Text-to-SQL Translation with Various Neural Networks
CS224N Project Final Report

Yipeng He yipenghe@stanford.edu
Di Bai dibai@stanford.edu
Wantong Jiang wantongj@stanford.edu

Abstract

Generating SQL queries from natural language has long been a popular and useful task attracting considerable interest. Most of the existing models are based on Long Short-Term Memory (LSTM) network with various attention mechanisms. Recently a new neural network architecture called Transformer has been proved to gain higher accuracy in neural machine translation scenarios. In addition, CNN has been proved to be efficient for text classification. In this project, we aim to explore different existing text-to-SQL models and analyze on them, as well as implement our model using Transformer and CNN. We evaluate how accuracy would be affected. Our experiments find that introducing syntactical structures of SQL to neural network can be helpful to text-to-SQL translation. Furthermore, for aggregator prediction, transformer and CNN encoders for questions could achieve comparable results to LSTM encoders.

1 Introduction

Generating SQL queries from questions in natural languages to help people easily retrieve data from databases has long been an interesting while challenging problem. A model on this task shall not only understand natural language questions but also generate corresponding SQL queries. With the emergence of various popular "natural language to SQL" datasets, such as WikiSQL [1] and Spider [2], many teams have contributed to solving this task. Seq2SQL [1] builds the WikiSQL dataset and proposes a sequence-to-sequence deep learning model on this dataset. They use reinforcement learning method to resolve order matters in SQL conditions. SQLNet [3] eliminates reinforcement learning by introducing dependencies on SQL structures and reflecting the dependencies via a specific attention mechanism. The state-of-the-art model, SyntaxSQLNet [4] is based on SQLNet but generates a more complex syntax tree which even supports recursive SQL queries. In this project we first explored and evaluated Seq2SQL and SQLNet on WikiSQL for inspirations on our model.

While most of the existing models are based on long short-term memory (LSTM) model [5], recently new models have been proposed and proved efficiency in some specific tasks. Transformer [6], a simple network only based on attention mechanism achieves highest performance in English-to-German and English-to French tasks. CNN [7] has also been proved to show excellence in text classification tasks. In this project, we aim to utilize these models on "natural language to SQL" problem, exploring how they perform in this scenario. With limited time and resources, we only focus on 'SELECT' clause in SQL grammar.

2 Related Work

Synthesizing SQL queries has long been a popular research topic. Seq2SQL [1] has been the first deep neural network based approach to solve this problem. They constructed WikiSQL, one of the largest

natural language query to SQL dataset and developed a deep neural network for translation. They translated SQL based on SQL structures and utilized reinforcement method to predict SQL conditions. They are the first to decompose SQL queries into different parts and predict each part to narrow the output space. Our model also utilizes this and predicts each part of SQL independently. Instead of "sequence-to-sequence" model, SQLNet [3] proposes a "sequence-to-set" model and eliminates reinforcement learning by constructing dependency relationship on SQL queries and developed a specific attention mechanism based on this dependency. They improved the prior state-of-the-art by 9 points to 13 points. Our model refers to their work on how they encode natural language questions together with column names in the WikiSQL dataset.

While most of the models for this task use LSTM network, nowadays, alternative networks are proposed to be effective and efficient in some specific tasks. Typically, Transformer [6], the first sequence transduction model based entirely on attention achieves state-of-the-art in both English-to-German and English-to-French problems as well as shows time efficiency. CNN [7] has been proved to be effective in text classification tasks. In our work, we introduce these two models on SQL translation problem. Our baseline model uses sequence-to-sequence neural machine translation with attention [8]. We simply view SQL as another language and translate English into SQL without any SQL specific optimization.

3 Approach

3.1 Baseline model

Text-to-SQL can be viewed as a language translation problem, so we implemented a LSTM [5] based neural machine translation model as our baseline. We implemented a bidirectional LSTM encoder and a unidirectional LSTM decoder. The model computes multiplicative attention using encoder hidden states and concatenate attention output with decoder hidden states. Pretrained GloVe [9] word embeddings is used and set fixed during training process. Note that the model has to deal with many non-vocabulary tokens (e.g., ID number and email address), the character-level model implemented by ourselves in assignment 5 is also used, and therefore the model can generate target words when the original decoder produces unknown token. The character embeddings are trained from scratch. No table information is used in this model, while the two models (seq2SQL and SQLNet) below takes table information in addition to the question as inputs.

