Deep Neural Parsing for Database Query

Hongyu Xiong
Department of Applied Physics
Stanford University
Stanford, CA 94305
hxiong2@stanford.edu

Kaifeng Chen
Department of Applied Physics
Stanford University
Stanford, CA 94305
kfchen@stanford.edu

Yinglan Ma
Department of Computer Science
Stanford University
Stanford, CA 94305
yinglanm@stanford.edu

Haixun Wang
Search NLP Team
Facebook, Inc.
Menlo Park, CA 94025
haixun@fb.com

Abstract

Question-Answering (QA) problem has been a long interest to the machine learning community. In this paper, we propose a deep neural parser for QA on table/database queries and searches. We transform this problem into building and training a sequence-to-sequence (seq2seq) deep neural network to translate natural language queries to SQL-like logical forms, which could then be used as intermediate states to query into the table/database. To improve the model, we use GloVe pretrained vectors to represent words and tag each word in the queries with possible field name in the corresponding table. We augment a new dataset based on the Wikitable dataset [1] for training and testing, and our model finally achieves a 74.7% training accuracy and 71.0% dev accuracy on this dataset, which gives a new benchmark in research related to Wikitable.

1 Introduction

1.1 Problem Statement

Question-Answering problem has been continuously attracting the interests of the machine learning community, since the way to establish query-and-response is essential for machines to achieve higher level of artificial intelligence (AI). Specifically, semantic parsing of the question should pave the way for the following access into the knowledge base for answers. Using tables/databases as knowledge bases would be a starting point, since the structure of table is simple and the methods developed on small tables could be scaled to massive database. However, the current methods developed either require huge grammatical or logical analysis [1] or serve as a demonstrative toy which is difficult to scale on larger database [2]. Here, we propose to apply deep neural network to parse natural language queries for answers. This technology will benefit both end users and domain experts in application related to database queries.

The general approach of our work is to turn this Question-Answering problem into a translation problem. We will build and train a sequence-to-sequence (seq2seq) neural network to translate the natural language queries into a SQL-like logical sentences which will be used to query into the database. In order to incorporate the table information into the model, we have tried two methods: (1) tag each word in the query with related field name, and (2) directly concatenate all the field names of the table in front of the query. We use GloVe pretrained vectors to represent the words in

the query (disable training) and use trainable embedding to represent the tag and the token in the logical forms.

We introduce our datasets with more details in section 1.2. Simply speaking, we mainly use the Wikitable dataset [1] in this work for the evaluation of our seq2seq model. In order to train our model, we prepare the data to the similar forms used in the synthetic dataset of Neural Enquirer [2], and increase the amount of data by augmentation; then we randomly divide the dataset to training set and development set. We report the training accuracy and development accuracy of our model and compare with the previous work in section. 4.

Year	City	Country	Nations
1896	Athens	Greece	14
1900	Paris	France	24
1904	St. Louis	USA	12
...	...	...	...
2004	Athens	Greece	201
2008	Beijing	China	204
2012	London	UK	204

