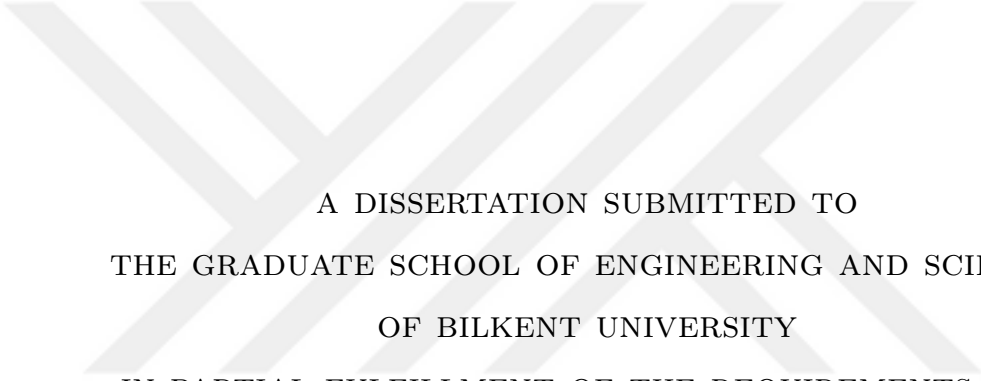


# TOWARDS DEEPLY INTELLIGENT INTERFACES IN RELATIONAL DATABASES



A DISSERTATION SUBMITTED TO  
THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE  
OF BILKENT UNIVERSITY  
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR  
THE DEGREE OF  
DOCTOR OF PHILOSOPHY  
IN  
COMPUTER ENGINEERING

By  
Arif Usta  
August 2021

Towards Deeply Intelligent Interfaces in Relational Databases

By Arif Usta

August 2021

We certify that we have read this dissertation and that in our opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

---

Özgür Ulusoy(Advisor)

---

İbrahim Körpeoğlu

---

Fazlı Can

---

İ. Sengör Altıngövde

---

Engin Demir

Approved for the Graduate School of Engineering and Science:

---

Ezhan Karaşan  
Director of the Graduate School

# ABSTRACT

## TOWARDS DEEPLY INTELLIGENT INTERFACES IN RELATIONAL DATABASES

Arif Usta

Ph.D. in Computer Engineering

Advisor: Özgür Ulusoy

August 2021

Relational databases is one of the most popular and broadly utilized infrastructures to store data in a structured fashion. In order to retrieve data, users have to phrase their information need in Structured Query Language (SQL). SQL is a powerfully expressive and flexible language, yet one has to know the schema underlying the database on which the query is issued and to be familiar with SQL syntax, which is not trivial for casual users. To this end, we propose two different strategies to provide more intelligent user interfaces to relational databases by utilizing deep learning techniques. As the first study, we propose a solution for keyword mapping in Natural Language Interfaces to Databases (NLIDB), which aims to translate Natural Language Queries (NLQs) to SQL. We define the keyword mapping problem as a sequence tagging problem, and propose a novel deep learning based supervised approach that utilizes part-of-speech (POS) tags of NLQs. Our proposed approach, called *DBTagger* (DataBase Tagger), is an end-to-end and schema independent solution. Query recommendation paradigm, a well-known strategy broadly utilized in Web search engines, is helpful to suggest queries of expert users to the casual users to help them with their information need. As the second study, we propose *Conquer*, a CONtextual QUery Recommendation algorithm on relational databases exploiting deep learning. First, we train local embeddings of a database using Graph Convolutional Networks to extract distributed representations of the tuples in latent space. We represent SQL queries with a semantic vector by averaging the embeddings of the tuples returned as a result of the query. We employ cosine similarity over the final representations of the queries to generate recommendations, as a *Witness-Based* approach. Our results show that in classification accuracy of database rows as an indicator for embedding quality, *Conquer* outperforms state-of-the-art techniques.

*Keywords:* Intelligent User Interfaces, Relational Databases, NLIDB, Keyword Mapping, Deep Learning, Graph Neural Networks, Query Recommendation.

## ÖZET

# İLİŞKİSEL VERİ TABANLARINDA DERİN AKILLI ARAYÜZLER ÜZERİNE

Arif Usta

Bilgisayar Mühendisliği, Doktora

Tez Danışmanı: Özgür Ulusoy

Ağustos 2021

İlişkisel veri tabanları, yapısal bir şekilde veri depolamayı sağlayan, en popüler ve yaygın olarak kullanılan alt yapılardan biridir. Veriye erişmek için, kullanıcılar ulaşmak istedikleri bilgileri Yapılandırılmış Sorgu Dili (SQL) kullanarak uygun sorgularla ifade etmek zorundadır. SQL oldukça anlatımcı ve esnek bir dildir; ancak kullanıcının, sorgunun yazıldığı veri tabanının temelinde yatan şemayı bilmesi ve SQL sözdizimine aşina olması gerekir, ki sıradan kullanıcılar için bu durum kolay değildir. Bu amaçla, derin öğrenme tekniklerinden yararlanarak ilişkisel veri tabanlarında kullanılmak üzere daha akıllı kullanıcı arayüzleri yapmak için iki farklı strateji önerilmektedir. İlk çalışmamızda, doğal dil sorgularını SQL'e tercüme etmeyi amaçlayan İlişkisel Veri Tabanına Yönelik Doğal Dil Arayüzleri'nde kullanılan anahtar sözcük eşleme problemi için çözüm sunulmaktadır. Anahtar sözcük eşleme, dizi etiketleme problemi olarak ele alınmaktadır. Doğal dil sorgularının sözcük türlerini de kullanan bu özgün yaklaşım, gözetimli derin öğrenme kullanılmaktadır. *DBTagger* (Database Tagger) adı verilen bu yaklaşım, uçtan uca ve şemadan bağımsız bir çözümdür. Web arama motorlarında yaygın olarak kullanılan ve oldukça bilinen bir strateji olan sorgu tavsiyesi paradigması, uzman kullanıcıların sorgularını sıradan kullanıcılara tavsiye ederek bilgiye erişim sürecinde yardım eder. İkinci çalışmamızda, derin öğrenmeden yararlanarak ilişkisel veri tabanlarında kullanılmak üzere bağlamsal sorgu tavsiye algoritması, *Conquer*, sunulmaktadır. İlk olarak, veri tabanı satırlarının gizli uzadaki temsillerini oluşturmak amacıyla Çizge Sinir Ağları kullanılmaktadır. SQL sorguları, sorgunun sonucunda dönen satırların temsillerinin ortalaması alınarak semantik vektör ile temsil edilmektedir. Tanık-Temelli bir yaklaşım olarak, tavsiyeler üretmek için sorguların nihai temsillerinin üzerine kosinüs benzerliği kullanılmaktadır.

*Anahtar sözcükler:* İlişkisel Veri Tabanlarında Doğal Dil Arayüzü, Anahtar Sözcük Etiketleme, Derin Öğrenme, Çizge Sinir Ağları, Sorgu Tavsiye.

## Acknowledgement

I would like to dedicate this thesis to my mother who had supported me throughout my education with immense love and care. I know that she would be proud had she been able to witness this.

First and foremost, I would like to express my deepest gratitude to my advisor, Prof. Dr. Özgür Ulusoy, for his invaluable guidance, motivation, and support throughout my academic journey.

I would like to thank Prof. Dr. İbrahim Körpeoğlu and Assoc. Prof. Dr. İ. Sengör Altıngövdde for monitoring my thesis studies and giving me constructive feedbacks during our meetings. I would also like to thank Prof. Dr. Fazlı Can and Asst. Prof. Dr. Engin Demir for kindly accepting to be in my thesis committee.

I would also like to acknowledge TUBİTAK for supporting me financially for this thesis under grant number 118E724.

I would like to take this opportunity to thank my dearest friends Fuat, Taylan, Hasan, and Mustafa with whom I have shared a lot during my doctoral studies. I really appreciate their support and understanding at my difficult times.

I feel myself lucky to have such a loving family, who always supported my dream of being a scientist since primary school. I can not thank enough my sister, Duygu, for taking care of our dearest father during my studies.

Last but not least, I would like to thank my lovely wife, Hazal, for making me believe in myself, helping me to regain my motivation with her endless cheer and support. None of this would have been possible without her.

Part of this work is reprinted with permission; based on the work, Arif Usta, Akifhan Karakayali, and Özgür Ulusoy. 2021. DBTagger: multi-task learning for keyword mapping in NLIDBs using Bi-directional recurrent neural networks. Proc. VLDB Endow. 14, 5 (January 2021), 813–821. DOI:<https://doi.org/10.14778/3446095.3446103>

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation and Contributions . . . . .	2
1.2	Outline . . . . .	7
<b>2</b>	<b>Related Work</b>	<b>9</b>
2.1	Keyword Mapping in NLIDB . . . . .	9
2.1.1	NLIDBs and Keyword Mapping Approaches . . . . .	9
2.1.2	Deep Learning Solutions for Sequence Tagging in NLP . . . . .	13
2.2	SQL Query Recommendation in Databases . . . . .	13
2.2.1	Distributed Representations in Databases . . . . .	13
2.2.2	Query Recommendation in Databases . . . . .	15
<b>3</b>	<b>Distributed Representations in Deep Learning</b>	<b>18</b>
3.1	Word Embeddings . . . . .	18
3.2	Character Embeddings . . . . .	21
3.3	Sentence Level Representations . . . . .	21
3.4	Transformers . . . . .	24
3.5	Graph Neural Networks . . . . .	29
<b>4</b>	<b>Keyword Mapping in NLIDB (DBTagger)</b>	<b>34</b>
4.1	Deep Sequence Tagger Architecture . . . . .	38
4.2	DBTagger Architecture . . . . .	41
4.3	Annotation Scheme . . . . .	45
4.3.1	POS Tags . . . . .	45
4.3.2	Type Tags . . . . .	46
4.3.3	Schema Tag . . . . .	47

<b>5</b>	<b>SQL Query Recommendation in Databases (Conquer)</b>	<b>49</b>
5.1	Embedding Relational Database into Latent Space Using Graph Neural Networks . . . . .	49
5.1.1	Classifying Tables as Entity or Relation in Databases . . .	52
5.1.2	Tuple-Tuple-Value (TTV) Graph . . . . .	52
5.1.3	Tuple-Tuple (TT) Graph . . . . .	54
5.1.4	Node Embeddings with Graph Convolutional Networks . .	55
5.1.5	Hybrid Approach: Combining Graph Neural Networks with External Deep Learning Techniques . . . . .	57
5.2	SQL Query Recommendation Based on Tuple Representations . .	59
<b>6</b>	<b>Experimental Evaluation</b>	<b>63</b>
6.1	Keyword Mapping in NLIDB . . . . .	63
6.1.1	Datasets . . . . .	63
6.1.2	Experimental Setup . . . . .	65
6.1.3	Comparison with Unsupervised Baselines . . . . .	68
6.1.4	Translation Accuracy . . . . .	71
6.1.5	Impact of DBTagger Architecture . . . . .	73
6.1.6	Efficiency Analysis . . . . .	75
6.2	SQL Query Recommendation in Databases . . . . .	77
6.2.1	Datasets . . . . .	77
6.2.2	Experimental Setup . . . . .	78
6.2.3	Tuple Classification . . . . .	79
6.2.4	t-SNE Visualization of Database Tuples . . . . .	82
6.2.5	Generated Recommendations . . . . .	82
<b>7</b>	<b>Conclusion</b>	<b>88</b>

