query, and annotations from previous layer.

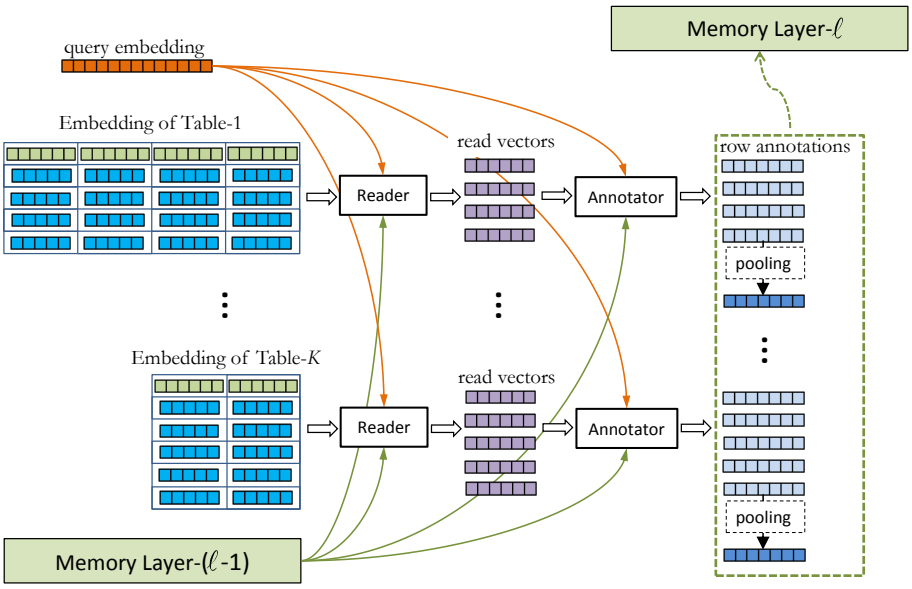


Figure 5: Executor-$(\ell, 1)$ and Executor-(ℓ, K) in multiple tables case

3.4 Handling Multiple Tables

Real-world KBs are often modeled by a schema involving various tables, where each table stores a specific type of factual information. In this section we present NEURAL ENQUIRER-M, adapted for simultaneously operating on multiple KB tables. A key challenge in this scenario is that the multiplicity of tables requires modeling interaction between them. For example, NEURAL ENQUIRER-M needs to serve *join* queries, whose answer is derived by joining fields in different tables.

Basically, NEURAL ENQUIRER-M assigns an executor to each table $\mathcal{T}_k$ in every execution layer ℓ, denoted as Executor-(ℓ, k). Figure 5 pictorially illustrates Executor-$(\ell, 1)$ and Executor-(ℓ, K) working on Table-1 and Table-K respectively. Within each executor, the Reader is designed the same way as single table case, while we modify the Annotator to let in the information from other tables. More specifically, for Executor-(ℓ, k), we modify its Annotator by extending Eq. (4) to leverage computational results from other tables when computing the annotation for the m-th row:

$$\begin{aligned}\mathbf{a}^{\ell}_{k,m} &= f^{\ell}_{\mathrm{A}}(\mathbf{r}^{\ell}_{k,m}, \mathbf{q}, \mathbf{a}^{\ell-1}_{k,m}, \mathbf{g}^{\ell-1}_{k}, \hat{\mathbf{a}}^{\ell-1}_{k,m}, \hat{\mathbf{g}}^{\ell-1}_{k}) \\ &= \mathsf{DNN}^{(\ell)}_{2}([\mathbf{r}^{\ell}_{k,m}; \mathbf{q}; \mathbf{a}^{\ell-1}_{k,m}; \mathbf{g}^{\ell-1}_{k}; \hat{\mathbf{a}}^{\ell-1}_{k,m}; \hat{\mathbf{g}}^{\ell-1}_{k}])\end{aligned}$$

This process is illustrated in Figure 6. Note that we add subscripts $k \in [1, K]$ to the notations to index tables. To model the interaction between tables, the Annotator incorporates the "relevant" row annotation, $\hat{\mathbf{a}}^{\ell-1}_{k,m}$, and the "relevant" table annotation, $\hat{\mathbf{g}}^{\ell-1}_{k}$ derived from the previous execution results of other tables when computing the current row annotation.