x_1: *"Greece held its last Summer Olympics in which year?"*
y_1: {2004}
x_2: *"In which city's the first time with at least 20 nations?"*
y_2: {Paris}
x_3: *"Which years have the most participating countries?"*
y_3: {2008, 2012}
x_4: *"How many events were in Athens, Greece?"*
y_4: {2}
x_5: *"How many more participants were there in 1900 than in the first year?"*
y_5: {10}

```
==========================
query: how long is the game that has 320 medals <eos>
table:
(#_medals, 510) (#_participants, 190) (#_duration, 130) (#_audience, 70) (country_gdp, 430) (year, 2172)
(country_size, 520) (country_population, 420) (host_city, Los_Angeles) (country, Vietnam)
(#_medals, 540) (#_participants, 70) (#_duration, 30) (#_audience, 30) (country_gdp, 370) (year, 2088)
(country_size, 460) (country_population, 470) (host_city, Amsterdam) (country, Iraq)
(#_medals, 110) (#_participants, 190) (#_duration, 230) (#_audience, 100) (country_gdp, 570) (year, 2116)
(country_size, 210) (country_population, 580) (host_city, Athens) (country, US)
(#_medals, 310) (#_participants, 420) (#_duration, 310) (#_audience, 80) (country_gdp, 510) (year, 1968)
(country_size, 70) (country_population, 400) (host_city, London) (country, Ukraine)
(#_medals, 250) (#_participants, 430) (#_duration, 150) (#_audience, 410) (country_gdp, 580) (year, 2044)
(country_size, 190) (country_population, 370) (host_city, Oslo) (country, Portugal)
(#_medals, 240) (#_participants, 110) (#_duration, 350) (#_audience, 570) (country_gdp, 370) (year, 1964)
(country_size, 200) (country_population, 340) (host_city, Munich) (country, Russia)
(#_medals, 550) (#_participants, 240) (#_duration, 340) (#_audience, 30) (country_gdp, 560) (year, 2156)
(country_size, 110) (country_population, 110) (host_city, Cortina_Ampezzo) (country, Philippines)
(#_medals, 490) (#_participants, 570) (#_duration, 600) (#_audience, 230) (country_gdp, 450) (year, 1952)
(country_size, 290) (country_population, 200) (host_city, Macau) (country, Brazil)
(#_medals, 310) (#_participants, 460) (#_duration, 590) (#_audience, 220) (country_gdp, 510) (year, 2000)
(country_size, 70) (country_population, 400) (host_city, Dubai) (country, Ukraine)
(#_medals, 320) (#_participants, 190) (#_duration, 480) (#_audience, 40) (country_gdp, 190) (year, 1960)
(country_size, 200) (country_population, 100) (host_city, Cortina_Ampezzo) (country, New_Zealand)

query type: select_where
col_ids_attention: [0, 2]
target answer: 480
==========================
```

Figure 1: The Wikitable dataset we have been focusing on in this paper [1] (left) and a synthetic dataset from Neural Enquirer [2] (right), which we have studied and borrowed ideas from. Both datasets contain natural language queries, table with multiple objects and field entries, and the corresponding answers.

1.2 Dataset

In this work, we use an open source dataset Wikitable [1] for training and evaluation of our model, and study on the synthetic dataset from Neural Enquirer [2] to formulate intermediate SQL-like logical forms (Figure. 1).

The synthetic dataset consists of a training data with $100,000$ [$Query$, $Answer$, $Table$] triples, and a test data with 20000 [$Query$, $Answer$, $Table$] triples, with specific grammar-based questions and logical forms. This dataset contains four fixed logical forms, and has been tested using our model at the first place. However, since this dataset is generated based a small vocabulary (~ 300) and a very limited number of synthetic grammars for each logical form, it is not diverse enough for general Question-Answering purpose, where the complexity of natural language query and continuous evolution of vocabulary are difficult to be included into this few categories.

The Wikitable dataset [1] contains 22,033 queries on 2,108 tables. Since the dataset itself does not contain SQL-like logical forms, we extended the logical-form rules of the synthetic dataset, and annotated nearly 200 selected Wikitable [1] queries with their corresponding SQL-like logical forms. In our augmented dataset, we generate close to 4000 questions and their corresponding logical forms. The dataset then is randomly shuffled and divided into training data and development data.

2 Related Work

Most semantic parsing methods utilize rule-based features and symbolic processing to construct semantic representations of natural language queries. Recent works by Pasupat and Liang[1], Liang et al.[3], and Clarke et al.[4] attempt to backpropagate query execution results to revise the semantic representation of a query. This approach, however, is hindered by the intractable search space due to the complexity and flexibility in natural language.

Another previous work, Neural Enquirer[2], constructs distributed representations of both the knowledge base tables and the queries, and executes queries through a **memory-based neural network** over a few iterations. Ref. [2] claims a close-to 100% accuracy on their own synthetic

dataset using this end-to-end model with step-by-step attention supervision. A similar work Neural Programmer [5] implements a memory-based deep neural network on Wikitable, and is trained end-to-end with weak supervision of question-answer pairs, achieving a 37.7% accuracy, slightly better than [1]. As we can see, the memory-based neural network could perform well on small vocabulary and limited complexity such as the synthetic dataset; however, it is difficult for this type of models to generalize on a growing vocabulary and on more complex dataset with natural language queries. Therefore, for the memory based works, the requirement of large training data on a rather small query table makes it hard to scale to larger database QA, especially when there are a lot of unseen vocabulary. It would be more realistic to first transform the natural language query into some generic intermediate forms which could reliably access to database.