# List of Figures

3.1	Neural network architecture for word2vec approaches . . . . .	19
3.2	Neural network architecture of Embeddings from Language Models (ELMo) . . . . .	20
3.3	High level view of sequence to sequence neural network architecture (Encoder - Decoder) . . . . .	22
3.4	Architecture of RNN based Encoder - Decoder solution for machine translation problem . . . . .	23
3.5	High level architecture of sentence representation as a prediction problem (Skip-Thoughts) . . . . .	24
3.6	Neural network architecture for transformers . . . . .	26
3.7	Neural network architecture of self-attention mechanism in transformers . . . . .	27
3.8	Sample random walk on graph and its reflection in latent space .	30
3.9	Information aggregation by convolution in graph through neighboring nodes . . . . .	31
3.10	Batch of input graphs for the example graph given in Figure 3.9 .	32
4.1	ER diagram of a subset of IMDB movie database . . . . .	35
4.2	Recurrent Neural Network (RNN) architecture . . . . .	38
4.3	Architectures of Long-Short Time Memory (LSTM) and Gated Recurrent Units (GRU) . . . . .	39
4.4	Architecture of deep sequence tagger neural network . . . . .	40
4.5	Architecture of DBTagger Network . . . . .	42
4.6	Hard parameter sharing in multi-task learning . . . . .	43
5.1	A sample ER Diagram for Scholar Schema . . . . .	50



5.2	Example instances for Scholar Schema . . . . .	51
5.3	Example TTV graph extracted from sample database instances in Figure 5.2 . . . . .	54
5.4	Sample TT graph extracted from sample database instances in Figure 5.2 . . . . .	55
5.5	Flowchart diagram of the recommendation algorithm, Conquer . .	61
6.1	Run time comparison of DBTagger with unsupervised state-of-the- art keyword mapping approaches . . . . .	75
6.2	Memory usage comparison of DBTagger with unsupervised state- of-the-art keyword mapping approaches . . . . .	76
6.3	Graph Convolutional Network Model Architecture . . . . .	79
6.4	tSNE visualization of the models in the geography dataset. For each baseline, plots are depicted side-by-side. From top to bottom, each row of plots represents word2vec, doc2vec, fastText, and USE baselines, respectively . . . . .	83
6.5	tSNE visualization of the models in the college dataset. For each baseline, plots are depicted side-by-side. From top to bottom, each row of plots represents word2vec, doc2vec, fastText, and USE baselines, respectively . . . . .	84
6.6	Visualization of the queries along with accessed tuples for geogra- phy dataset . . . . .	86
6.7	Visualization of the queries along with accessed tuples for college dataset . . . . .	86

# List of Tables

4.1	An example natural language query with its associated tags corresponding to each word in three different levels . . . . .	45
5.1	Author Table . . . . .	51
5.2	Paper Table . . . . .	51
5.3	Venue Table . . . . .	51
5.4	Writes Table . . . . .	51
5.5	Sample tuple sentences used for textual deep learning for node features . . . . .	58
6.1	Statistics of the public relational databases used for keyword mapping problem in NLIDBs . . . . .	64
6.2	Statistics of the schemas used from the Spider dataset for keyword mapping problem in NLIDBs . . . . .	64
6.3	Accuracy results of neural models with different task weights for fine tuning . . . . .	65
6.4	F1 results of neural models with different task weights for fine tuning . . . . .	66
6.5	Accuracy scores of unsupervised baselines for relation and non-relation matching . . . . .	67
6.6	Comparison of Pipeline utilizing tags output by DBTagger with state-of-the-art translation solutions . . . . .	72
6.7	Performance of Neural Models with Different Architectures in accuracy-F1 metrics for public databases . . . . .	74
6.8	Performance of Neural Models with Different Architectures in accuracy-F1 metrics for schemas in the Spider dataset . . . . .	74

6.9 Statistics of the public datasets used for the query recommendation  
algorithm . . . . . 78

6.10 Accuracy-F1 results of tuple classification using tuple embeddings  
for datasets along with baselines . . . . . 81

6.11 Example pairs of most similar queries from both datasets . . . . . 87



# Chapter 1

## Introduction

Amount of processed data has been growing rapidly pertaining to technology, leading to database systems to have a great deal of importance in today's world. Amongst the systems, relational databases are still one of the most popular infrastructures to effectively store data in a structured fashion. To extract data out of a relational database, *structured query language (SQL)* is used as a standard tool. Although SQL is a powerfully expressive language, even technically skilled users have difficulties using SQL. Along with the syntax of SQL, one has to know the schema underlying the database upon which the query is issued, which further causes hurdles to use SQL. Consequently, casual users find it even more difficult to express their information need, which makes SQL less desirable.

To remove this barrier, an ideal solution is to provide a search engine like interface, such as Google or Bing in databases. One of the reasons why search engines are popular among casual users in various domains such as e-commerce or education [1, 2, 3, 4] is that search engines have user friendly interfaces providing an easy-to-use keyword-based search functionality [5]. Although, keyword-based search is a simple interaction mechanism between the search engine interface and the user, such an interface still can be inadequate for certain users to express their information needs. In order to obviate such inefficacy, researchers have introduced paradigms such as query recommendation or auto-completion [5] to

make search interfaces more intelligent to serve the user more conveniently.

Deep Learning is one of the paradigms to employ to further improve user friendliness and therefore consequently search performance of retrieval systems in various domains such as biomedical [6], image [7] or education [8]. Although it is demonstrated to be effective in many tasks, relational databases as a retrieval system has been overlooked to apply deep learning to enhance search performance of users. In order to try to obviate lack of such studies, in this thesis we propose two different solutions exploiting deep learning techniques to help create intelligent interfaces to relational databases.

## 1.1 Motivation and Contributions

**The first study** we propose towards creating intelligent interfaces to relational databases is in the context of *Natural Language Interfaces to Databases (NLIDB)*. Natural language is the ideal choice of users when they are expressing their search needs. Hence, having a natural language query interface is the first step to make towards providing user friendly environment to the users for their search experience. Therefore the goal of NLIDB systems is to break barriers mentioned above between the database and the user to make it possible for even casual users to employ their natural language to extract information.

To this end, many works have been published recently attacking the research problem of translation of natural language queries into SQL; such as conventional pipeline based approaches [9, 10, 11, 12] or end-to-end solutions using encoder-decoder based deep learning approaches [13, 14, 15, 16, 17]. Neural network based solutions seem promising in terms of robustness, covering semantic variations of queries. However, they struggle for queries requiring translation of complex SQL queries, such as aggregation and nested queries, especially if they include multiple tables. They also have a huge drawback in that they need many SQL-NL pairs for training to perform well, which makes conventional pipeline based solutions still an attractive alternative. [18].

In the translation pipeline, one of the most important sub-problems is *keyword mapping*, as noted in [19] as an open challenge to be addressed in NLIDBs. *Keyword mapping* task requires to associate each token or a series of consecutive tokens (e.g., keywords) in the natural language query to a corresponding database schema element such as table, attribute or value. It is the very first step of resolving ambiguity for translation. Xu et. al [15] also note that during the translation of the query, *where* clause is the most difficult part to generate which further signifies the task of *keyword mapping*.

Most of the pipeline-based state-of-the-art works do not provide a novel solution to the problem of *keyword mapping*, rather they utilize unsupervised approaches such as simple look-up tables looking for exact matches or pre-defined synonyms [20, 9]; or they make use of an existing lexical database [10] such as WordNet [21]; or they exploit domain information to extract an ontology to be used for the task [11]; or they employ distributed representations of words [12] such as word2vec [22] to calculate semantic similarity of tokens over database elements. Although these approaches are effective to some extent, they fail to solve various challenges yielded by the task of *keyword mapping* single-handedly.

In order to address the sub-problem of *keyword mapping* solely, two particular works have been published recently. In [23], Yavuz et. al. try to improve translation accuracy by focusing on condition statements in *where* clause only. In the study, n-grams of words in the NL query are extracted and queried against database schema elements such as attribute names. They evaluate their approach on well known WikiSql dataset [14]. However, the dataset itself is comprised of databases having only a single table which makes it difficult to assess validity of their approach in more complex databases. Their methodology of utilizing n-grams of words in the query to associate possible *value* is also not efficient, especially in online scenarios.

Baik et. al. [24] present *TEMPLAR*, another approach that deals with the *keyword mapping* directly along with join-path inference. They exploit query logs of databases to improve accuracy of mapping. Yet, their approach is not a standalone solution, requiring an existing NLIDB to generate certain metadata

annotations including multi-word entities, conditions, and operators correctly to be used in mapping. They try to rank similarities between these metadata annotations coming from NLIDB and candidate query-fragments extracted from query logs using word2vec [22], which limits the solution to be dependent to a well performing existing NLIDBs.

In order to address all of the challenges the keyword mapping problem raises we propose **DBTagger**, a novel deep sequence tagger architecture used for *keyword mapping* in NLIDBs. Our approach is applicable to different database domains requiring only handful of training query annotations and practical to be deployed in online scenarios finding tags in just milliseconds. In particular, we make the following contributions by proposing DBTagger:

- We tackle the keyword mapping problem as a sequence tagging problem and borrow state-of-the-art deep learning approaches tailored for well-known NLP tasks.
- We extend the neural structure for sequence tagging, by utilizing *multi-task learning* and *cross-skip connections* to exploit the observation we made in natural language query logs of databases, that is, schema tags of keywords are highly correlated with part-of-speech (POS) tags.
- We manually annotate query logs from three publicly available relational databases, and five different schemas belonging to Spider [25] dataset.
- We evaluate DBTagger, with above-mentioned query logs in two different setups. First, we compare DBTagger with unsupervised baselines preferred in state-of-the-art NLIDBs. In the latter, we evaluate DBTagger architecture by comparing with different supervised neural architectures. We report new state-of-the-art accuracy results for keyword mapping in all datasets.
- We provide comprehensive run time and memory usage analysis over the existing keyword mapping approaches. Our results show that, DBTagger is the most efficient and scalable approach for both metrics.

While expressing their information need, users may prefer different phrases, which may depend on their level of expertise and the schema of the underlying database. In order to pose effective queries in relational databases, users may need to be familiar with SQL syntax and the domain of the database, which is not always the case. Users may have little information about the schema underlying the database or they may not be certain about what to search for. Therefore, it is vital to have a user friendly interface for retrieval systems such as relational databases. Although, keyword search provides flexible use of search expression for users, one may not be able to express his search intent in correct manner using proper words. Therefore, Query Recommendation, as one of the well-known paradigms utilized to improve users' search experience especially in Web [5], can be handy. In order to address problems mentioned above, **as a second study**, we propose a SQL query recommendation algorithm in relational databases to make the search interface more user friendly.

Although multiple users have the same search intent, they may use different words to express their intent, which causes ambiguity. Another issue, specifically valid in relational databases, is that user may not be an expert and not have enough technical background to express search intent in words. In order to overcome the issue of lack of ability expressing the information need, most of the search engines provide query recommendation service to further improve search experience of users. The main objective behind our approach is to recommend previous SQL queries to the user given the current SQL query. It aims to find similar queries that are previously issued by some other users. In order to provide a recommendation, a similarity function is to be determined between queries first. After finding those candidate queries, a ranking scheme is applied on those queries to create final list of suggested queries to the user.

In recent years, neural network based vector representations of textual data have become the state-of-the-art technique to exploit for solving various NLP tasks. It is demonstrated by Collobert et. al. [26] that utilizing dense vector representations in a simple deep learning architecture yields superior results in several NLP tasks such as named-entity recognition (NER), semantic role labeling (SRL) and POS tagging. Distributional hypothesis [27], which assumes



words appearing in similar contexts tend to have similar meaning, played vital part in success of learning representations of words. Mikolov et. al. [22] utilize this assumption to learn word embeddings in an unsupervised fashion by introducing two different approaches, namely skip-gram and common bag-of-words (CBOW). In addition to word level representations, character embeddings [28, 29] and different composition level of words [30] have also been explored for different downstream NLP tasks such as sentiment classification. Learning representations of sentences have various application scenarios such as machine translation, next sentence prediction, and so on. Inspired by the skip-gram approach, Encoder-Decoder approaches have been utilized in recent works [31, 32] to learn meaningful representation of the sentences for such tasks.