For this part, we wrote the code ourselves and reused part of our own assignment 5 code.

3.2 Seq2SQL model

Seq2SQL [1] is the first deep neural network based approach to solve text-to-SQL translation problem. The model consists of three components, corresponding to three parts of a SQL query–aggregation operator, SELECT column, and WHERE clause. The model first extracts information of table, question and SQL vocabulary, concatenates them into an input sequence and encodes them by a two-layer bidirectional LSTM encoder. Then it uses different networks (LSTM + linear + softmax combination) to predict three different parts, using cross entropy loss for the first two and policy gradient for the last. It also uses attention as above to improve performance. It uses GloVe word embeddings and character n-gram embeddings [10] for the task. We fixed the embeddings during training.

For this part, we used the Seq2SQL source code (`https://github.com/xiaojunxu/SQLNet/blob/master/sqlnet/model/seq2sql.py`) but modified it to work with python 3.

3.3 SQLNet model

SQLNet [3] employs a more refined slot filling strategy. It takes advantage of the fact that the queries in the WikiSQL dataset can all be represented in the form below.

```
SELECT $AGG $SELECT_COL
WHERE $COND_COL $OP $COND_VAL
(AND $COND_COL $OP $COND_VAL)*
```

The model then predicts the column of SELECT clause ($SELECT_COL), the aggregator ($AGG), number of conditions (number of AND) and column ($COND_COL), operator ($OP) and value ($COND_VAL) for each condition using LSTM models with cross entropy loss. This model also uses column attention to improve performance. Different predictors may get different columns when predicting a specific slot and use attention mechanism to process the column information. For example, the aggregator predictor will get the ground truth $SELECT_COL as input in addition to the question embedding. The predictor then encodes the column name and applies attention to the encoded column. The column attention mechanism improved the performance by 3% on both dev set and test set [3]. The slot filling strategy has been further proved effective by TypeSQL [11].

The main difference between Seq2SQL and SQLNet is the generation of the WHERE clause. Seq2SQL generates the WHERE clause using a seq2seq strategy, and thus the order of the conditions matters. However, the condition orders of the WHERE clause actually doesn't matter with respect to the actual execution. Thus, SQLNet instead first predicts a set of columns and then predicts the $OP and $COND_VAL separately for the columns, so the order of these conditions doesn't matter anymore. Also, seq2SQL doesn't not provide additional column information to component predictors, while SQLNet does and apply column attention.

For this part, we used the SQLNet source code (`https://github.com/xiaojunxu/SQLNet/blob/master/sqlnet/model/sqlnet.py`) but modified it to work with python 3.

3.4 SQLNet with BERT word embedding

Bidirectional Encoder Representations from Transformers (BERT) [12] has proved to be effective in various natural language processing tasks. Therefore we decided to run SQLNet with BERT word embedding to see how a better word embedding affects the model performance. At first we tried to apply BERT word embeddings at training time but this makes the training extremely low. Therefore we preprocessed the training set with BERT word embeddings and produced a file recording all the word embeddings in a npy file. At training time, our model loads the npy file and maintains a dictionary with words as key and the word embedding array as value. Note that for a token that is in BERT's vocabulary, BERT produces a word vector with length 768. For an unseen token, BERT tokenizes it to tokens in BERT's vocabulary and provides embeddings on those tokens. For example, BERT doesn't have a token for "60-60", so it will tokenize "60-60" to "60", "-", and "60" and produce three 768 word vectors. In sum, for a unseen word, we took the average of the subtokens' word vector to give one word vector representation of the unseen word.

For this part, we used our modified python 3 compatible SQLNet code and BERT embedding package(`https://github.com/imgarylai/BERT-embedding`). We changed the extract vocabulary code to use BERT embedding. Before the decision to use preprocessed word embedding, we also experimented with running BERT in real time and integrated it into the SQLNet model. We wrote this part of the code ourselves.

3.5 $AGG and $SELECT_COL predictor

In this paper we explored the usage of transformer and CNN encoders for input questions and we used the output to predict the $AGG and $SELECT_COL. We used the $AGG and $SELECT_COL prediction pipeline as in SQLNet.