There recently are related works [6, 7] trying to transform natural language queries into logical forms that could be parsed systematically into specific database such as GeoQuery (U.S. geography) and ATIS (Air flight). Ref. [6] uses the Seq2seq and Seq2Tree model to learn the sequential and hierarchical structure of the logical forms; Ref. [7] proposes neural attention to copy specific words in the query directly to the tokens in output logical form, and further uses data augmentation to increase the size of their dataset. We benefit quite a lot from both works. However, both works are not able to incorporate the table information into the model, and the logical forms are hard to generalized to other databases.

3 Approach: Attention-based Seq2Seq Recurrent Neural Network

Our aim is to learn a model which maps natural language input $q = x_1, x_2, ..., x_{|q|}$ to a SQL-like logical form representation of its meaning $a = y_1, y_2, ..., y_{|a|}$, which later is used to make inquiry into the table. The conditional probability $p(a|q)$ is decomposed as:

$$p(a|q) = \Pi_{t=1}^{|a|} p(y_t|y_{<t}, q) \tag{1}$$

The above equation shows that each word generate depends on the whole input query q and the generated words before it.

Here we propose an efficient way to tackle this problem by using sequence to sequence (seq2seq) model with recurrent neural networks (RNN). We show such a modern deep learning approach has many advantages such as fast training and improved accuracy. Our method consists of an encoder which encodes natural language query q into a vector representation $h_{|q|}$ and a decoder following behind to generate $y_1, y_2, ...y_{|a|}$ conditioned on the encoding vector $h_{|q|}$.

3.1 Logical Forms of a Natural Language Query

In general, questions based on a given database table, which are usually natural utterances from human, can be indirectly connected to the answer to the query through logical forms. Logical form is a SQL-like sentence following certain grammar patterns that can be easily understood by machines. For example, one possible logical form of the question "`which nation obtained the most gold medals`" is "`argmax Nation Gold`". With a correct implementation of answer query based on logical forms, the Question-Answering problem is reduced to a Question-Logical-form translation problem. Previous works [3] where Lambda expression is used for the Question to Logical-form translation, suffer from the problems of slow training and the lack of simplicity and elegance.

In our model, we considered a total of 16 different categories of logical forms, including the 4 logical forms proposed in Ref. [2]. Typically, a logical form contains key words like `where` or `select`, which are of the same meanings in SQL. For example, the logical form "`where Year equal 2013 select 2nd_Venue`" selects the value in the field `2nd_Venue` where the field `Year` has the value 2013. In addition to basic `where` and `select` clauses, we also include simple comparison syntaxes such as `Sum`, `Diff` and `Average`. For example, the logical form "`where Country equal Netherlands select Masters as A where Country equal England select Masters as B sum A B`" computes the sum of the number of masters from Netherlands and England.

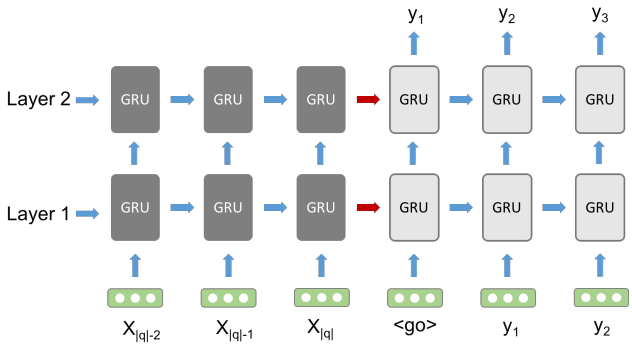

Figure 2: Two-layer sequence-to-sequence recurrent neural network with GRU cell, a typical paradigm for neural translation machine (NTM).

3.2 Attention-based Encoder-Decoder Translation Model

The vanilla model we use is a attention-based encoder-decoder model typically used for neural translation machine (NTM). Both encoder and decoder are recurrent neural networks (RNN) with gate-regulated unit (GRU) for each cell. (Fig. REF) The equation for multilayer NTM is

$$h_t^l = GRU(h_{t-1}^l, h_t^{l-1}) \tag{2}$$

where h_t^l is the hidden state for time t at layer l, and $h_t^0 = x_t$.