Though distributed representation of words have been explored in several different setups with different neural architectures and objectives for general text, there are not many works on learning representations for textual values in a different source such as database. Relational databases is one of the most common infrastructure to store data (mainly text) in a structured fashion. As noted in [33], deep learning approaches especially distributed representations can be effective for multiple problems in database community where a similarity objective is required to be defined such as entity resolution, query recommendation, or query clustering.

Inspired from the above observation, we propose **Conquer**, a recommendation algorithm that exploits distributed representations trained solely on a relational database. In our approach, we first train a neural model utilizing graph convolutional networks (GCN) [34] to learn local embeddings for database tuples. After learning representations, we calculate an aggregated representation for particular SQL query by using representations of the tuples returned in the resulting set. For candidate queries, we perform a cosine similarity to find out possible SQL queries to recommend. In particular we make the following contributions:

- We propose a SQL query recommendation algorithm utilizing a well-known deep learning technique, graph convolutional networks. To the best of our knowledge, our study is the first to employ deep learning in SQL query

recommendation system in relational databases.

- We automatically extract two different graph outputs from a relational database regarding the database properties to use in training with GCN.
- Compared to another study [35] in which an output graph is extracted from a relational database, we perform our training in supervised fashion.
- In order to employ training in supervised fashion, we propose a heuristic to associate each tuple with a target label as an application of *self-supervised learning*.
- We train node embeddings of database tuples along with textual values residing in the database utilizing GCN. Consequently, we learn embeddings for both tuples and words appearing in those tuples simultaneously.
- In addition to a standalone training with graph neural networks, we propose a hybrid approach in which we externally train features of node in the graph by state-of-the-art distributed representation techniques utilized in natural language processing tasks as an application of *transfer learning*. We utilize these externally trained representations for node features in the input graph which is later trained further with GCN.
- We evaluate our solution in 2 different datasets, in which we outperform state-of-the-art baselines in the task of classification of database tuples in both accuracy and F1 metrics.

## 1.2 Outline

The rest of the thesis is organized as follows. In the next section, we summarize the related studies in the literature by categorizing them into the fields they fall into. In Chapter 3, we provide a thorough explanation of current state-of-the-art distributed representation techniques in deep learning. We give methodology of our first proposed work, ***DBTagger***, in Chapter 4. Similarly, in Chapter 5, we give detailed information about our second proposed work, ***Conquer***. Following

these chapters, we provide experimental results of each proposed work in Chapter 6. We conclude the thesis in Chapter 7 by pointing out potential future works to be studied.



# Chapter 2

## Related Work

### 2.1 Keyword Mapping in NLIDB

In this section, we cover the related works mainly in two categories. In the first group, we provide a literature review about recent NLIDB systems and their keyword mapping approaches in the translation pipeline. In the latter, we mention the works that use deep learning to solve mainstream sequence tagging problems, such as part-of-speech (POS) and named-entity recognition (NER) in NLP.

#### 2.1.1 NLIDBs and Keyword Mapping Approaches

Although the very first effort [36] of providing natural language interface in databases dates back to multiple decades ago, the popularity of the problem has increased due to some recent pipeline based systems proposed by the database community, such as SODA [9], NALIR [10], ATHENA[11] and SQLizer[12]. SODA [9] provides a natural language interface in which users can express their query intent in keywords. In their proposed system, they use look-up tables, namely inverted indexes for both base and metadata of the database to match

query tokens into their respected tags. The main disadvantage of SODA is that it only expects keywords, which are not enough to express query in certain cases.

NALIR [10] is a parse-based NLIDB system. The system first parses the input using Stanford Parser [37] and extracts dependency tree solely based on it. In the linguistic parse tree, nodes represents any words or phrases that can be mapped to SQL instruments. It tries to map keywords into candidate database schema elements using two different similarity metrics, similarity in meaning and spelling, respectively. They use Wordnet [21] to evaluate similarity in meaning (semantic) and use Jaccard Coefficient between q-grams of words/phrases and candidates. Using a threshold  $\tau$ , they determine whether the processed schema element is a candidate. During the mapping, NALIR interacts with the user to solve ambiguities, which is the strength of the system.

NALIR heavily depends on the user interaction step to resolve ambiguities during mapping, which dramatically affects translation accuracy. Athena [11] is another pipeline-based system which uses domain knowledge, ontology, to make the translation. Given a relational database on a particular domain, Athena needs ontology of the domain to be extracted first. As a first step, ontology evidence annotator is used to map keywords in the query to possible ontology elements. This process is done using inverted index generated on database metadata and unique values. On top of the inverted index, they also enrich the index with synonyms and semantic variations of the metadata.

SQLizer [12] uses a semantic parser to extract a query sketch from the natural language query, and iteratively repairs the sketch until a confidence threshold is met. Output of the semantic parser includes mapped hints from the natural language query words as n-grams. This mapping is not a complete mapping; database elements are not included in the initial sketch, therefore mapping includes only natural language query tokens. Mapping is updated in each iteration and completed at the end. One important weakness of these systems is that they rely on manually defined rules for the translation.

Recently, end-to-end approaches utilizing encoder-decoder based architectures

[14, 15, 38, 23, 39, 40, 16, 41, 42, 17] in deep learning have become more popular to deal with the translation problem. In such a setup, the problem becomes a machine translation where NLQ-SQL pairs are used.

Seq2SQL[14] uses a Bi-LSTM to encode a sequence that contains columns of the related table, SQL keywords and question. It outputs three different components; aggregation classifier, select column pointer, and where clause components. One of the problems with this approach is that there is only one output for the select column pointer. The study [14] also provides a dataset called *WikiSql* to the research community working on NLIDB problem for evaluation.

WikiSql is comprised of 26,531 tables and 80,654 pairs which can be used for input for the translation problem. However, the dataset includes databases having only a single table which simplifies the problem especially for the translation. SQLNet[15] defines a seq-to-set approach to eliminate reinforcement learning process of Seq2SQL. Network outputs a score for each column and a thresholding is applied to decide which columns will be in the where clause set. The output structure of SQLNet is similar to Seq2SQL and it has the same weaknesses.

In another study which used WikiSql dataset, Yavuz et al.[23] employ a process called candidate generation to create keyword mappings to be used in *where* clause in SQL translation specifically. They first create all the n-grams from the question. Then, using each n-gram they query each column of the given table to find the matches. Finally they create a mapping set for each column and n-gram match. The solution proposed is not scalable for bigger databases and not practical for online usage, since the number of n-grams to be used for mapping candidates increases with the number of tokens in the query, which makes the whole process time consuming.

All of the proposed deep learning based methods use pre-trained word embedding vectors for input to the model. Therefore, keyword mapping is implicitly handled by the model. However, TypeSql [39] tries to enrich input data augmenting entity tags by performing similarity check over the database or knowledge base. They try to embed these tags during training, and feed concatenation of

both these embeddings and word embeddings into the network as input. Similarly, [40] tries to find possible constant values in the query by performing similarity matching.

Due to the limited nature of WikiSql dataset, having a single table for each database, another important dataset called Spider [25] is provided to the community. Consequently, many studies proposed recently [16, 41, 42, 17, 43] have evaluated their solutions on the Spider dataset. Different from the others, TaBERT [43] as a transformer based encoder, makes use of database content to generate dynamic representations along with contextual encodings to represent database columns. For a comprehensive survey covering existing solutions in NLIDB, the reader can refer to [44, 45].

Similar to our work, Baik et al. [24] propose TEMPLAR, to be augmented on top of existing NLIDB systems to improve keyword mapping and therefore translation using query logs. It uses the query logs to identify database fragments that later are to be mapped to query keywords for database value matching. For matching candidate fragments extracted from query logs, a semantic similarity model using word2vec [22] is employed. Though, TEMPLAR is not a standalone mapper, since it requires from a NLIDB system multiple preliminaries to function properly, including parsed keywords and associated metadata with each keyword, which are the main challenges yielded by the keyword mapping problem.

TEMPLAR requires the NLIDB system to parse the keywords that may contain multiple words, which is one of the major challenges for keyword mapping. Therefore, the mapper cannot be plugged into NLIDB pipelines that does not perform detailed keyword recognition and parsing.

Different from the previous works, DBTagger is an end-to-end keyword mapper, which does not require any processing or external source of knowledge. Also, to the best of our knowledge, our work is the first study utilizing deep neural networks in a supervised learning setup for keyword mapping.

### **2.1.2 Deep Learning Solutions for Sequence Tagging in NLP**

In NLP community, neural network architectures have been utilized in many research problems. As a pioneer in the field, Collobert et al. [46] propose Convolutional Neural Networks (CNN) based architecture with CRF layer on top to deal with the sequence tagging problem. Yao et al. [47] apply LSTM in sequence tagging without having CRF as the classification layer. Bi-directional RNN structure is employed first in a speech recognition problem in [48].

Later, instead of simple RNN networks, bi-directional LSTM is adopted and employed by Huang et al. [49] in NER problem. They use hand crafted spelling features for representation of the words in the sentence as the input layer. Following that study, Lample et al. [50] propose a similar architecture with the inclusion of word and character embeddings. They use pre-trained word embeddings along with character level embeddings to extract input matrix to feed into the network. They also train character level embeddings during the training utilizing LSTM. Their study stand as the state-of-the-art in sequence tagging problems in NLP.

Similar to [50], Ma and Hovy [51] propose a neural architecture where character embeddings is done through CNN instead of LSTM. For a comprehensive survey discussing the deep learning solutions for research problems in NLP community, [52] is a great read.

## **2.2 SQL Query Recommendation in Databases**

### **2.2.1 Distributed Representations in Databases**

Recently, distributed representations of words have been utilized in database problems such as entity resolution [53, 54, 35, 55] and schema matching [56]. Mainly, the works utilizing distributed representations in database community



fall into two broad categories; in the first group, pre-trained word embeddings such as word2vec [22], glove [57] or fastText [58] are used in the appropriate task as an input without any training, whereas the approaches in the second group train their local database embeddings to exploit the corresponding task to overcome disadvantages of global pre-trained embeddings.

The first work which tries to integrate word embeddings into the database field is done by Bordawekar et. al. [59]. They learn word level representations using word2vec [22] by treating each row as a sentence composing of values, both textual and numeric. There are couple of drawbacks with this approach. They learn only word level representations which do not cover plurality of words, typos or out-of-vocabulary tokens which is a common problem in bigger databases. Their sentence level composition is bag-of-words approach which does not take the relational part of the database into account such as connection between pair of instances. Although they consider these connections through foreign keys by expanding rows into bigger sentences, window size for similarity context is not enough to capture similarity between values residing in different tables. They exploit these representations in SQL-based analytics queries that they named as cognitive intelligence.

For entity matching task, both [53] and [54] use recurrent neural networks to train local database embeddings. The biggest difference between the two state-of-the-art solutions is that [54] also utilizes attention mechanism [60] to improve training efficacy and therefore representations. In these works, tuples are treated as sentences in which cell values represent words in these tuples. However, similar to work [59], these studies also do not take the structural information of a database, namely relations between tuples from different tables, into account.