3.5.1 Transformer encoder

Transformer architecture has been proved effective in translation tasks [6]. The architecture aims to process sequential data with parallelizable computation methods, and thus uses the attention mechanism and no LSTM is involved. To inject positional information of the input, positional embedding is used in transformer. We experimented to use the transformer encoder architecture shown in figure 1 to replace the LSTM encoders for questions in SQLNet and evaluate its performance.

For this part, we used modules from an existing pytorch implementation (`https://github.com/jadore801120/attention-is-all-you-need-pytorch`) but modified it to work with BERT word embedding. We also wrote code to compute word to index vectors and position vectors for inputs to the encoder transformer.

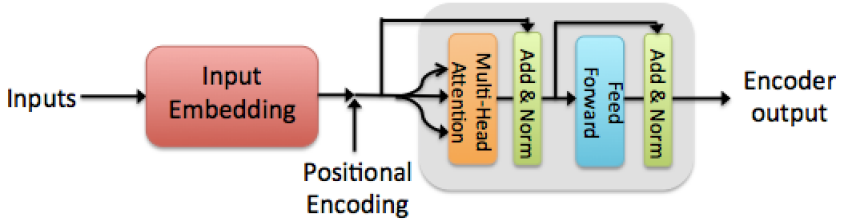

Figure 1: Transformer encoder structure

3.5.2 CNN encoder

Convolutional Neural Network (CNN) has been proved effective in text classification works [13]. We observed that the prediction of the aggregator is essentially a text classification task, and we thus also experimented using the convolutional network described in [13] as encoder, which is shown in figure 2 for questions for the aggregator prediction task. The basic idea is to process the input with varying kernel height but constant width as the word embedding length. The resulting tensors are concatenated and feed to a fully-connected layer to produce prediction score distribution.

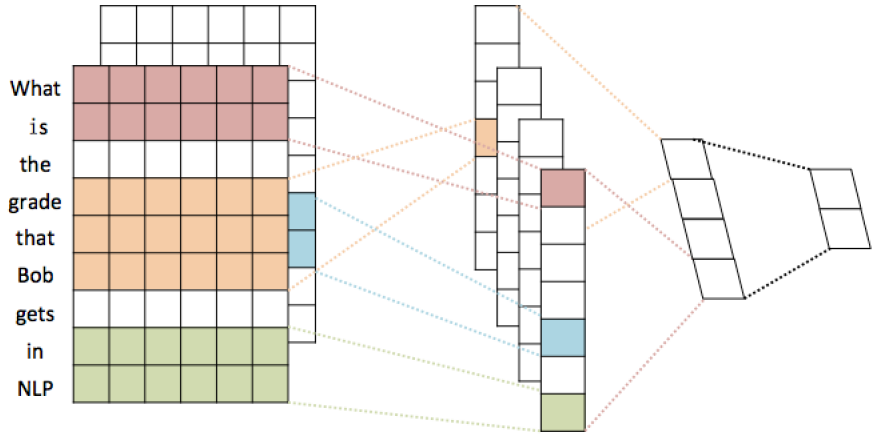

Figure 2: CNN encoder structure

For this part, we used modules from an existing pytorch implementation (`https://github.com/prakashpandey9/Text-Classification-Pytorch/blob/master/models/CNN.py`) but modified it to work with BERT word embedding.

3.5.3 $AGG predictions

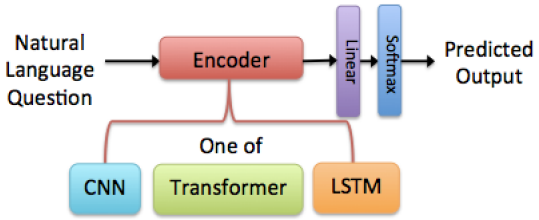

Figure 3: $AGG predictor structure.

We view predicting the $AGG slot as a 6-way classification problem. The possible values include MAX, MIN, COUNT, SUM, AVG and no aggregator. The input question word vector first goes through one of the three encoders (LSTM, transformer and CNN) to produce a hidden state embedding

as demonstrated in figure 3. The hidden state is then fed towards a fully-connected layer and a softmax layer to produce a probability distribution for the six aggregator types.