In the training process, the natural language queries serve as the encoder inputs, with each word in a query entering the GRU step by step and output a hidden state for the next time step; finally at the last time step the encoder outputs the last hidden state into the decoder, with `<GO>` as the first decoder input; then the tokens in SQL-like logical forms serve as the following decoder outputs; the hidden layer of the topmost GRU h_t^L in the decoder is used to predict the t-th output token:

$$p(y_t|y_{<t}, q) = \text{softmax}(W_o h_t^L)^T e(y_t) \tag{3}$$

where W_o is the output projection weight and $e(y_t) \in \{0,1\}^{|V_a|}$ is a one-hot vector for computing y_t's probability from the predicted distribution. In the evaluation process, the decoder input at each time step t would be the predicted output at former step. In this work, we are using a **greedy decoder**, where we generate the token at each step based on the current maximum probability (however, it does not necessarily yield the sentence with global maximum probability). The perplexities are calculated for the generated logical forms with the decoder inputs; updating are based on the evaluation of perplexities on dev set. (Fig. 2) The decoder also adopts an **attention mechanism** for better word alignment; in this case each decoder hidden vector would also dependent on a linear combination of all former encoder hidden vectors.

The word vectors representing the encoder inputs are the **pretrained GloVe vectors**, which are not update-able during the training due to the small dataset; however, for the embedding matrix of decoder inputs, since the tokens does not necessarily represent the same semantic meaning as the natural language words, we initialize it with xavier initializer and set it trainable during the training.

One possible problem of the vanilla model is that no information of the table is incorporated. Even though incorporating all the entries of a table (like what the Neural Enquirer [2] does) makes the model inscalable, including the field names or simply the name entity would be helpful for word alignment.

3.2.1 Field names/Name entity concatenation

In order to add the table information into the model, we first add a single name entity field in front of the query. For example, if we have the query `how many gold medals the country ranked 14 won` with a table containing field names [`Nation, Rank, Gold, Silver, Bronze, Total`], we know the first one `Country` is the name entity of the object, and we directly put it in front of the query to get `Country how many gold medals the nation ranked 14 won`.

We also take a step further, taking all the field names and directly concatenating them in front of the queries to get a longer input, for example `Country Rank Gold Silver Bronze Total how many gold medals the nation ranked 14 won`. For a better comparison, we test our models on the original query without concatenation as the baseline. (Fig. 3)

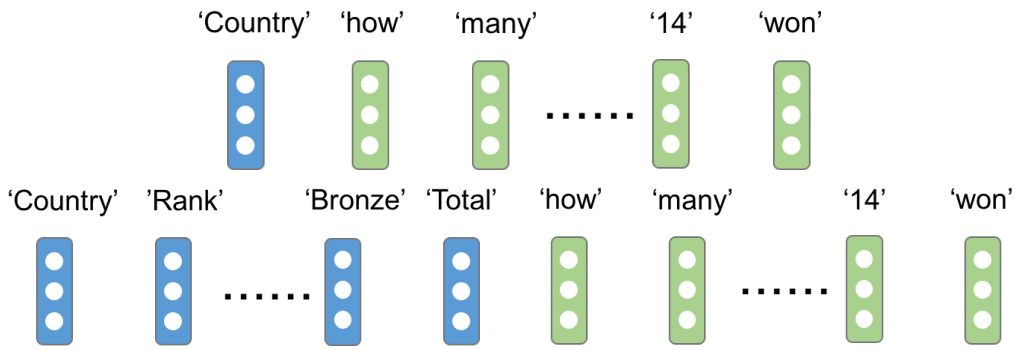

Figure 3: Field information concatenating in front of the original sequence. (Up) single name entity prefixed; (Down) all field names prefixed.

3.2.2 Field information tagging

With field names or single name entity prefixed, the processed query will contain table information thus provide a better learning than vanilla model. However, typically for a database query, not all the field names information is necessary to be included; and simply putting field names in front of the query does no good for the encoder to learn the sequential dependence of the original query itself, since it could be overfitted on the sequence of field names.

Therefore, we move beyond the simple fields concatenation, and try to tag each word in the original query with a specific token indicating information of the field names. **Each word will be tagged with a token.** If a word implicates a field name, then the tagging token is exactly the field name such as `Gold`; if a word is a numerical value, then the tagging token is `<num>`; if a word does not relate to any field names provided or is not a numerical value, then it is tagged with `<nan>`. In this case, only the necessary field names will be tagged with the query and un-related ones will not appear in the tagging. For example, if we have the query `how many gold medals the country ranked 14 won` with a table containing field names [`Nation, Rank, Gold, Silver, Bronze, Total`], the tagging of the query would be `<nan> <nan> Gold <nan> <nan> Nation Rank <num> <nan>`, with each token aligned with the word in the query in the same position.