Recently, another work [35] is proposed to embed database into semantic space using graph neural network strategy. Instead of treating tuples of the database as a sentence, [35] extracts a graph first from the database. The graph is undirected and a heterogeneous one, including different type of nodes reflecting different part of the database. Using the graph structure, the authors try to exploit the structural and hierarchical property of a database. For each tuple,

database value (token or keyword) and column, they create a node to generate the graph. Tuple nodes are then connected to token nodes which are connected to column nodes. In their extracted graph, there is no edge in between the same type of nodes, which they call *tri-partite* graph.

After the generation of the graph, they generate random biased walks over the graph, similar to the pioneering work *DeepWalk* [61], to generate sequences of nodes, which they later treat as sentences on which they perform the training to learn representation for each node and therefore each tuple, token and column. After generating the sequence of nodes, they apply training methodology of word2vec [22] to create a look up table for node representations, in which all of the instruments mentioned above reside. Since they follow word2vec methodology, they train their local embeddings in unsupervised fashion. However, similar to previous works, this study [35] also fails to capture the relational information in between tuples from different tables as well.

In this thesis, we propose a local database embedding strategy utilizing graph neural networks similar to [35]. However, instead of a tri-partite, heterogeneous graph, our graph is a homogeneous one having more dense connections including the ones between tuples from different tables. Another major difference is that instead of a random walk based embedding strategy, we train database embeddings in supervised fashion utilizing graph convolutional networks (GCN) [34]. Although we do not have natural labels for our tuples/nodes, we follow the strategy called *self-supervised learning* to implicitly associate each tuple/node with a label corresponding to the table it resides in with a novel algorithm we introduce considering database properties.

### 2.2.2 Query Recommendation in Databases

In [62], Yates et. al. try to cluster queries issued previously to generate candidate query sets. When clustering, they use semantic similarity between queries. The problem with this approach is that they do not consider current context when determining candidate queries. For that issue, Cao et. al. propose to use query

sequences to learn patterns to capture context information as well [63]. Although, for context aware query recommendation, query sequences sound promising, most of the query sessions include only single query [64], which makes it impossible to learn previous queries to suggest.

Although query recommendation in web search systems is a well studied problem [5], its application in relational databases has not been explored thoroughly. As initial studies, some SQL recommendation systems [65, 66, 67, 68, 69] have been proposed in relational databases in recent years. Chatzopoulou et. al. [65] create a recommendation matrix out of past queries of users and the tuples returned from those queries to apply the collaborative filtering technique. After finding the most probable tuples of high interest to the current user, a similarity score for each past query of users is calculated regarding these tuples of interest. The system ranks the queries according to their similarity scores and present these queries as recommendation to the user. In another similar study [69], the authors propose a recommendation system which exploits the query the user is issued currently in the session and content of the database instead of past queries of other users.

Following these works, QueRIE [67] and SnipSuggest [66] were proposed in the context of query recommendation in relational databases. Instead of sql queries or database tuples to be recommended, both studies preferred to focus on query fragments to employ recommendation system. Their difference lies in the approach for using similarity objective, that is, QueRIE exploits tuples (*witness-based*) returned from queries, whereas SnipSuggest utilizes a *feature-based* approach by exploring query fragments such as where clauses. Another difference is that, QueRIE recommends entire queries after finding similar query fragments with respect to the current query, while SnipSuggest tries to autocomplete the current prefix of the sql query by finding proper query fragments to append.

Recently, another work [70] was proposed following a different approach named *access area based*. In this thesis, we propose a recommendation system utilizing the witness based approach by focusing on database tuples returned by the query.

The novelty of our recommendation system is that we employ deep learning to calculate semantic similarity between tuples. To the best of our knowledge, there is no recommendation system proposed in the literature that makes use of semantic representations or deep learning in the context of recommendation in relational databases.



## Chapter 3

# Distributed Representations in Deep Learning

### 3.1 Word Embeddings

Popularity of word embeddings increased with the development of word2vec algorithm introduced by Mikolov et. al. [22]. Propelling side of their work is that vectors learned for words reveal compositionality, that is, applying simple algebraic operations on word vectors yields a result vector close to semantic representation of the composition of the words.

There are mainly two approaches introduced by Mikolov et. al. [22] to define the context for similar words. These are *Skip-Gram* and *CBOW* models. In both models, border of the similarity context is determined by a pre-defined window of various sizes tailored for different learning tasks. Depending on the size of the window, a pair of words is considered to be similar and processed in learning stage. CBOW model tries to predict the target word using the words surrounding inside the window, whereas in Skip-Gram model, the network takes the target word as input and it tries to predict the probabilities for each context word surrounding the target word. Both approaches for word2vec embedding are depicted in Figure

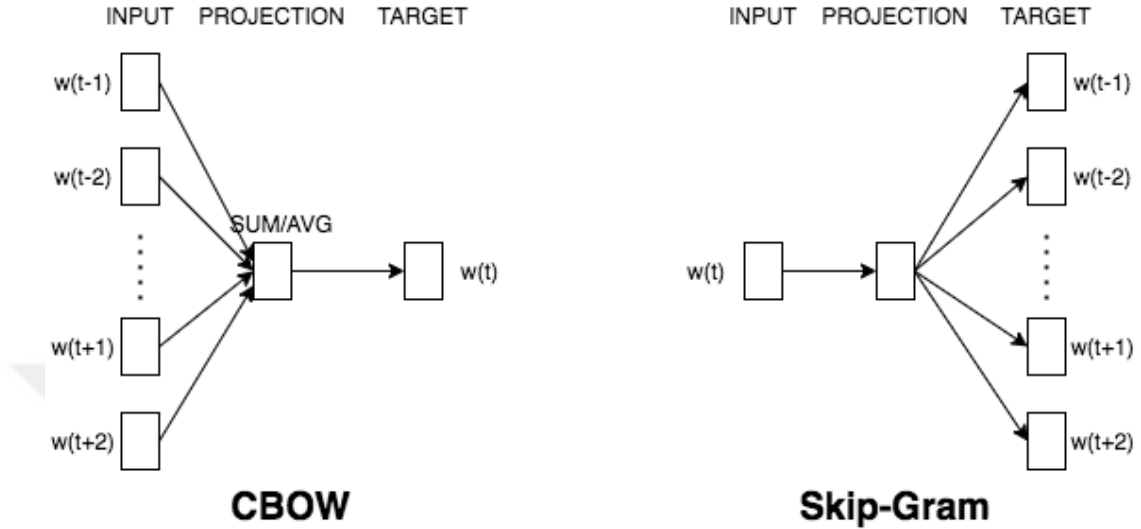


Figure 3.1: Neural network architecture for word2vec approaches

3.1.

In their following work [71], Mikolov et. al. address performance issues of word2vec learning algorithm. They introduce *negative sampling* to tackle performance issues of, especially, the Skip-Gram approach. Instead of updating all weights in a gradient, negative sampling tries to reinforce the strength of weights for a pair of words that are actually in the same context by reducing strength of weights for words that are not in the context. The approach simply samples a set of words to be used as negative context. Therefore, structure of the neural network architecture changes in a sense that the output layer becomes a simple binary classifier to differentiate positive and negative rather than a multi-class softmax.

Glove [57] is another word embedding algorithm which is a count-based model. Count-based co-occurrence matrix of words is processed and then factorized to get word representations. The main drawback of these approaches is the phenomena known as the out-of-vocabulary (OOV) issue. Especially, in large vocabularies, it is not feasible to store a vector representation for each possible unique word.

Alternative works have been proposed to deal with the OOV issue such as ELMo [72] and fastText [58]. ELMo utilizes bi-directional language model with

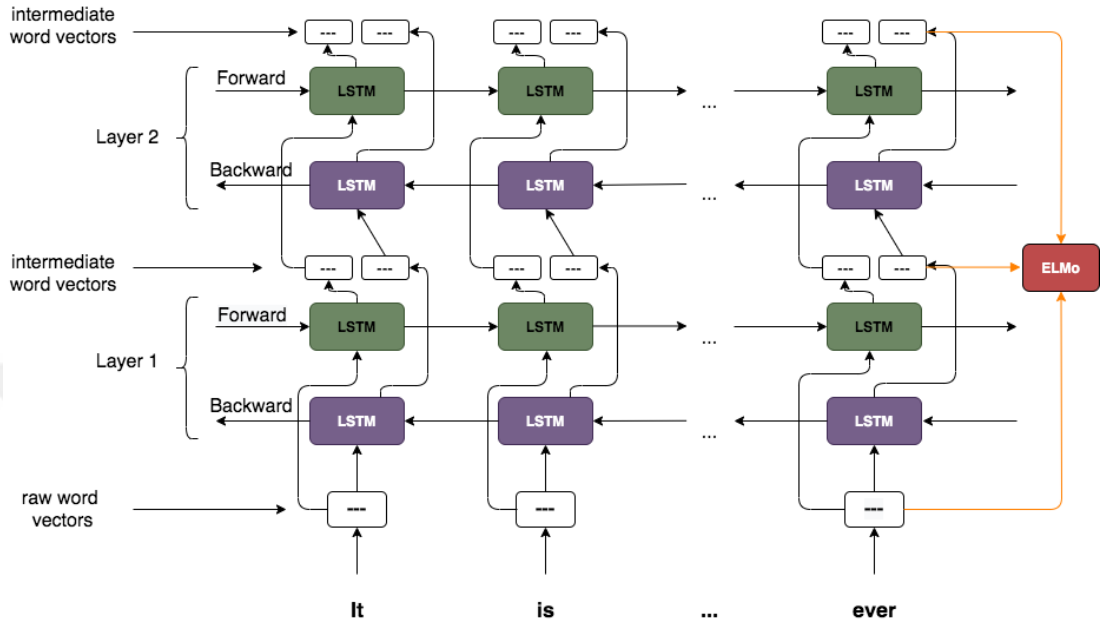


Figure 3.2: Neural network architecture of Embeddings from Language Models (ELMo)

the objective of sentence embedding. With 2 layers on top of each other, final representation of a word has 3 different inputs, which is shown in detail in Figure 3.2. ELMo representations have mainly 3 core features, which are as follows:

1. The representations are contextual meaning that during learning of word representations, context words are taken into account.
2. Unlike word2vec, the representations are deep, which indicates that in order to find out a representation of a word, forward pass (inference step) is done through neural layers instead of a lookup table.
3. In addition to context words, representations carry information from the characters inside the word as well, which deals with the OOV issue, word2vec and glove suffer from.

Similarly, fastText [58] deals with the OOV issue by splitting the words into character n-grams before learning word representations. Instead of word representations, sub-words are trained to produce embeddings, which makes it possible for even unseen words to have representations using character n-grams.

## 3.2 Character Embeddings

Word embeddings are useful to capture syntactic and semantic similarity between words that can be utilized in various NL tasks. Yet, in certain tasks such as POS-tagging and NER, intra-word (subword) information can be useful as well. Many works [73, 74] leverage character embeddings in such NLP tasks. Character embeddings is not only good for certain NLP tasks, but it is also effective to capture word level representations both semantically and orthographically [29, 75].

In addition to being effective to capture word representations, character embeddings are also beneficiary to deal with unseen words, for which word embedding methods such as word2vec and glove fail to extract a representation. In terms of memory usage, character embeddings methods are much more efficient than word embedding methods as well, since they need to store a look up representation table for each unique character as opposed to storing each unique word, which can be in millions depending on the corpus. FastText [58] and ELMo [72] utilize character level information to extract word level representations, which consequently solves the OOV issue.

## 3.3 Sentence Level Representations

In [76], Bengio et. al. learned distributed representations of words using neural language models which are then compiled into sentence representations using joint probabilities. Following analogies in [22], Le and Mikolov proposed paragraph2vec (doc2vec), an extension to word2vec, which captures representation of any textual data with varying length. They evaluated their sentence representations in sentiment analysis and information retrieval tasks, and reported that it yields better results compared to other baselines, both traditional machine learning and neural network approaches [77].