3.5.4 $SELECT_COL predictions

The $SELECT_COL predictor computes a probability distribution for the given column list. It assumes that the table is visible to the predictor and we need to make sure the column names are just correct so they are exposed to the model. It first encodes the question and the column names to word embeddings. Then the distribution is computed using the following equations. E_{col_i} and $E_{Q|col_i}$ represent the encoding of the column names and the input question. u_a, U_c and U_q represent trainable matrices and C represents total number of columns from the table associated with the input question. In this way, the possibility distribution of column to be selected is dependent both on the column names as well as the natural language question.

$$P_{selcol}(i|Q) = \mathbf{softmax}(sel)_i$$

$$sel_i = (u_a^{sel})^T \mathbf{tanh}(U_c^{sel} E_{col_i} + U_q^{sel} E_{Q|col_i}), \forall i \in \{1...C\}$$

We don't explore CNN in $SELECT_COL prediction because unlike $AGG, which has limited value range, $SELECT_COL can have various values depending on the table. So it's really not a classification problem but more like an information retrieval problem. We couldn't figure out an appropriate way to integrate column name information into the CNN architecture, so here we only introduce Transformer architecture.

4 Experiments

4.1 Data

The dataset this project uses is WikiSQL [1]. WikiSQL is a large scale hand-annotated natural language to SQL dataset. It consists of 80654 SQL queries extracted from 24241 HTML tables from Wikipedia.

WikiSQL dataset is built upon an assumption that each SQL query is only based on one table. As a result, each query only consists of SELECT and WHERE clauses. Here we show a typical query sample below.

Text query:

```
Which school did Herb Williams go to?
```

Corresponding SQL query:

```
SELECT school_name
WHERE  student_name = 'Herb Williams'
```

4.2 Evaluation method

Two main metrics are used to evaluate accuracy of our generated queries: query-match accuracy and execution accuracy.

Query-match accuracy: This refers to the percentage of matches between the ground truth queries and our generated queries. Instead of comparing the strings directly, queries are divided into several parts: the column in SELECT clause, the aggregator in SELECT clause, and each condition in 'WHERE' clause. Particularly, this measurement aims to minimize the negative effect of condition ordering. For instance, 'WHERE id = 1 AND course = NLP' should be equal to 'WHERE course = NLP AND id = 1'. We achieve this by transferring the WHERE clause into a set of conditions and comparing each element in the set.

Execution accuracy: This refers to the percentage of matched execution results between the ground truth queries and our generated queries. There may be different SQL queries that both correct for the same question. Thus we also use this accuracy to measure the correctness of our queries.

To evaluate our transformer encoders and CNN encoder, we evaluated the break down results of $AGG and $SELECT_COL predictions separately using exact match.

4.3 Experiment details

We used PyTorch [14] to implement this project.

- **Word embedding:** We used fixed pretrained GloVe word embeddings and BERT word embeddings.
- **Learning rate:** We used 10^{-3} for NMT baseline, 10^{-3} for Seq2SQL and 10^{-4} for SQLNet as in the original code.
- **Number of epoch:** We set the maximum training epoch as 100.
- **Optimization and regularization:** We used ADAM [15] to train to regularize.
- **Loss functions:** For NMT baseline, we used cross entropy loss between predicted result and ground truth as in assignment 4 and 5. For Seq2SQL and SQLNet, we used their original loss functions in [1] and [3].
- **$AGG and $SELECT_COL prediction**: We disabled the training of the WHERE clause when training aggregator and selection predictor. $AGG and $SELECT_COL were trained separately.
- **Transfromer**: We used a hidden size of 512, 3 layers and 4 multi-head attention modules.
- **CNN**: We used kernel heights of 1 3 5 for the three CNN layers, stride 1, padding 0 and output channel 64. We used a dropout rate 0.2 for the last linear layer.

4.4 Results

Table 1: Overall test results on the WikiSQL task.

Model	$\mathrm{Acc}_{\mathrm{qm}}$	$\mathrm{Acc}_{\mathrm{ex}}$
NMT (GloVe)	29.9%	–
Seq2SQL (GloVe)	49.8%	57.9%
SQLNet (GloVe)	58.4%	65.3%
SQLNet (BERT)	61.4%	67.8%
TypeSQL(GloVe) [11]	68%	74.5 %

$\mathrm{Acc}_{\mathrm{qm}}$ and $\mathrm{Acc}_{\mathrm{ex}}$ indicate query-match accuracy and execution accuracy respectively.