A relevant row annotation stores the data fetched from row annotations of other tables, while a relevant table annotation summarizes the table-wise execution results from other tables. We now describe how to compute those annotations. First, for each table $\mathcal{T}_{k'}$, $k' \neq k$, we fetch a relevant row annotation $\bar{\mathbf{a}}^{\ell-1}_{k,k',m}$ from all row annotations $\{\mathbf{a}^{\ell-1}_{k',m'}\}$ of $\mathcal{T}_{k'}$ via attentional reading:

$$\bar{\mathbf{a}}^{\ell-1}_{k,k',m} = \sum_{m'=1}^{M_{k'}} \frac{\exp(\gamma(\mathbf{r}^{\ell}_{k,m}, \mathbf{q}, \mathbf{a}^{\ell-1}_{k,m}, \mathbf{a}^{\ell-1}_{k',m'}, \mathbf{g}^{\ell-1}_{k}, \mathbf{g}^{\ell-1}_{k'}))}{\sum_{m''=1}^{M_{k'}} \exp(\gamma(\mathbf{r}^{\ell}_{k,m}, \mathbf{q}, \mathbf{a}^{\ell-1}_{k,m}, \mathbf{a}^{\ell-1}_{k',m''}, \mathbf{g}^{\ell-1}_{k}, \mathbf{g}^{\ell-1}_{k'}))} \mathbf{a}^{\ell-1}_{k',m'}.$$

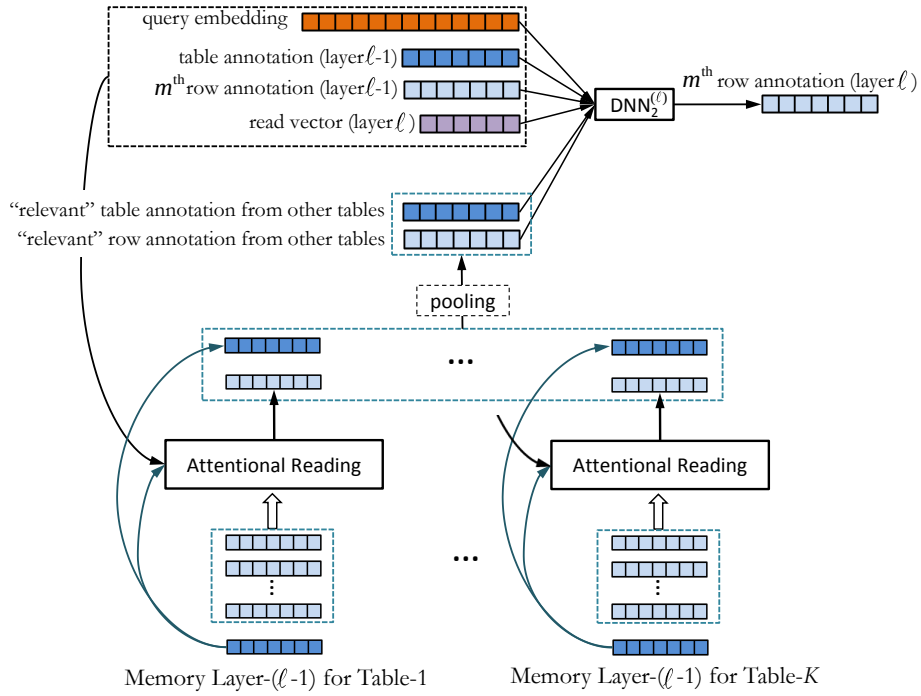


Figure 6: Illustration of Annotator for multiple tables case

Intuitively, the attention weight $\gamma(\cdot)$ (modeled by a DNN) captures how important the m'-th row annotation from table $\mathcal{T}_{k'}$, $\mathbf{a}_{k',m'}^{\ell-1}$ is with respect to the current step of execution. After getting the set of row annotations fetched from all other tables, $\{\bar{\mathbf{a}}_{k,k',m}^{\ell-1}\}_{k'=1,k'\neq k}^{K}$, we then compute $\hat{\mathbf{a}}_{k,m}^{\ell-1}$ and $\hat{\mathbf{g}}_{k}^{\ell-1}$ via a pooling operation[2] on $\{\bar{\mathbf{a}}_{k,k',m}^{\ell-1}\}_{k'=1,k'\neq k}^{K}$ and $\{\mathbf{g}_{k'}^{\ell-1}\}_{k'=1,k'\neq k}^{K}$:

$$\langle \hat{\mathbf{a}}_{k,m}^{\ell-1}, \hat{\mathbf{g}}_{k}^{\ell-1} \rangle = \hat{f}_{\text{POOL}}(\{\bar{\mathbf{a}}_{k,1,m}^{\ell-1}, \bar{\mathbf{a}}_{k,2,m}^{\ell-1}, \ldots, \bar{\mathbf{a}}_{k,K,m}^{\ell-1}\}; \{\mathbf{g}_1^{\ell-1}, \mathbf{g}_2^{\ell-1}, \ldots, \mathbf{g}_K^{\ell-1}\}).$$