When feeding into the seq2seq model, for each word in the original query, which is represented by GloVe pretrained word vector, we concatenate it with the vector of its tagged token, which uses the **same vocabulary and vector representation as the logical forms (decoder inputs)**; the intention is to give a better alignment between logical forms and the input query. We benefit this idea directly from the neural-attention-based copying technique adopted in Ref. [7]. Therefore, the length of each encoder input is the same as in vanilla model; however, the dimension of each input vector is doubled due to the tagging. For example, if the dimension of each word embedding is 100, then the dimension of each input vector in our tagging-based model is 200. (Fig. 4)

We design an algorithm to automatically output the tagging given the query and its corresponding field names. The current algorithm contain several steps: (0) initialize with all the tagging as `<nan>`; (1) identify the numerical values and tag with `<num>`; (2) compare string-by-string between the words in query and the field names, for word with high-similarity to a field name, we tag the word with that field name; (3) for each word in semantic space of the word vector (this step we represent the field names also in GloVe vectors) and tag with the nearest neighbors.

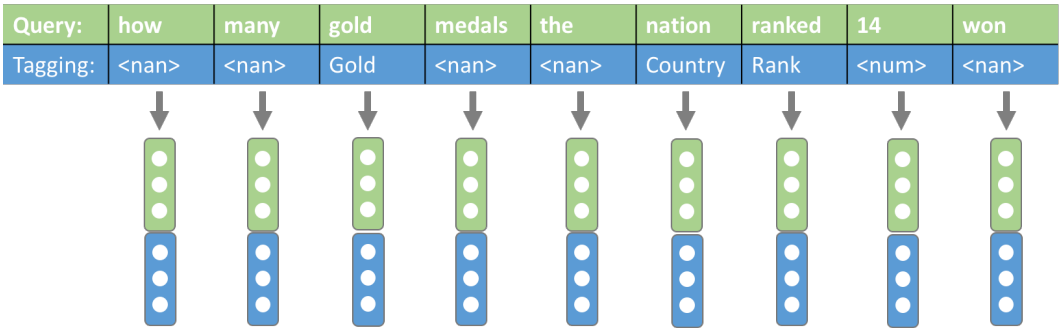

Figure 4: Tagging each input query with field information. Each word will be tagged with a token. Each word vector is concatenated with its tagged token when feeding into the seq2seq model.

3.3 Data Augmentation

In order to generate more data without decreasing the complexity of the natural language sentence, we basically use three methods to augment each query, which give us a $30 \sim 40$ times increment of the selected data.

(1) For each $(query, logical\_form)$ pair, based on the given logical form, we can change the field names in the form to similar but different ones. For example, the query `which nation won the most gold medals` has logical form `argmax Nation Gold` could be augmented to `argmax Nation Silver`, `argmax Nation Bronze`, and `argmax Nation Total`. (2) For each field name appearing in the logical forms, we could use different possible words that field could take in the corresponding query. For example, if we have the query `how many gold did Romania won` with corresponding logical form `where Nation equal Romania select Gold`, we can alter the value with respect to `Nation`, such as `where Nation equal India select Gold`, or `where Nation equal Spain select Gold`. (3) For each query, if there are words that could be replaced by its synonyms based on [8], it could generate several new queries with the same logical forms. For example, the query `which nation won the most gold medals` could be augmented with another query `which country won the most gold medals`, and both of the original and new ones correspond to the same logical form `argmax Nation Gold`.

4 Experiments

We compare our model against the previous system on the Wikitable dataset [1] we studied. We extend the rules for logical forms of the synthetic dataset, and label near 200 selected Wikitable queries to correct SQL-like logical forms manually on our own, and augment them to obtain 3711 $(query, logical\_form)$ pairs; we randomly divide them into a $3,000$ training set and 711 dev set.

We have described our data augmentation in previous section and present our experimental settings and results below. Finally, we conduct model analysis in order to understand what the model learns.

Our implementation and improvements are based upon the basic seq2seq model provided by Tensorflow framework [9]. We run our code on Microsoft Azure platform, which is equipped with Tesla M60 GPU capability.

4.1 Model Comparison

Our seq2seq model, which is based upon Ref. [9], has many hyper parameters that can affect the training efficiency. These hyper parameters include the number of layer, the unit size, the buckets classifying the sentence length, and the optimizer used for weights update. In this section, we focus on discussing the impacts of these hyper parameters on the training and evaluation of the data. The comparisons between different model parameters are also tabulated in detail.