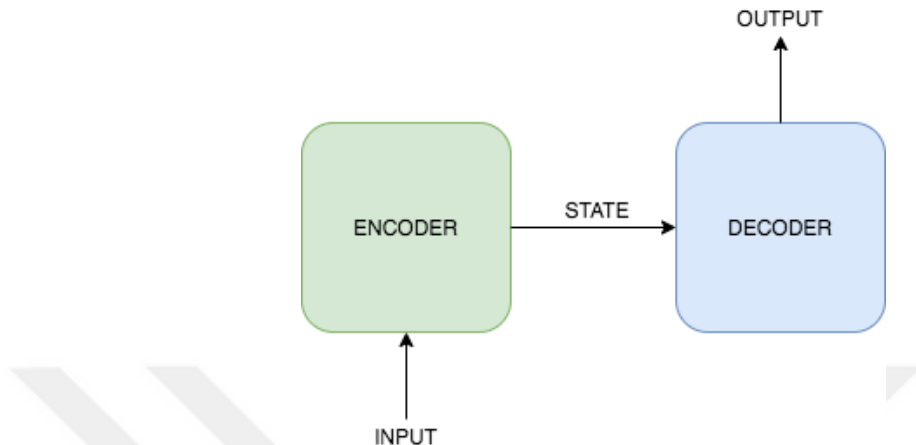


Figure 3.3: High level view of sequence to sequence neural network architecture (Encoder - Decoder)

Convolutional neural networks have been utilized to extract sentence representations in many works [78, 79]. An alternative work AdaSent, which is a hierarchical sentence model, was proposed by Zhao et. al. [80].

Recently, encoder-decoder approaches [31, 81, 32] have been exploited for sentence representations as well. There are two main parts in these seq2seq architectures; which are namely *Encoder* and *Decoder*. Encoder is responsible of constructing an intermediate level semantic representation, which later will be utilized by decoder to apply in a particular task. The high level depiction of seq2seq architecture is shown in Figure 3.3. For instance, given an input as a sentence, Encoder outputs a semantic representation for the sentence, which can be used as an input in another NLP task. This process of using pre-trained representations in another deep learning task is called *transfer learning*.

In many natural language processing tasks, sequence to sequence (*seq2seq*) network architectures utilizing recurrent neural networks in both encoder and decoder part of the architecture have been at the core of state-of-the-art solutions including language modelling, text classification, and machine translation. The seq2seq architecture utilizing RNN in both encoder and decoder parts is illustrated in Figure 3.4. In each time step, Encoder RNNs output a value and a calculated hidden state for the current time step,  $h_i$ ; given an input  $w_i$  and hidden state coming from previous time step,  $h_{i-1}$ . The output values are discarded

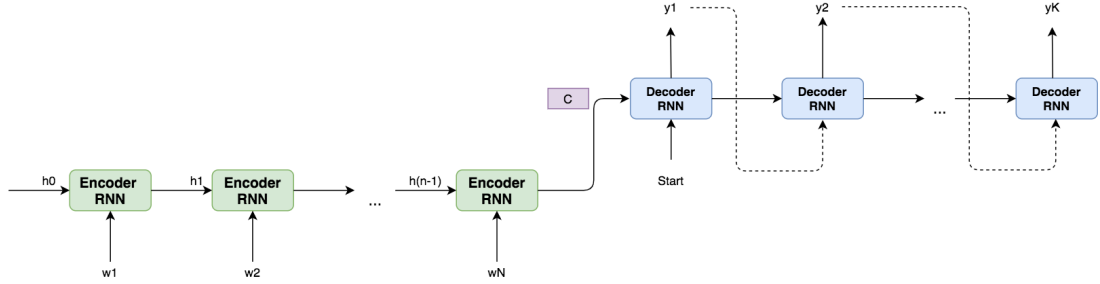


Figure 3.4: Architecture of RNN based Encoder - Decoder solution for machine translation problem

whereas hidden states are carried over other Encoder RNNs at future time steps. The hidden state value  $h_i$  for a time step  $i$  is calculated as follows,

$$h_i = F(W^{hh}h_{i-1} + W^{hx}x_i) \quad (3.1)$$

where  $hh$  and  $hx$  refers to weights to be trained for hidden states and inputs, respectively.

Through time steps, RNNs in Encoder encapsulate the information from previous steps and pass the cumulative hidden state to the next step. Last RNN in Encoder outputs the intermediate representation of the input sequence to be used by Decoder RNNs, which is called *context vector*,  $C$  in Figure 3.4. The initial state of Decoder RNNs are set to this  $C$  value and each output at the previous time step,  $y_{i-1}$ , is also given as input along with the current hidden state to produce output  $y_i$ .

Although, vanilla RNN are popular among possible choices, for either encoder or decoder, one can come up with many choices including CNN [79, 29], or other gates such as LSTM [82] and GRU [83]. When family of RNN networks are chosen for either one of the parts, they are mostly connected in bi-directional manner [48], having both *forward* and *backward* directions. The family of RNN networks with bi-directional connections have been shown to be the state-of-the-art architecture to implement in many NLP tasks.

In [31], Kiros et. al. learn sentence representations in a prediction setup for

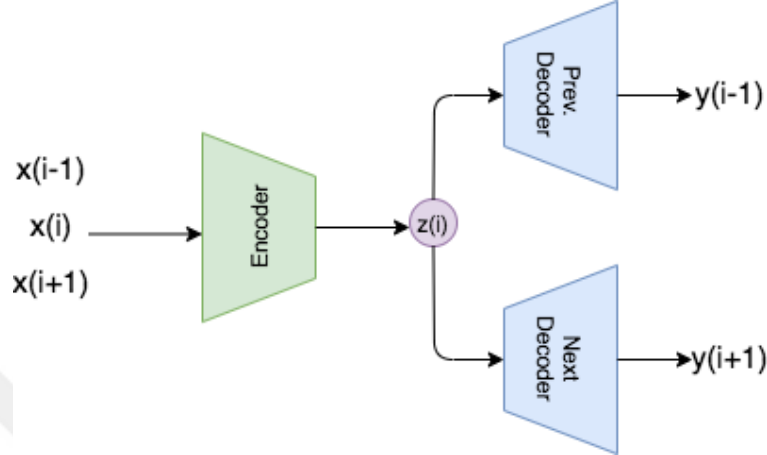


Figure 3.5: High level architecture of sentence representation as a prediction problem (Skip-Thoughts)

the next and previous sentences given the current one, similar to the skip-gram approach in [22], which they name their work as *Skip-Thoughts*. They use RNN-RNN encoder-decoder pairs with GRU [83] units, which is depicted in Figure 3.5 in a high level view. For training, input is comprised of sentence triplets; which are the current sentence ( $x_i$ ), the previous sentence ( $x_{i-1}$ ) and the next sentence ( $x_{i+1}$ ). Encoder tries to extract an intermediate representation ( $z_i$ ), which will be used by both decoders to try to generate ( $x_{i-1}$ ) and ( $x_{i+1}$ ), as an analogy to the skip-gram approach in word2vec [22].

Gan et. al. [81] utilize CNN-LSTM encoder-decoder architecture to model sentence representations with the bag-of-words approach in encoder part. Similar to [31], RNN-based encoder-decoder architecture is proposed in [32] with efficient learning strategy inspired by negative sampling in [71].

### 3.4 Transformers

Although sequence to sequence architectures utilizing uni or bi-directional recurrent neural networks have been shown to be effective in many NLP tasks, there are limitations of having these encoder-decoder architectures (e.g. Figure 3.4), especially in a task which requires a long sequential input. Major drawbacks of

the *seq2seq* architecture are 2-fold which are as follows:

1. When the input sequence gets longer, it is not effectively possible to accumulate all the information in Encoder and pass it to the very last step in Decoder, which makes the neural model to lose significant information during training.
2. Related to the first problem, having longer sequences causes vanishing gradient problem due to much deeper network required to train input of longer sizes. Although, LSTM and GRU units are proposed to deal with this problem, they also cannot solve the problem entirely.

In order to address aforementioned problems in *seq2seq* neural networks, a mechanism called *attention* [60, 84] is introduced. The attention mechanism tries to exploit hidden state values of Encoder RNNs at previous time steps instead of a single context vector output by the RNN at last time step by providing those values directly without applying any non-linear function to the Decoder RNNs. In other words, in each time step Decoder RNN has access to hidden state values of entire input sequence in Encoder to utilize during inference. Therefore, Decoder RNN can focus on certain parts of the input, paying more "attention", to generate a better output.

Although, attention [60] is a great step to solve the problems mentioned above about the *seq2seq* architecture, there is still an important limitation about sequential RNNs. Due to their nature, these sequential stack of RNNs are not suitable for parallelization. Therefore, *transformer* architecture [85] is introduced by Vaswani et. al. to tackle all the limitations arising from the *seq2seq* architecture for NLP tasks. Instead of recurrent neural networks, the transformer solely depends on attention mechanism, as the name of their study [85] is "Attention is all you need". Different from the attention approaches introduced in [60] and [84], they also propose a novel attention mechanism called *self-attention*. The general architecture of transformer network is illustrated in Figure 3.6.

In Figure 3.6, left side represents Encoder block whereas right side represents

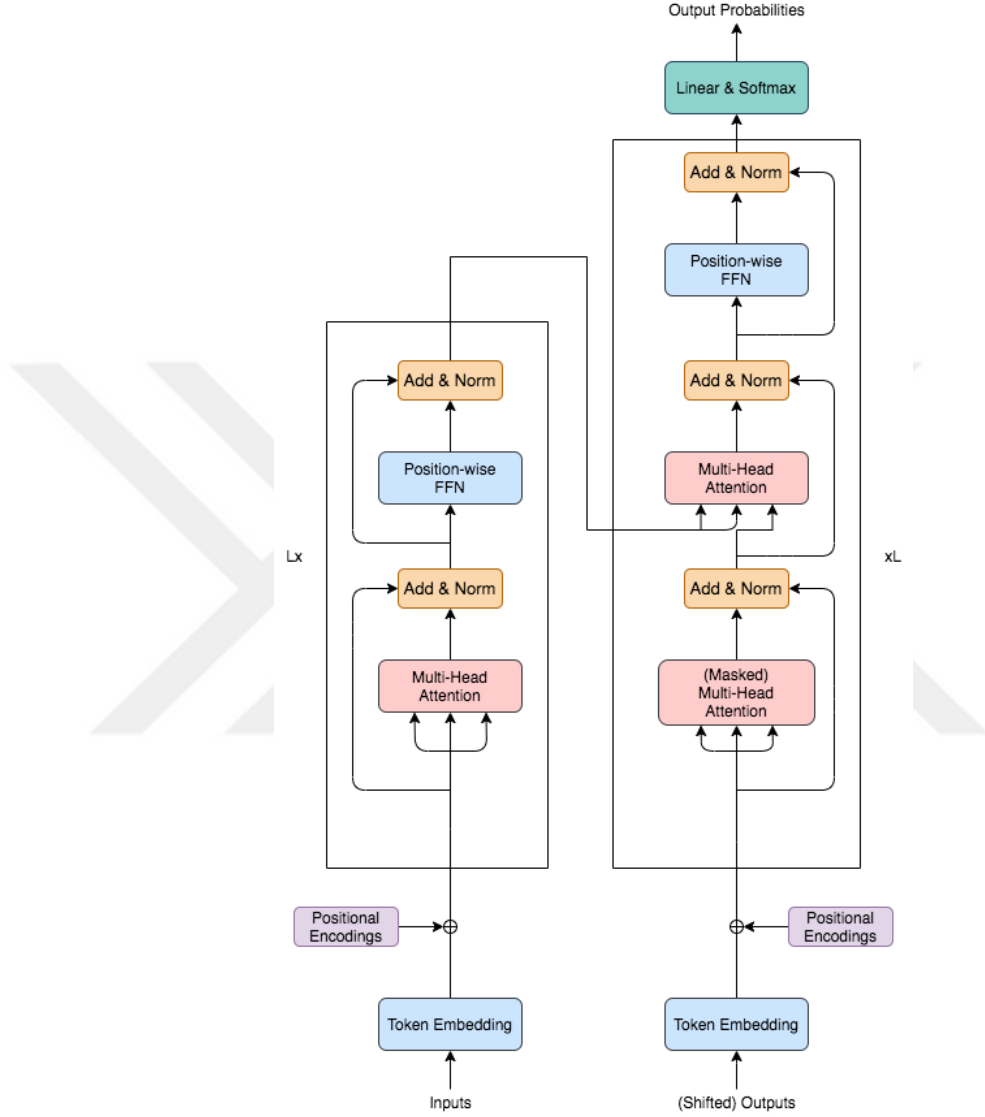


Figure 3.6: Neural network architecture for transformers

Decoder block, similar to the structure in Figure 3.3. The biggest difference between Encoder and Decoder is that Decoder has another layer of masked multi-head attention. The Encoder and Decoder blocks are actually utilized in stacks, which is given as 6 in the paper [85]. The general flow of information between these stacks is as follows:

- First encoder block gets token embeddings and after processing through self-attention and feed forward network layers, it passes the information to the second block of encoder.