In summary, relevant row and table annotations encode the local and global computational results from other tables. By incorporating them into calculating row annotations, Neural Enquirer-M is capable of answering queries that involve interaction between multiple tables.

Finally, Neural Enquirer-M outputs the ranked list of answer probabilities by normalizing the value of $g(\cdot)$ in Eq. (6) over each entry for very table.

4 Learning

Neural Enquirer can be trained in an *end-to-end* (N2N) fashion in Question Answering tasks. During training, both the representations of queries and table entries, as well as the execution logic captured by weights of executors are learned. More specifically, given a set of $N_{\mathcal{D}}$ query-table-answer triples $\mathcal{D} = \{(Q^{(k)}, \mathcal{T}^{(i)}, y^{(i)})\}$, we optimize the model parameters by maximizing the log-likelihood of gold-standard answers:

$$\mathcal{L}_{\text{N2N}}(\mathcal{D}) = \sum_{i=1}^{N_{\mathcal{D}}} \log p(y^{(i)} = w_{mn} | Q^{(i)}, \mathcal{T}^{(i)}) \tag{7}$$

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

[2]This operation is trivial in our experiments on two tables.

This can be alleviated by softly guiding the learning process via controlling the attention weights $\tilde{w}(\cdot)$ in Eq. (3). By enforcing $\tilde{w}(\cdot)$ to bias towards a field pertain to a specific operation, we can "coerce" the executor to figure out the logic of this particular operation relative to the field. As an example, for Executor-1 in Figure 1, by biasing the weight of the `year` field towards 1.0, only the value of `year` field will be fetched and sent for computing annotations, in this way we can force the executor to learn how to execute the `where` clause (`where year` $<$ `2008`). This setting will be referred to as *step-by-step* (SbS) training. Formally, this is done by introducing additional supervision signal to Eq. (7):

$$\mathcal{L}_{\text{SbS}}(\mathcal{D}) = \sum_{i=1}^{N_{\mathcal{D}}} [\log p(y^{(i)} = w_{mn} | Q^{(i)}, \mathcal{T}^{(i)}) + \alpha \sum_{\ell=1}^{L} \log \tilde{w}(\mathbf{f}^{\star}_{k,\ell}, \cdot, \cdot)] \tag{8}$$

where α is a scalar and $\mathbf{f}^{\star}_{k,\ell}$ is the embedding of the field name known *a priori* to be relevant to the executor at Layer-ℓ in the k-th example.

5 Experiments

In this section we evaluate Neural Enquirer on synthetic QA tasks with queries with varying compositional depth. We will first briefly describe our synthetic QA task for benchmark and experimental setup, and then discuss the results under different settings.

5.1 Synthetic QA Task

We present a synthetic QA task to evaluate the performance of Neural Enquirer, where a large amount of QA examples at various levels of complexity are generated to evaluate the single table and multiple tables cases of the model. Our data has the same complexity as a real-world one, but we manipulate it (e.g., reshuffling of entry values) to generate enough training instances. Starting with "artificial" tasks eases the process of developing novel deep models [14], and has gained increasing popularity in recent advances of the research on modeling symbolic computation using DNNs [6, 15].

Our synthetic dataset consists of query-table-answer triples $\{(Q^{(i)}, \mathcal{T}^{(i)}, y^{(i)})\}$. To generate such a triple, we first randomly sample a table $\mathcal{T}^{(i)}$ of size 10×10 from a synthetic schema of Olympic Games, which has 10 fields and 60 values for city and country respectively, 6 values for continents names, and 120 numbers for rest of the fields. Figure 7 gives an example table with one row. Next, we generate a query $Q^{(i)}$ associated with its gold-standard answer $y^{(i)}$ on $\mathcal{T}^{(i)}$ using predefined templates. Our synthetic QA task consists of four types of queries in pseudo SQL style, ranging from simple "select-where" queries to more complex nest queries involving multiple steps of computation. Table 1 summarizes those queries. In our experiments we use this procedure to generate benchmark datasets of different sizes. Each dataset of size M contains M training/testing examples respectively, which is generated by sampling $2M$ examples and then split them to two sets of equal size. To make the artificial task harder, we enforce that all queries in the testing set do not appear in the training set[3]. Additionally, for multiple tables task (Section 5.6), we sample two tables for each example, which consist of one *Olympic Game* table of size 10×7 and one *Country* table of size 10×4, as illustrated in Figure 8 (only show one row). The *country* field is the key linking the two tables.