4.1.1 Layer numbers & unit size tuning

The number of layers and the unit size control the depth and the width of a neural network, respectively. The product of them controls the overall trainable parameters that will be updated in back propagation. Given our data size, we considered several combinations of the number of layer and the unit size, to avoid over-fitting as well as under-fitting, we vary the number of layers from 2 to 4 and the unit size from 64 to 1024.

4.1.2 Bucket engineering

Our seq2seq model also makes use of **bucketing**, which is a method to efficiently handle sentences of different lengths for both natural language queries as $encoder\_inputs$ and SQL-like logical forms as $decoder\_inputs$. We have in total 4 different buckets to handle $(query, logical\_form)$ pairs with different lengths, and we carefully tuned each bucket size to make the number of data in each bucket does not vary too much with each other.

For the vanilla model (without tagging or field/entity concatenation) and tagging model, the best buckets we have got are with sizes $[(10, 8), (13, 10), (18, 13), (23, 21)]$; for model with single name entity concatenation, the best buckets we have got are with sizes $[(11, 8), (14, 10), (19, 13), (24, 21)]$; for model with all field names concatenation, the best buckets we have got are with sizes $[(14, 7), (19, 11), (24, 15), (30, 21)]$; for the synthetic dataset, the optimized buckets sizes are $[(10, 8), (17, 10), (26, 15), (40, 21)]$.

4.1.3 Optimizer

Stochastic gradient descent (sgd), which updates the parameters based on each example $x^{(i)}$ and label $y^{(i)}$, usually have fast convergence despite of its unpredictable fluctuations. Another different set of algorithms which depend on the momentum of the gradients, for example Adaptive Momentum Estimation (Adam), can take into account the change of gradient as the neural network evolves. Especially, Adam has the advantage of taking both the first order and the second order of momentum into consideration, thus significantly dampening the fluctuations that constantly occur in the case of sgd. Here, we consider both algorithms for our optimizer.

4.2 Last hidden state output

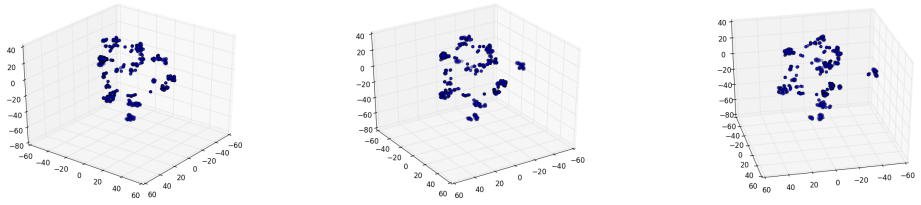

Figure 5: $t\text{-}sne$ visualization of the last hidden states of fully-trained encoder with one batch of natural language queries as encoder inputs. The hidden states are with dimension 1024, deducted to 3-dimension based on a PCA algorithm. The figures show the same .3-dimensional plot from different perspectives. As we can see the hidden vectors of the encoder inputs cluster to 16 centroids.

We output the last hidden state of encoder for each query (concatenate the states if there are multiple layers) and then use $t\text{-}sne$ algorithm (a PCA-based dimension deduction method) to plot in 3-dimensional space. All the vectors are clustering to 16 centroids, which represent 16 different logical forms we have.

4.3 Error Analysis

Here instead of comparing the results to the query directly, we analyze the accuracy of our model by comparing the logical forms. Based on the discussions in 4.1.1, 4.1.2 and 4.1.3, we consider the following 11 cases with different hyper parameters listed in Table. 1

Table 1: The 11 cases with different hyper parameters

	Optimizer	step size	# of iterations	# of layer	size of each layer
combination 1	SGD	0.05	15000	2	64
combination 2	Adam	0.0005	15000	2	64
combination 3	Adam	0.0005	15000	2	128
combination 4	Adam	0.0005	15000	2	256
combination 5	Adam	0.0005	15000	2	512
combination 6	Adam	0.0005	15000	2	1024
combination 7	Adam	0.0005	15000	3	64
combination 8	Adam	0.0005	15000	3	128
combination 9	Adam	0.0005	15000	3	256
combination 10	Adam	0.0005	15000	3	512
combination 11	Adam	0.0005	15000	4	64