Table 1 suggests that changing the GloVe word embedding to BERT word embedding improves the performance of the SQLNet model by 3% in query match accuracy and 2.5% in execution match accuracy. This matches our expectation since BERT has been proved powerful on many natural language tasks. However, TypeSQL [11] also used GloVe word embedding and achieved a state-of-the-art accuracy. This result may indicate that a change in model structure may be more beneficial than further improving the word embedding.

The average execution accuracy is higher than query-match accuracy by 6 to 7%. This is expected because different SQL queries could have the same semantic meaning, and thus achieve same performance in realistic usage. Note that our test results of SQLNet (GloVe) are 3% lower than the results posted in SQLNet paper [3] because we trained for less epoches due to time limitation and we didn't set embeddings as trainable.

The baseline model performs relatively poor on this task because the task requires the model to perform semantic parsing to the question. The baseline model must learn to capture the structures of SQL queries itself. In contrast, the designs of Seq2SQL and SQLNet both take advantage of the structured pattern of SQL queries in WikiSQL dataset and decompose the task into three subtasks.

The significant performance improvement shows that we should continue to adapt this strategy when developing our models.

Table 2: Test results on $AGG prediction.

Model	train accuracy	dev accuracy	test accuracy
LSTM encoder	99.2%	89.2 %	90.0%
Transformer encoder	86.9%	86.6%	86.6%
CNN encoder	99.1%	86.4%	85.3%

Table 2 shows that the transformer and CNN encoder performed similarly on this task on the development data set and the test data set, but they both performed worse than the original LSTM encoder. However, the transformer performed slightly better than the CNN encoder. Notably, the LSTM and CNN were able to overfit the training dataset, while the transformer encoder didn't.

During the training and experiments, we observed that the transformer can be difficult to train. When we set the hidden size of the transformer to 1024 and 2048, the transformer barely made progress even on the aggregator prediction task. Only after we set the hidden size of transformer layers to be 512 did the transformer successfully made progress on the aggregator task.

We also tried to apply the transformer encoder to the selection clause prediction task, but the result isn't ideal. Table 3 shows that the accuracy of the transformer improved very little, which was from

Table 3: Test results on $SELECT_COL prediction.

Model	dev accuracy	test accuracy
LSTM encoder	89.6%	88.5%
Transformer encoder	19.6%	18.7%

around 18% to about 19%. The LSTM encoder still performs very well on this task. The transformer encoder modul structure is the same as the one we used for aggregator prediction. However, the aggregator transformer encoder was able to produce comparable results, while the transformer encoder for this task failed to make good progress. This may indicate that the same transformer encoder may not be able to directly replace bidirectional-LSTMs in different tasks, even if the inputs are similar. Due to the constraints of credits and time, we expect to do more in-depth study on this topic in the future.

5 Analysis

We would like to provide error analysis and insight in what errors the transformer encoder is making.

5.1 Column meaning error

Question: "How many tries against were there with 17 losses?"
Columns in table: "Club", "Played", "Drawn", "Lost",
"Points for", "Points against", "Tries for", "Tries against", "Try bonus", "Losing bonus"
Ground truth aggregator: No aggregator
Transformer encoder prediction: COUNT

Question: "What were the number of sales before 1991?"
Columns in table: "Year", "Álbum", "Charts", "Sales", "Certification"
Ground truth aggregator: No aggregator
Transformer encoder prediction: COUNT

The predictor predicted a COUNT aggregator for the above code while the ground truth is that the query requires no aggregator. The predictor may have learned to produce COUNT aggregator when there are phrases like "how many" and "the number of". However, in these two examples, the data in

column "Tries" and "Sales" already contains the total number of tries and the total number sales. This may indicate that the prediction system lacks understanding of meanings of column names. Column names like "Tries" and "Sales" implies the data of the column are summed data for a particular row.