[3]This may make the sizes of training/testing sets slightly different

year	host_city	#_participants	#_medals	#_duration	#_audience	country	continent	country_size	population
2008	Beijing	3,500	4,200	30	67,000	China	Asia	960	130

Figure 7: Single table example in the synthetic QA task (only show one row)

Table-1

country	year	host_city	#_participants	#_medals	#_duration	#_audience
China	2008	Beijing	3,500	4,200	30	67,000

Table-2

country	continent	population	country_size
China	Asia	130	960

Figure 8: Multiple tables example in the synthetic QA task (only show one row)

5.2 Setup

We use the same configuration for all testing cases. All DNNs in Neural Enquirer are instantiated with one hidden layer, except for the last executor, which has two hidden layers. We set the dimensionality of word/entity embeddings and row/table annotations to 20, hidden layers to 50, and the hidden states of the GRU in query encoder to 100. α in Eq. (8) is set to 0.5. We pad the beginning of all input queries to a fixed size.

Neural Enquirer is trained via standard back-propagation. Objective functions are optimized using SGD in a mini-batch of size 100 with adaptive learning rates (AdaDelta [16]). The model converges fast within 200 epochs. Unless otherwise noted, results reported in Section 5.3 are obtained using *end-to-end* (N2N) training setting (Eq. 7). We evaluate the performance of Neural Enquirer in terms of accuracy, defined as the fraction of correctly answered queries.

5.3 Main Results

To simulate the read-world scenario where queries of various types are issued to the model, we constructed a *mixtured* dataset with $110K$ examples, consisting of $60K$ *simple* queries (Select_Where, Superlative and Where_Superlative), with $20K$ for each type, and $50K$ complex Nest queries. We trained a Neural Enquirer model with five executors using end-to-end (N2N) training setting (Eq. 7). The results are summarized in the first two rows of Table 2[4]. We also break down the overall accuracy (85.2%) on this dataset to study the performance for each individual type. Neural Enquirer is very effective in answering simple queries like Select_Where, Superlative and Where_Superlative, whose accuracies are above 96%. Nest queries, which are much more complicated, also registers a decent performance of 68%. These results suggest that our proposed model is very effective in answering complex, compositional queries.

To further understand why our model is capable of handling compositional queries, we study the attention weights $\tilde{w}(\cdot)$ of Readers (Eq. 3) for executors in intermediate layers, and the answer probability (Eq. 6) the last executor outputs for each entry in the table. We randomly sampled two queries (Q_1 in Figure 9 and Q_2 in Figure 10) in the mixtured dataset that our model answers correctly and visualized their corresponding values (cf. Figure 9 and 10). We find that each executor actually *learns* its execution logic from just the correct answers in end-to-end training. For Q_1, the model executes the query 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 *country_size* field as input and computes

[4]This mixed training setting yields performance slightly better than each of four types of queries trained separately (See Appendix B), suggesting that Neural Enquirer for one particular query type can benefit from the instances for other types.

Query Type	Example Queries
Select_Where	`select #_participants where host_city = Beijing` `select country where year = 2012`
Superlative	`argmax(host_city, year)` `argmin(country, population)`
Where_Superlative	`where #_medals < 3,000, argmin(country, #_duration)` `where country_size < 770, argmax(host_city, #_audience)`
Nest	`where host_city=Beijing,select year as A,where year>A,argmin(host_city,year)` `where year=1992,select population as A,where population<A,argmax(#_medals,#country_size)`

Table 1: Example queries in our synthetic QA task

Overall accuracy (N2N)	85.2%			
Breakdown accuracy (N2N)	Select_Where 99.7%	Superlative 99.8%	Where_Superlative 96.9%	Nest 68.0%
Overall accuracy(SbS)	99.8%			
Breakdown accuracy (SbS)	Select_Where 100%	Superlative 100%	Where_Superlative 99.8%	Nest 99.7%

Table 2: Performance on mixtured dataset

intermediate annotations, while Executor-4 focuses on the *#_duration* field. Finally, the last executor outputs high probability for the *#_audience* field (the 6-th column) in the 5-th 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. We can guess confidently that in executing Q_1, Executor-3 performs the conditional filtering operation (`where` clause in Q_1), and Executor-4 performs the first part of `argmax` (find the maximum value of *#_duration*), while the last executor finishes the execution by assigning high probability for the *#_audience* field of the row with the maximum value of *#_duration*.