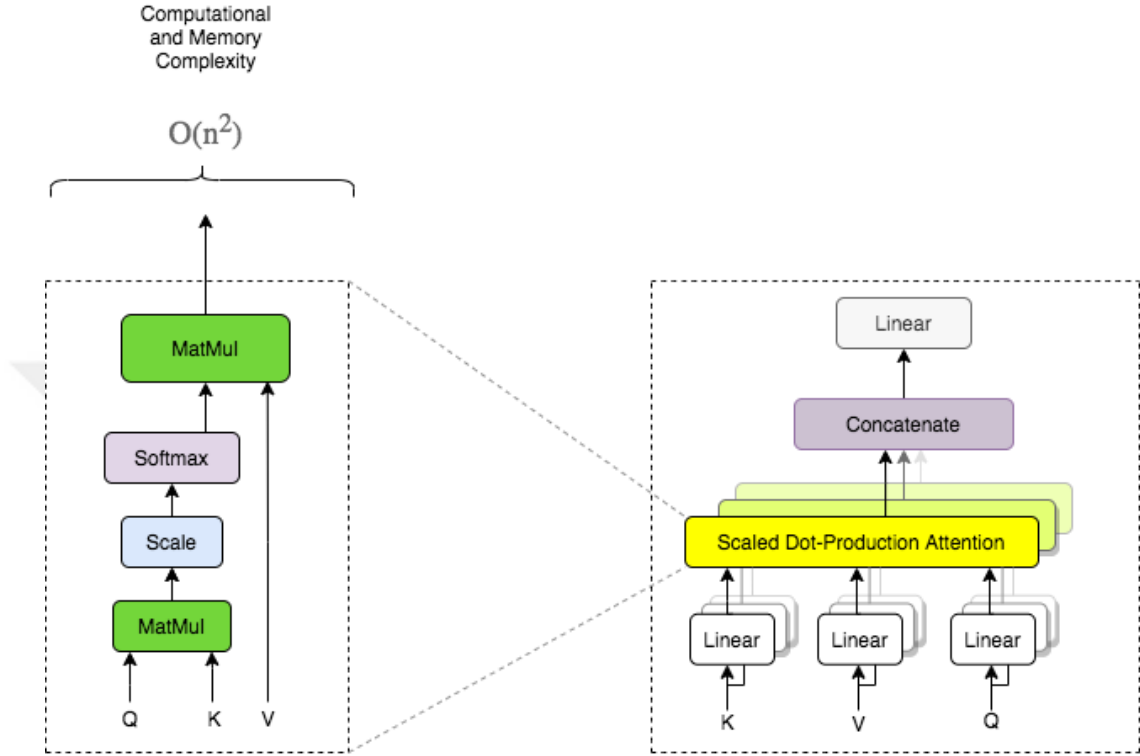


Figure 3.7: Neural network architecture of self-attention mechanism in transformers

- The last encoder block then passes all the information accumulated to each decoder blocks (i.e., the arrow in between Encoder and Decoder block in Figure 3.6, going from the last layer in encoder block to the multi-head attention layer in decoder block).

As mentioned above, Decoder blocks have 2 attention mechanisms; the first one is self-attention for target sequence whereas the second one utilizes information from encoder blocks as well. The architecture of self-attention network is given in Figure 3.7. Attention, in general, [60] is introduced to tackle the problem arising in tasks requiring long sequential inputs in seq2seq architectures utilizing RNNs. In these architectures, RNNs are responsible for inferring the relation between tokens in the sequential input. However, in transformers, there is no usage of recurrent neural networks. Therefore, self-attention is introduced to capture relational information between tokens in the input.

There are 3 different vectors calculated from input to each encoder; which are

*Key* ( $K$ ), *Value* ( $V$ ) and *Query* ( $Q$ ), as shown in Figure 3.7. These vectors are also trained during learning phase and updated. The calculation of self-attention for word  $w_i$  is as follows:

1. For each word  $w_j$ , we calculate a dot product in between query vector of  $w_i$ , that is  $q_i$ , and key vector of word  $w_j$ , that is  $k_j$  ( $q_i \bullet k_j$ ). This value represents the importance of other words when encoding a particular word.
2. These values are then normalized first by dividing by square root of the dimension size of key vector and then by applying soft-max activation function to make sure that sum of these importance scores for each word is 1.
3. Each normalized pair scores are then multiplied by value vectors of words,  $v_j$ . This whole process of calculations is illustrated in the left side of Figure 3.7. After multiplication, each score is summed to generate self-attention, a single vector, for word  $w_i$ .

Self-attention is calculated for each word in parallel, that is why the process is also referred to as *Multi-Head Attention* (e.g., Figure 3.6). After calculating self-attention for each word, these are concatenated and passed into the linear transformation layer, as shown in the right side of Figure 3.7.

After the introduction of transformer networks [85], there have been breakthrough studies utilizing transformer networks in NLP tasks such as BERT (Bidirectional Encoder Representations from Transformers) [86] and GPT [87, 88]. Although they both commonly employ transformers, there are differences in their solutions. The biggest difference is that, BERT only uses blocks of encoders while GPT employs only blocks of decoders.

BERT [86] also exploits a different objective for training language models called *masked language modelling*, in which random tokens among the input sequence are masked to generate by the model. BERT takes the input as a sequence and outputs a sequence at once.

GPT [87] is more similar to conventional language models in terms of being auto-regressive, which means that GPT iteratively generates one token at a time. For training, outputs are considered to be shifted version of input sequence of tokens.

After the success of these pre-trained models in many NLP tasks, there have been many works introduced by researchers on top of the transform structure especially, such as XLNet [89], Albert [90], Roberta [91], and DistilBERT [92]

### 3.5 Graph Neural Networks

Graphs are ubiquitous data structures that can be employed in many problems in computer science. They are not only effective storing data that exhibits structural information, they are also beneficial to visualize and therefore to sense characteristics of the data [93]. Because of these properties, they also play key role in machine learning due to fast evolving deep learning studies in the field. One of the deep learning applications employed on graph data is called *node embeddings*, which seeks to learn a distributed representation for each node in a graph to reflect structural information. Later, these representations are to be utilized in many graph related problems such as node classification, edge prediction, and recommender systems.

Among deep learning studies working with graph data structures, also referred as *Graph Neural Networks (GNN)*, node embeddings refers to the problem of encoding nodes into low-dimensional vector space where geometric relations represent interactions in the original graph [94]. Graph neural networks can be regarded as an application of a general encoder-decoder framework [95] in which the encoder tries to extract a low-dimensional vector representation in latent space whereas the decoder tries to decode structural information out of that representation. In addition to these encoder-decoder functions, it is also important to come up with appropriate similarity objective in between nodes to calculate the loss and consequently train the model effectively.



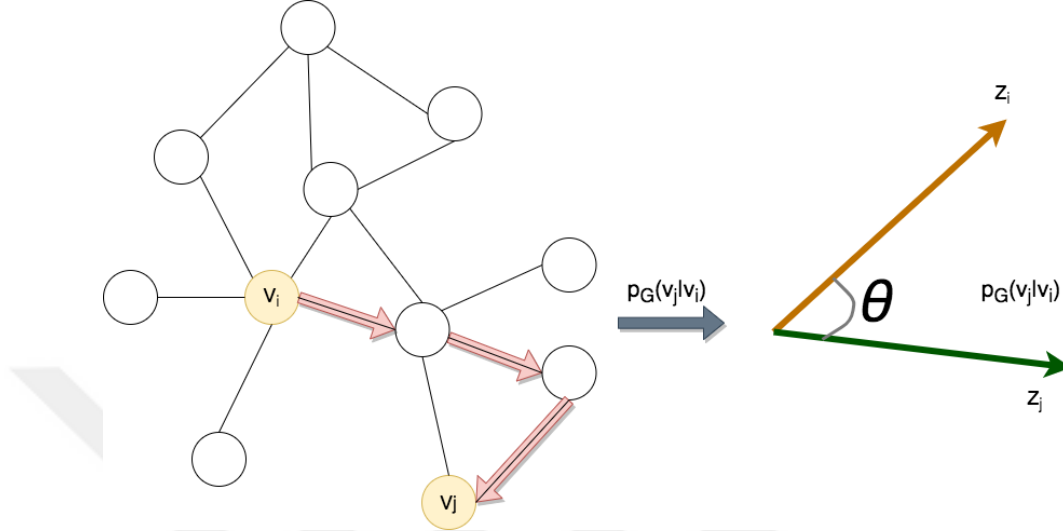


Figure 3.8: Sample random walk on graph and its reflection in latent space

One of the approaches for node embeddings is *shallow embedding* [95], which basically tries to extract look up tables (mappings) for both encoder and decoder functions. Among shallow embedding approaches, some of most popular and state-of-the-art solutions in GNN problem are DeepWalk [61], LINE [96], and node2vec [97]. DeepWalk and node2vec first sample a set of fix length random walks over the graph, an example of which is illustrated in Figure 3.8. Basically, they generate walks starting from each node and try to estimate co-occurrence of node pairs processing walks. Ideally, the probability of a node  $V_j$  is visited during a walk which starts at node  $V_i$  is to be roughly proportional to the angel in between output embedding vectors  $Z_i$  and  $Z_j$ . node2vec differs from DeepWalk study in the way walks are generated. node2vec has two parameters to control walks to consider both local neighboring nodes and global communities (e.g., breadth vs depth first traversal)

Although these shallow node embedding methods are effective in variously structured graphs, they have certain drawbacks [95] which can be summarized as follows.

- These techniques basically try to train a lookup table for encoders. During training, no parameters are shared in between nodes, which might be problematic especially for big graphs in terms of memory requirement.