For all the above 11 cases, the perplexities for the training data, and the 4 buckets of the dev data, for vanilla queries, queries prefixed with entity, queries prefixed with field, and queries with proper tagging, are summarized in Table. 2

Table 2: Table for perplexities for the train data and the 4 buckets of the dev data.

combination		1	2	3	4	5	6	7	8	9	10	11
vanilla	train	1.06	1.03	—	1.00	—	1.00	—	1.00	—	—	—
	dev bucket 0	1.11	1.15	—	1.28	—	2.50	—	3.50	—	—	—
	dev bucket 1	1.04	1.05	—	1.23	—	4.77	—	5.06	—	—	—
	dev bucket 2	1.03	1.30	—	1.14	—	1.20	—	5.63	—	—	—
	dev bucket 3	1.06	1.09	—	1.02	—	1.24	—	1.30	—	—	—
entity prefixed	train	1.00	1.04	1.00	1.00	1.02	1.00	1.00	—	1.00	—	—
	dev bucket 0	1.19	1.11	1.05	1.05	1.01	1.03	1.04	—	1.16	—	—
	dev bucket 1	1.12	1.04	1.01	1.05	1.00	1.02	1.04	—	1.09	—	—
	dev bucket 2	1.06	1.02	1.01	1.00	1.00	1.01	1.00	—	1.06	—	—
	dev bucket 3	1.02	1.01	1.02	1.02	1.04	1.04	1.02	—	1.01	—	—
field prefixed	train	1.06	1.00	1.00	1.00	1.02	—	1.00	1.01	1.01	—	1.01
	dev bucket 0	1.28	1.39	1.09	1.09	1.09	—	1.11	1.08	1.17	—	1.06
	dev bucket 1	1.14	1.08	1.04	1.03	1.03	—	1.05	1.03	1.07	—	1.09
	dev bucket 2	1.05	1.00	1.19	1.00	1.01	—	1.01	1.02	1.02	—	1.02
	dev bucket 3	1.11	1.02	1.01	1.02	1.03	—	1.01	1.05	1.03	—	1.02
tagging	train	—	1.01	1.01	1.00	1.00	1.00	—	—	1.00	1.00	—
	dev bucket 0	—	1.13	1.20	1.18	1.04	1.05	—	—	1.10	1.08	—
	dev bucket 1	—	1.04	1.02	1.03	1.02	1.02	—	—	1.03	1.01	—
	dev bucket 2	—	1.06	1.04	1.01	1.00	1.00	—	—	1.06	1.00	—
	dev bucket 3	—	1.02	1.01	1.03	1.03	1.01	—	—	1.02	1.02	—

As we can see from table. 2, for most cases, the perplexities for the train data and the four buckets in the dev data reach stable small values, which indicates the convergence of the training. Most of the perplexities are within the range of $(1.00, 1.10)$, and with a more complicated model, such as prefixing queries with fields or adding additional tagging, can further reduce the perplexities. For example, with an entity prefix, the perplexity of the train data can reduce to 1.00 as compare to 1.06 from the vanilla case for combination 1. Even though we also notice some differences in perplexities between the last three models, it is hard to tell the optimal model from the perplexities listed here. To better evaluate the accuracy of the above four models, we further show the accuracies for the train data and the dev data in Table. 3

Table 3 summarizes the training accuracy and the dev accuracy for the four models for all combinations listed in Table 1. The vanilla model, which does not include extra information about the table, is the least accurate. In contrast, the models using field as the prefix, which strongly enhances the connection between the query and the table, has a persistent high accuracy for almost all hyper-

Table 3: Table for accuracies of the train data and the dev data

combination		1	2	3	4	5	6	7	8	9	10	11
vanilla	train	0.390	0.512	—	0.516	—	0.556	—	0.524	—	—	—
	dev	0.361	0.464	—	0.459	—	0.323	—	0.310	—	—	—
entity prefixed	train	0.491	0.501	0.491	0.493	0.566	0.689	0.503	—	0.491	—	—
	dev	0.430	0.460	0.419	0.430	0.512	0.652	0.423	—	0.412	—	—
field prefixed	train	0.586	0.637	0.694	0.694	0.699	—	0.688	0.694	0.690	—	0.612
	dev	0.520	0.580	0.668	0.672	0.688	—	0.633	0.653	0.637	—	0.581
tagging	train	—	0.527	0.537	0.571	0.724	0.747	—	—	0.549	0.567	—
	dev	—	0.449	0.487	0.505	0.692	0.710	—	—	0.492	0.530	—