Compared with the relatively simple Q_1, Q_2 is more complicated, with two extra `where` and `select` operations, and requires five steps of execution. From the weights visualized in figure 10, we can find that the last three executors function similarly as the case in answering Q_1, yet the execution logic for the first two executors is a bit obscure. We posit that this is because during 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 gradient problem.

We also investigate the case where our model fails to deliver the correct answer for complicated queries. Figure 11 gives such a query Q_3 together with visualized weights. Similar as Q_2, Q_3 requires five steps of execution. Besides messing up the weights in the first two executors, the last executor, Executor-5, predicts a wrong entry as the query answer, instead of the highlighted (in red rectangle) correct entry.

5.4 With Additional Step-by-Step Supervision

To alleviate the vanishing gradient problem when training on complex queries as described in Section 5.3, in our next set of experiments we trained our Neural Enquirer model using step-by-step (SbS) training (Eq. 8), where we encourage each executor to attend to a specific field that is known *a priori* to be relevant to its execution logic. The results are shown in the last two rows of Table 2. With stronger supervision signal, the model significantly outperforms the results in end-to-end setting, and achieves near 100% accuracy on all types of queries, which shows that our proposed Neural Enquirer is capable of leveraging the additional supervision signal given to intermediate layers in SbS training

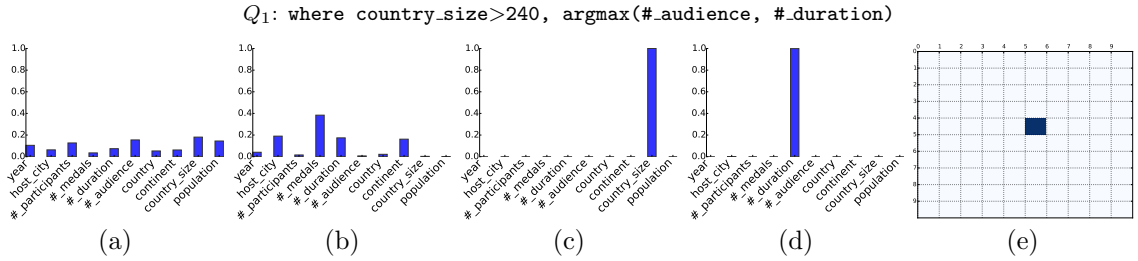


Figure 9: Weights visualization of query Q_1. (a)-(d) attention weights for intermediate executors (Executor-1 to Executor-4); (e) answer probability of each entry given by the last executor (Executor-5)