5.2 Mix of SUM and COUNT

Question: "How many first games are associated with 5 games played and under 3 games lost?"
Columns in table: "First game", "Played", "Drawn", "Lost", "Percentage"
Ground truth aggregator: SUM
Transformer encoder prediction: COUNT

The predictor predicted COUNT where the ground truth is SUM. SUM and COUNT can be easily mixed with one another even for humans. Here if we use COUNT, it is counting the total rows that satisfies the condition instead of the total number of first games as requested in the query. This may indicate the system hasn't truly understood the usage of COUNT and SUM.

5.3 Transformer for $SELECT_COL prediction

The $SELECT_COL predictor with transformer performs particularly bad, so the analysis for particular examples may not make much sense. However, we would like to provide some of our insights on the training difficulties and the problem with the model.

Question: "What is the rank with 0 bronze?"
Columns in table: "Rank", "Nation", "Gold", "Silver", "Bronze", "Total"
Ground truth column: "Rank"
Ground truth column query: "SELECT sum(bronze) WHERE total GT 27"
Transformer encoder prediction: "Bronze"

Question: "What is the average Bronze for rank 3 and total is more than 8?"
Columns in table: "Rank", "Nation", "Gold", "Silver", "Bronze", "Total"
Ground truth column: "Bronze"
Ground truth column query: "SELECT avg(bronze) WHERE rank EQL 3 AND total GT 8"
Transformer encoder prediction: "Bronze"

Note that in these two examples, the ground truth columns "rank" and "bronze" are both inside the question. They should both be predicted in the final SQL query, but one in the SELECT clause, and one in the WHERE clause. This might indicate the transformer encoder lack the capability to break the question into SELECT clause part and WHERE clause part, while LSTMs can achieve this. Training transformer in limited time has been difficult to get progress. Even for the transformer encoder for aggregator, the loss sometimes fail to make progress due to bad initialization.

6 Conclusion

In this paper, we started out seeking improvements of the SQLNet model with different word embeddings. Our results show that BERT embeddings perform better compared with commonly used GloVe. We then tried to implement a model that utilizes architectures like Transformer and CNN to replace LSTM, and our results proved that these architectures can be used to predict the aggregator clause and get good accuracy. They have the potential to replace LSTM in text classification tasks like predicting $AGG. However, the transformer encoder was difficult to train when being applied to $SELECT_COL predictor. We expect to get better results by fine tuning parameters and longer-time training in the future.

7 Acknowledgements

We would like to thank Professor Mannings and Head TA Abigail See for their great lectures and instructions. We would like to thank Sahil Chopra for giving us great feedbacks and mentorships. We also would like to thank Microsoft Azure for providing generous Azure credits.

References

[1] Victor Zhong, Caiming Xiong, and Richard Socher. Seq2sql: Generating structured queries from natural language using reinforcement learning. *CoRR*, abs/1709.00103, 2017.

[2] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *arXiv preprint arXiv:1809.08887*, 2018.

[3] Xiaojun Xu, Chang Liu, and Dawn Song. Sqlnet: Generating structured queries from natural language without reinforcement learning. *arXiv preprint arXiv:1711.04436*, 2017.

[4] Tao Yu, Michihiro Yasunaga, Kai Yang, Rui Zhang, Dongxu Wang, Zifan Li, and Dragomir Radev. Syntaxsqlnet: Syntax tree networks for complex and cross-domaintext-to-sql task. *arXiv preprint arXiv:1810.05237*, 2018.

[5] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[6] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.

[7] Alexis Conneau, Holger Schwenk, Loïc Barrault, and Yann Lecun. Very deep convolutional networks for text classification. *arXiv preprint arXiv:1606.01781*, 2016.

[8] Minh-Thang Luong and Christopher D Manning. Achieving open vocabulary neural machine translation with hybrid word-character models. *arXiv preprint arXiv:1604.00788*, 2016.

[9] Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[10] Kazuma Hashimoto, Caiming Xiong, Yoshimasa Tsuruoka, and Richard Socher. A joint many-task model: Growing a neural network for multiple nlp tasks. *arXiv preprint arXiv:1611.01587*, 2016.

[11] Tao Yu, Zifan Li, Zilin Zhang, Rui Zhang, and Dragomir Radev. Typesql: Knowledge-based type-aware neural text-to-sql generation. *arXiv preprint arXiv:1804.09769*, 2018.

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[13] Yoon Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

[14] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[15] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.