parameter combinations. In addition, for the last three models considered here, the training and dev accuracies have a significant enhancement when the hidden state size reaches a certain value. For example, the training accuracy for tagging model boosts up to 72.4% from 57.1% when the unit size changes from 256 to 512. This is because compared to a small hidden state size, a larger hidden state size is more accurate in covering the relations between a query, and the field the query refers to. In addition, adding more layers actually does not help in improving the accuracy. Finally, we highlight that in the case of combination 6, the training accuracy of tagging model, which tags each word in the query to its most related field, reaches 74.7% with a 71.0% dev accuracy, outperforming all the other models.

5 Conclusions & Future work

In this paper, we present a seq2seq neural translation machine for parsing natural language database query into SQL-like logical form, which could generically be used to access into a wide range of databases. To improve the model, we add field information into the original query by (1) concatenating all field names or single name entity directly in front of the query; (2) tagging each word in the query with the most related field name. After extensive comparisons with different hyperparameters on these models, we obtain the best model based on **query tagging of a 2-layer 1024-hidden-size network**, with training accuracy of 74.7% and development accuracy of 71.0% on the augmented dataset based on Wikitable, significantly higher than the previous studies [1, 5].

Our current augmented dataset is not exactly the real Wikitable queries; however, the SQL-like logical forms we generalize for now covers more than 80% of the natural language queries in Wikitable, so even considered the query types not included in the logical forms such as "True or False" and "next or previous", the accuracy our model could achieve is still much higher than previous works [1, 5]. So the next step of our research is to extent our logical forms to include the other 20% query types and directly test on the real Wikitable questions.

Regarding the model, we also propose a plenty of improvements we could work on in the future. In fact, we can further prefect our tagging model. Although the current tagger works well on identifying various related field names with near-zero false positive rate, there are some false negatives where the tagger fails to identify certain words to related field names. For example, in a query `what is the diff between gold medals won by spain and japan` we have the tag `<nan> <nan> <nan> diff <nan> Gold <nan> <nan> <nan> <nan> <nan>`, where both `spain` and `japan` are failed to be tagged with the field name `Nation`. In the future we will try to improve our tagger to better capture name entities and unseen words in the query.

Furthermore, for some of our incorrect logical forms generated, we realize that the neighbor tokens make sense, which means the decoder could well preserves the short range dependencies. Since our model is currently based on a greedy decoder, it does not necessarily generate the sentence with global maximum probability. Therefore, we plan to improve the decoder based on beam search during evaluation process, so the generated logical forms could be much closer to the global optimum, thus resulting in an improved accuracy.

Acknowledgments

We would like to thank Hongyu's summer intern manager, Haixun Wang from Facebook Search NLP team for project topic choice and guidance. We would also like to thank our CS224N TA, Danqi Chen for all the informative suggestions and useful discussions.

References

[1] Panupong Pasupat and Percy Liang. Compositional semantic parsing on semi-structured tables. *CoRR*, abs/1508.00305, 2015.

[2] Pengcheng Yin, Zhengdong Lu, Hang Li, and Ben Kao. Neural enquirer: Learning to query tables. *CoRR*, abs/1512.00965, 2015.

[3] Percy Liang, Michael I. Jordan, and Dan Klein. Learning dependency-based compositional semantics. *CoRR*, abs/1109.6841, 2011.

[4] James Clarke, Dan Goldwasser, Ming-Wei Chang, and Dan Roth. Driving semantic parsing from the world's response. In *Proceedings of the Fourteenth Conference on Computational Natural Language Learning*, CoNLL '10, pages 18–27, Stroudsburg, PA, USA, 2010. Association for Computational Linguistics.

[5] Arvind Neelakantan, Quoc V. Le, Martín Abadi, Andrew McCallum, and Dario Amodei. Learning a natural language interface with neural programmer. *CoRR*, abs/1611.08945, 2016.

[6] Li Dong and Mirella Lapata. Language to logical form with neural attention. *CoRR*, abs/1601.01280, 2016.

[7] Robin Jia and Percy Liang. Data recombination for neural semantic parsing. *CoRR*, abs/1606.03622, 2016.

[8] Xiang Zhang, Junbo Jake Zhao, and Yann LeCun. Character-level convolutional networks for text classification. In *NIPS*, 2015.

[9] Tensorflow seq2seq model.