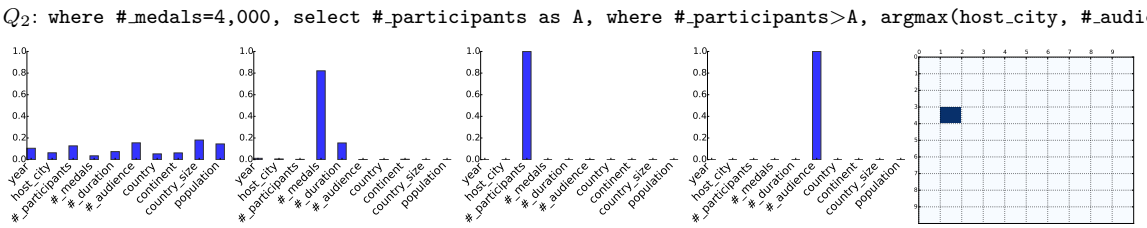


Figure 10: Weights visualization of query Q_2

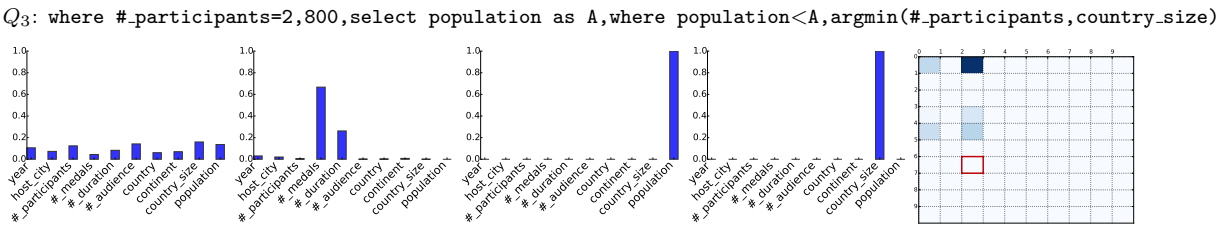


Figure 11: Weights visualization of query Q_3 (an incorrectly answered query)

setting, and answering complex and compositional queries with perfect accuracy.

Let us revisit the query Q_2 in SbS setting with the weights visualization in Figure 12. In contrast to the result in N2N setting (Figure 10) where the attention weights for the first two executors are obscure, the weights in every executor are perfectly skewed towards the actual field pertain to each layer of execution (with a weight 1.0). Quite interestingly, the attention weights for Executor-3 and Executor-4 are exactly the same with the result in N2N setting, while the weights for Executor-1 and Executor-2 are significantly different, suggesting Neural Enquirer learned a different execution logic in the SbS setting.

5.5 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, which is particularly severe for QA. It is hard to cover existing tail entities, while at the same time new entities appear in user-issued queries and back-end KB everyday. Quite interestingly, we find that a simple variation of Neural Enquirer is able to handle unseen entities almost without any loss of accuracy.

Basically, we divide the words in the vocabulary into *operation* words and *entity* words. Operation words contain all the numbers (e.g., *90*) and operator names (e.g., *select*), whose embeddings carry semantic meaning relevant to execution and should be optimized during training; while embeddings for entity words (e.g., *Beijing*, *China*) function in a way as index to facilitate the matching between entities in queries and tables during the layer-by-layer execution of Neural Enquirer. After randomly initializing the embedding matrix

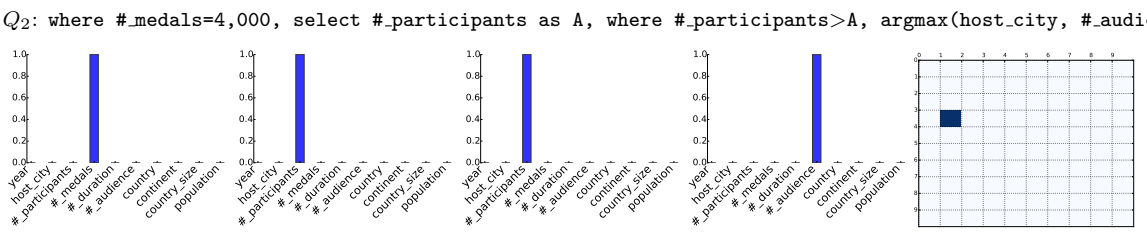


Figure 12: Weights visualization of query Q_2 in step-by-step training setting

Query Type	SELECT_WHERE	SUPERLATIVE	WHERE_SUPERLATIVE	NEST
acc. OOV Setting	99.3%	99.6%	93.7%	62.6%
acc. Non-OOV Setting	99.9%	99.8%	95.0%	62.7%

Table 3: Comparable results of OOV and Non-OOV settings

L, we only update the embeddings of operation words in training, while keeping those of entity words unchanged. For each type of query we trained a separate model on $20K$ examples with this new procedure, while testing its performance on another $20K$ examples sampled purely using OOV entities (i.e., all country and city names unseen in the training set). We compare the results in this OOV setting with models trained/tested on datasets of the same size and for the same type of query, while without any OOV entities[5]. Table 3 lists the results. As it shows NEURAL ENQUIRER training in this OOV setting yields performance comparable to that in the non-OOV setting, indicating that operation words and entity words play different roles in query execution, whose properties are worth future investigation.

5.6 Multiple Tables Queries

In our final set of experiments we present preliminarily results for NEURAL ENQUIRER-M, which we evaluated on SELECT_WHERE queries. We sampled a dataset of $100K$ SELECT_WHERE queries on two tables, out of which roughly half size of queries (denoted as "Join") require joining the two tables to derive answers. We tested on a model with three executors. Table 4 lists the results. The accuracy of join queries is lower than that of non-join queries, which is caused by additional interaction between the two tables involved in answering join queries.