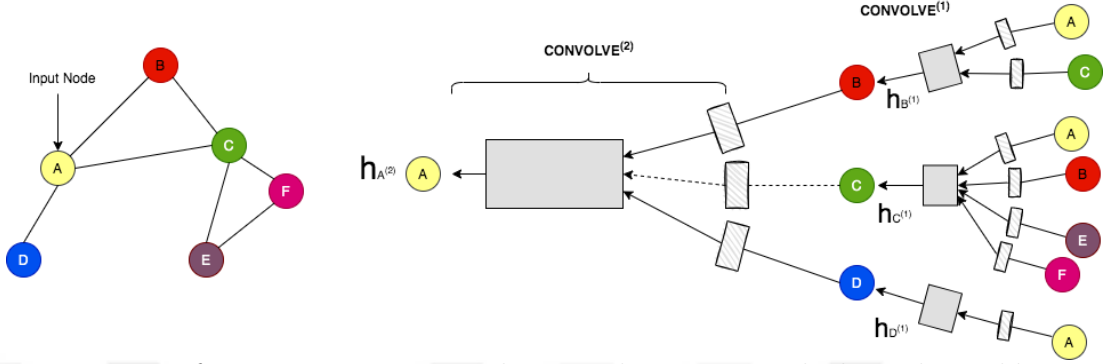


Figure 3.9: Information aggregation by convolution in graph through neighboring nodes

- They fail to exploit node features that may be available at the time for training. These features may exhibit significant signals to consider embeddings, which limits efficacy of the approaches.
- Similar to word embedding approaches that output a lookup table (e.g., word2vec or glove), shallow node embedding techniques are transductive, which means they cannot produce embeddings for the nodes not seen during training.

In order to address the above limitations, various graph neural network architectures [34, 98, 99, 100] have been proposed following a paradigm called *message passing*. Message passing refers to a technique which is used as information aggregation from neighboring nodes during training of node embeddings. This technique tries to exploit the important features nodes might have during training to come up with better representations for nodes.

Graph convolutional networks (GCNs) [34] are one of the most popular and successful works that try to leverage this neighborhood aggregation technique. An illustration of the neighborhood aggregation technique is provided in Figure 3.9. For a particular node  $A$ , GCN processes features of neighboring nodes that are 2 hops away at most.

In the encoding phase, the initial embeddings of a node is set to node features if available. Then at each iteration, nodes aggregate embeddings of neighboring

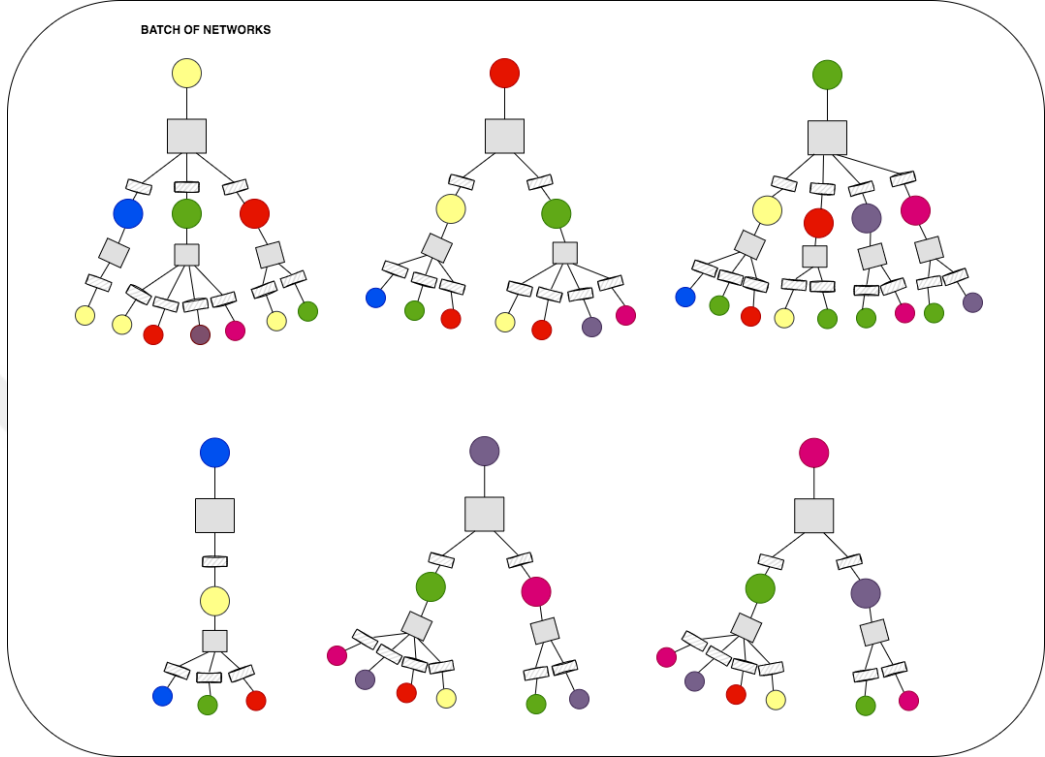


Figure 3.10: Batch of input graphs for the example graph given in Figure 3.9

nodes to update their embeddings. After accumulating information from neighbors, a non-linear function (e.g., Relu) is applied. For aggregation, GCN uses weights during combination of the information (i.e., filter), therefore the operation is named as convolve. The difference between GCN and other architectures employing neighboring aggregation lies in the way they perform these aggregation and combination of the accumulated information with the node information from the previous layer.

This aggregation operation is done for each node in a single batch, named as *full-batch*. For each node, sub-graphs composing neighboring nodes including the ones that are 2 hops away are extracted as input. Full-batch generation of the graph given in Figure 3.9 is depicted in Figure 3.10. The nodes with the same color represents a particular node in the graph given in Figure 3.9. Another advantage of GCN like approaches is that these aggregations in a single iteration of a full-batch can be done in parallel.

At the end of the last convolution layer, GCN employs a softmax layer for

supervised learning with classification objective. In order to calculate the loss, cross-entropy loss is utilized. More information about graph neural networks; their architectures, their applications to the specific problems in machine learning and their advantages/disadvantages can be found in survey papers [95, 101].



## Chapter 4

# Keyword Mapping in NLIDB (DBTagger)

Consider<sup>1</sup> the below natural language query examples run on the sample IMDB movie database shown in Figure 4.1 to better understand the challenges in *keyword mapping* problem.

**Example NL Query 1.** *"What is the writer of The Truman Show?"*

**Challenge 1.** The very first challenge in *keyword mapping* is to differentiate and categorize tokens in the query either as database relevant or not. For instance, some of the words in Example 1 (e.g., "is", "the", "of") are just stop words that are needed not to be considered as potential mapping target. An ad-hoc solution is to filter certain words using a pre-defined vocabulary, however such a solution removes "The" in Example 1 preceding the actual database value that needs to be mapped, which will cause the wrong translation.

**Challenge 2.** Another important challenge is to detect multi-word entities (mostly database values), "The Truman Show" in Example 1. The most common approach is to build look-up tables or indexes on n-grams of database values and

---

<sup>1</sup>This chapter is based on the work [102]; Arif Usta, Akifhan Karakayali, and Özgür Ulusoy. 2021. DBTagger: multi-task learning for keyword mapping in NLIDBs using Bi-directional recurrent neural networks. Proc. VLDB Endow. 14, 5 (January 2021), 813–821. DOI:<https://doi.org/10.14778/3446095.3446103>

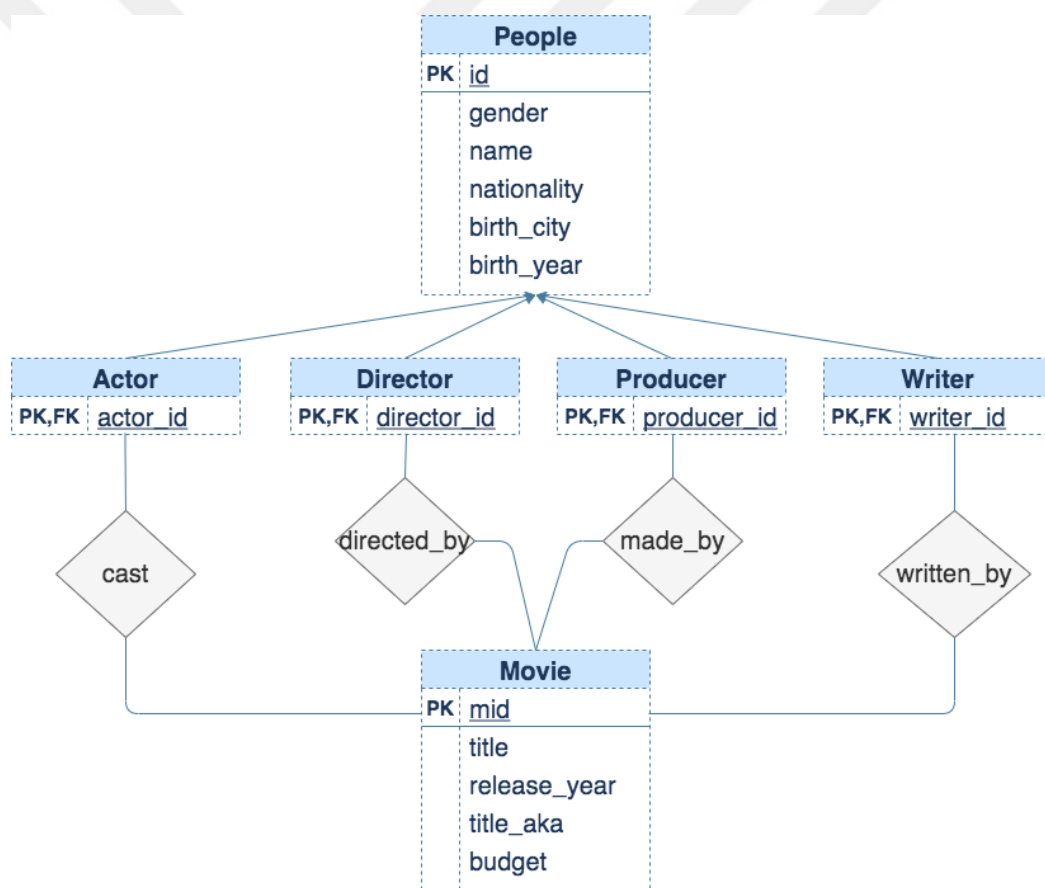


Figure 4.1: ER diagram of a subset of IMDB movie database

calculate semantic and/or lexical similarity over the candidates. Yet, this is a costly process for on-the-fly calculations regarding possible n-grams of the given NL query.

**Example NL Query 2.** *"Find all movies written by Matt Demon."*

**Example NL Query 3.** *"How many movies are there that are directed by Steven Spielberg and featuring Matt Demon?"*

**Challenge 3.** Consider the queries given in Examples 2 and 3. In the queries, tokens ("written" and "featuring") referring to database tables are syntactic and semantic variations of the actual table ("written.by" and "cast", respectively) that they mapped to in the database (Figure 4.1). To handle such a challenge, lexical and semantic similarities of tokens over database elements (table and attributes) can be calculated using a third party database such as WordNet [21]. However, in addition to being a costly process to calculate such similarities online, such a solution cannot cover all possible variations of every map target in the database schema. Also, similarity calculation approach requires a manually crafted threshold,  $\tau$ , to determine how much similarity is sufficient to map to a particular schema element, which makes it undesirable.

**Challenge 4.** One of the usages of *keyword mapping* step is to resolve ambiguities before getting into translation step. In the above examples, "Matt Demon" refers to a database value residing in multiple tables (e.g., actor, writer). Actual mapping of the keyword is determined by the mappings of neighbouring words surrounding, which implies that query-wise labelling considering *coherence* rather than independent labelling can be beneficiary.

**Challenge 5.** In addition to an effective solution, an ideal keyword mapping approach must be efficient to be deployed on interfaces where users run queries online. Mapper should output the result in reasonable time.