Query Type	Non-Join	Join	Overall
Accuracy	99.7%	81.5%	91.3%

Table 4: Accuracies of SELECT_WHERE queries on two tables

We find that NEURAL ENQUIRER-M is capable of identifying that the *country* field is the foreign key linking the two tables. Figure 13 illustrates the attention weights for a correctly answered join query Q_4. Although the query does not contain any hints for the foreign key (*country* field), Executor-$(1, 1)$ (the executor at Layer-1 on Table-1) operates on an ensemble of embeddings of the *country* and *year* fields, whose outputting row annotations (contain information of both the key *country* and the value *year*) are sent to Executor-$(2, 2)$ to compare with the *country* field in Table-2. We posit that the result

[5]This is essentially one set of experiments (the fourth column in Table 5) reported in Appendix B.

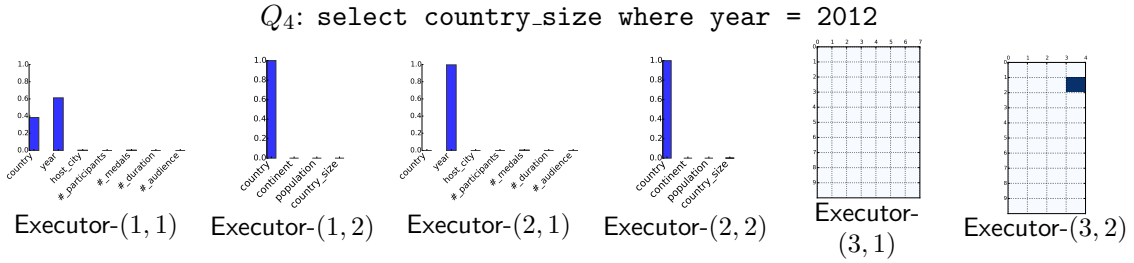


Figure 13: Weights visualization of query Q_4

of comparison is stored in the row annotations of Executor-$(2, 2)$ and subsequently sent to the executors at Layer-3 for computing the answer probability for each entry in the two tables.

6 Related Work

Our work falls into the research area of Semantic Parsing, where the key problem is to parse Natural Language queries into logical forms executable on KBs. Classical approaches for Semantic Parsing can be broadly divided into two categories. The first line of research resorts to the power of grammatical formalisms (e.g., Combinatory Categorial Grammar) to parse NL queries and generate corresponding logical forms, which requires curated/learned lexicons defining the correspondence between NL phrases and symbolic constituents [17, 7, 1, 18]. The model is tuned with annotated logical forms, and is capable of recovering complex semantics from data, but often constrained on a specific domain due to scalability issues brought by the crisp grammars and the lack of annotated training data. Another line of research takes a semi-supervised learning approach, and adopts the results of query execution (i.e., answers) as supervision signal [5, 3, 10, 11, 12]. The parsers, designed towards this new learning paradigm, take different types of forms, ranging from generic chart parsers [3, 11] to more specifically engineered, task-oriented ones [12, 8]. Semantic parsers in this category often scale to open domain knowledge sources, but lack the ability of understanding compositional queries because of the intractable search space incurred by the flexibility of parsing algorithms. Our work follows this line of research in using query answers as indirect supervision to facilitate end-to-end training using QA tasks, but performs semantic parsing in distributional spaces, where logical forms are "neuralized" to an executable distributed representation.

Our work is also related to the recent advances of modeling symbolic computation using Deep Neural Networks. Pioneered by the development of Neural Turing Machines (NTMs) [6], this line of research studies the problem of using differentiable neural networks to perform "hard" symbolic execution. As an independent line of research with similar flavor, Zaremba et al. [15] designed a LSTM-RNN to execute simple Python programs, where the parameters are learned by comparing the neural network output and the correct answer. Our work is related to both lines of work, in that like NTM, we heavily use external memory and flexible way of processing (e.g., the attention-based reading in the operations in Reader) and like [15], Neural Enquirer learns to execute a sequence with complicated structure, and the model is tuned from the executing them. As a highlight and difference from the previous work, we have a deep architecture with multiple layer of external memory, with the neural network operations highly customized to querying KB tables.

Perhaps the most related work to date is the recently published Neural Programmer proposed by Neelakantan et al. [9], which studies the same task of executing queries

on tables with Deep Neural Networks. NEURAL PROGRAMMER uses a neural network model to select operations during query processing. While the query planning (i.e., which operation to execute at each time step) phase is modeled softly using neural networks, the symbolic operations are predefined by users. In contrast NEURAL ENQUIRER is fully distributional: it models both the query planning and the operations with neural networks, which are jointly optimized via end-to-end training. Our NEURAL ENQUIRER model learns symbolic operations using data-driven approach, and demonstrates that a fully neural, end-to-end differentiable system is capable of modeling and executing compositional arithmetic and logic operations upto certain level of complexity.

7 Conclusion and Future Work

In this paper we propose NEURAL ENQUIRER, a fully neural, end-to-end differentiable network that learns to execute queries on 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 [11], 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*, pages 1699–1710, 2015.

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

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

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

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

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

[7] T. Kwiatkowski, E. Choi, Y. Artzi, and L. S. Zettlemoyer. Scaling semantic parsers with on-the-fly ontology matching. In *EMNLP*, pages 1545–1556, 2013.

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

[9] A. Neelakantan, Q. V. Le, and I. Sutskever. Neural Programmer: Inducing Latent Programs with Gradient Descent. *ArXiv e-prints*, Nov. 2015.

[10] P. Pasupat and P. Liang. Zero-shot entity extraction from web pages. In *ACL (1)*, pages 391–401, 2014.

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

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

[13] 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*, pages 1711–1721, 2015.

[14] 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.

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

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

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

[18] L. S. Zettlemoyer and M. Collins. Online learning of relaxed ccg grammars for parsing to logical form. In *EMNLP-CoNLL*, pages 678–687, 2007.

A Computation of GRU

Given a sequence of word embeddings in Q: $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T\}$, at each time step t, the GRU computes the hidden state $\mathbf{h}_t$ as follows:

$$\begin{aligned}
\mathbf{h}_t &= \mathbf{z}_t \mathbf{h}_{t-1} + (\mathbf{1} - \mathbf{z}_t)\tilde{\mathbf{h}}_t \\
\tilde{\mathbf{h}}_t &= \tanh(\mathbf{W}\mathbf{x}_t + \mathbf{U}(\mathbf{r}_t \circ \mathbf{h}_{t-1})) \\
\mathbf{z}_t &= \sigma(\mathbf{W}_z \mathbf{x}_t + \mathbf{U}_z \mathbf{h}_{t-1}) \\
\mathbf{r}_t &= \sigma(\mathbf{W}_r \mathbf{x}_t + \mathbf{U}_r \mathbf{h}_{t-1})
\end{aligned}$$

where $\mathbf{W}$, $\mathbf{W}_z$, $\mathbf{W}_r$, $\mathbf{U}$, $\mathbf{U}_z$, $\mathbf{U}_r$ are parametric matrices, $\mathbf{1}$ the column vector of all ones, and $\circ$ element-wise multiplication. We use the last hidden state, h_T as the vectorial representation of the query, i.e., $\mathbf{q} = \mathbf{h}_T$.

B Performance on Separate Datasets

We also studied the separate performance of Neural Enquirer in answering different types of queries on various sizes of datasets. Table 5 lists the results. This time for each type of query we used a model with an optimized number of executors, as indicated in the table. Neural Enquirer achieves 100% accuracy on simple queries when the size of training data grows to $50K$. It is also worth noting that, for Select_Where and Superlative queries that need a minimal steps of execution, their accuracies are already very close to 100% using only $5K$ training examples. Additionally, comparing the results obtained here with their counterparts in mixtured setting (second row in Table 2), we can observe an interesting fact that, the numbers reported here are actually lower, even though the model is tuned for a specific type of query with an optimal number of executors. As an example, the accuracy of Where_Superlative queries on $20K$ dataset is 95.0%, while in the mixtured dataset, Neural Enquirer achieves an accuracy of 96.9% (when tuned on $20K$ Where_Superlative queries together with other types of queries). This could be due to that different queries share some common structures, in mixtured training the model is presented with more information. Meanwhile, different types of queries act as regularizers to prevent over-fitting.

Query Type	$5K$	$10K$	$20K$	$50K$
Select_Where (2 executors)	99.9%	99.9%	99.9%	100%
Superlative (2 executors)	98.5%	98.8%	99.8%	100%
Where_Superlative (3 executors)	50.4%	75.8%	95.0%	100.0%
Nest (5 executors)	38.8%	54.4%	62.7%	65.0%

Table 5: Performance for different types of queries