In what follows, we summarize how our proposed solution, DBTagger, tackles each challenge mentioned above:

- DBTagger does not apply any pre-processing or filtering to the original NLQ to remove or detect non-relevant keywords in terms of mapping, which

covers the corner cases such as keywords (*The Truman Show*, shown in the example in the first challenge) having such possible stop words (i.e., the) as part. Our neural model differentiates the stop word "the" and part of keywords "the" with the help of Conditional Random Fields (CRF), since instead of independent classification, CRF utilizes query wise labelling, which helps the model to detect "the"s around candidate keywords to be part of the keyword. Not using any filtering and having CRF for query wise labelling address the Challenge 1 introduced above.

- Our model utilizes POS Tags of the tokens. We use the encoded representation of the sentence output from POS layer as input into the Tag Layer, and similarly output of Tag Layer as input into the Schema(Final) Layer. Such an architecture helps the model to abstract actual database values and to make it possible working with schema types to detect multi-word entities with the help of last layer CRF, without needing look-ups of n-grams of database values. Such a solution also helps us to detect syntactically or semantically varied words referring to database tables or columns (i.e., "featuring" word refers to "cast" table, in Challenge 3). In the training phase, such a word is represented with its POS Tag VBG (one of Verb tags) and type Tag TABLEREF to indicate table presence to the model along with its word representation carrying semantic of the word. This architecture of DBTagger addresses both Challenge 2 and 3.
- Similarly, resolving ambiguities of keywords is handled by query wise classification, thanks to CRF. For instance, VLDB may refer to both Journal or Conference name in a scholar schema. CRF layer determines final tag (either Journal.Name or Conference.Name) by considering labels of neighbouring words. If there is another word around VLDB with Journal tag, the model predicts Journal.Name for such a value, resolving ambiguity automatically (addressing Challenge 4), which currently state-of-the-art solutions for keyword mapping suffer to deal with.
- DBTagger is a deep neural model, which trains using the query logs of a particular schema only once, as offline setup. When the NLQ is to be translated in online, DBTagger just applies inference setup on the learned



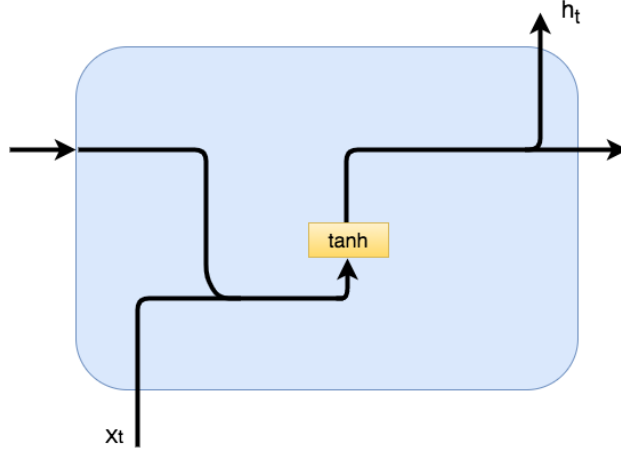


Figure 4.2: Recurrent Neural Network (RNN) architecture

model, which outputs the tags in milliseconds (Challenge 5). Most importantly, this run time is agnostic to database schema and number of tables, tuples, columns presenting in the particular schema, which shows that DB-Tagger is scalable for especially bigger tables.

In the following sections, we first give background information about neural network structure utilized for sequence tagging problems such as POS tagging and NER in NLP community. Next, we explain network structure of *DBTagger* by pointing out modifications we introduce on top of the state-of-the-art sequence tagging architecture. Lastly, we discuss how we annotate three different class labels of tokens to employ training.

## 4.1 Deep Sequence Tagger Architecture

POS tagging and NER refer to sequence tagging problem in NLP for a particular sentence to identify parts-of-speech such as noun, verb, adjective and to locate any entity names such as person, organization, respectively. We argue that these problems are formally similar to keyword mapping problem in NLIDB.

Recurrent Neural Networks (RNN) are at the core of architectures to handle such problems, since they are a family of networks that perform well on sequential

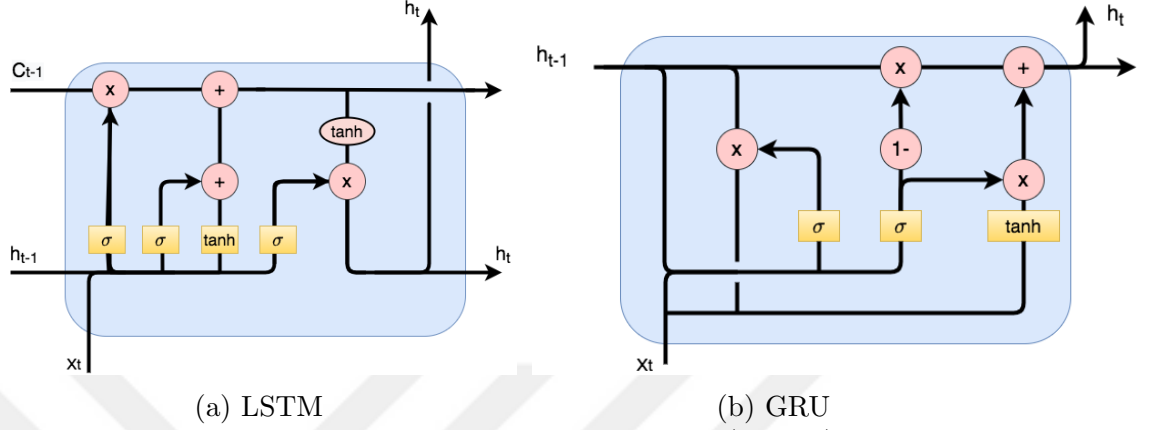


Figure 4.3: Architectures of Long-Short Time Memory (LSTM) and Gated Recurrent Units (GRU)

data input such as a sentence. In this particular problem, sequence tagging (*keyword mapping*), RNNs are employed to output a sequence of labels for the original sentence (the query), input as a sequence of words.

In RNN networks, which is depicted in 4.2, the basic goal is to carry past information (previous words) to future time steps (future words) to determine values of inner states and consequently the final output, which makes them preferable architecture for sequential data. Given  $x_t$  as input at time step  $t$ , calculation of hidden state  $h_t$  at time step  $t$  is as follows:

$$h_t = f(Ux_t + Wh_{t-1}) \quad (4.1)$$

In practice, however, RNN networks suffer from *vanishing gradient problem*, therefore the limitation was overcome by modifying the gated units of RNNs; such as LSTM [82] and GRU[83]. Compared to vanilla RNN, LSTM has *forget gates* and GRU comprises of *reset* and *update* gates additionally. In terms of complexity, GRU is simpler than LSTM, yet they perform similar empirically in many tasks.

We experimented with both structures and we chose GRU due to its better performance in our experiments. In GRU, Update Gates decide what information to throw away and what new information to add, whereas Reset Gate is utilized

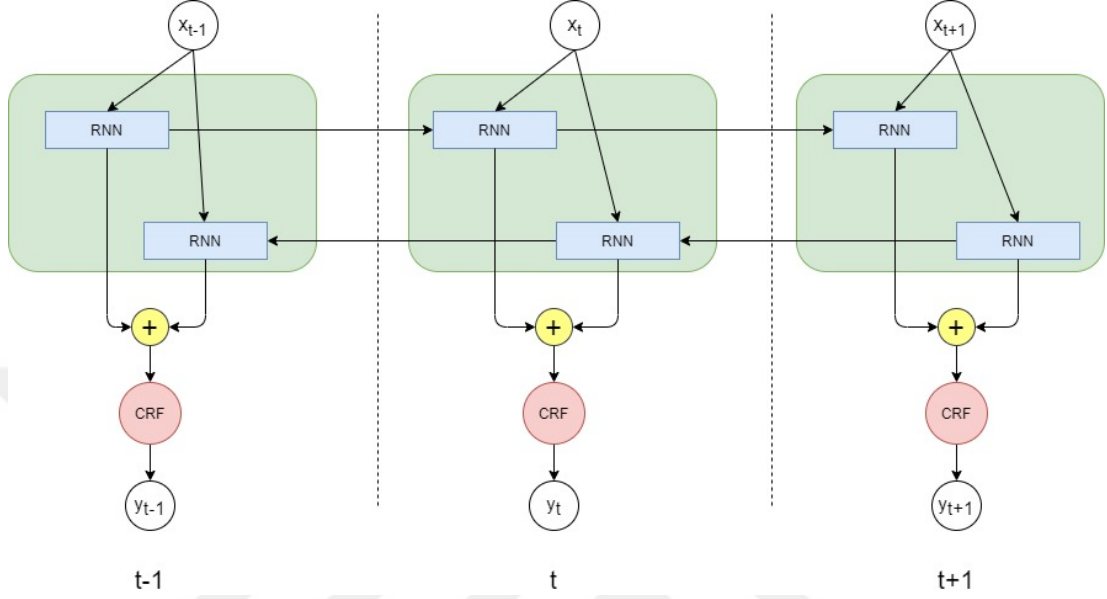


Figure 4.4: Architecture of deep sequence tagger neural network

to decide how much past information to forget. The calculation of GRU is as follows:

$$z = \sigma(U_z.x_t + W_z.h_{t-1}) \quad (4.2)$$

$$r = \sigma(U_r.x_t + W_r.h_{t-1}) \quad (4.3)$$

$$z_t = \tanh(U_z.x_t + W_s.(h_{t-1} \bullet r)) \quad (4.4)$$

$$z_t = \sigma(U_z.x_t + W_z.h_{t-1}) \quad (4.5)$$

In sequence tagging problem, in addition to past information we also have future information as well at a given specific time,  $t$ . For a particular word  $w_i$ , we know the preceding words (past information) and succeeding words (future information), which can be further exploited in the particular network architecture called, *bi-directional RNN* introduced in [48]. Bi-directional RNN has two sets of networks with different parameters called forward and backward. The concatenation of the two networks is then fed into the last layer, where the output is determined. This process is demonstrated in the Figure 4.4, named deep sequence tagger network.

Sequence tagging is a supervised classification problem where the model tries to predict the most probable label from the output space. For that purpose, although conventional *softmax* classification can be used, *conditional random field (CRF)* [103] is preferred. Unlike independent classification by softmax, CRF tries to predict labels sentence-wise by taking labels of the neighboring words into consideration as well. This feature of CRF is what makes it an attractive choice especially in a problem like *keyword mapping*. CRFs for each class of tags are appended to uni-directional GRU, depicted in lower part of the Figure 4.5. This finding is also reported in [50], where authors claim that CRF as the output layer gives 1.79 more accuracy in NER task. The final outlook of the architecture of deep sequence tagger is depicted in Figure 4.4.

## 4.2 DBTagger Architecture

Formally, for a given NL query, input  $X$  becomes a series of vectors  $[x_1, x_2, \dots, x_n]$  where  $x_i$  represents the  $i^{th}$  word in the query. Similarly, output vector  $Y$  becomes  $[y_1, y_2, \dots, y_n]$  where  $y_i$  represents the label (actual tag) of the  $y^{th}$  word in the query. Input must be in numerical format, which implies that a numerical representation of words is needed. For that purpose, the word embedding approach is state-of-the-art in various sequence tagging tasks in NLP [46] before feeding into the network. So, embedding matrix is extracted for the given query,  $W \in R^{n \times d}$ , where  $n$  is the number of words in the query and  $d$  is the dimension of the embedding vector for each word.

For the pre-calculated embeddings, there are different techniques to choose such as word2vec [22], Glove [57], fastText [58] or Elmo [72]. We used fastText[58] due to it being one of the representation techniques considering sub-word (character n-grams) as well to deal with the out of vocabulary token problem better.

We consider  $G$  to be 2-dimensional scores of output by the uni-directional GRU with size  $n \times k$  where  $k$  represents the total number of tags.  $G_{i,j}$  refers to score of the  $j^{th}$  tag for the  $i^{th}$  word. For a sequence  $Y$  and given input  $X$ , we

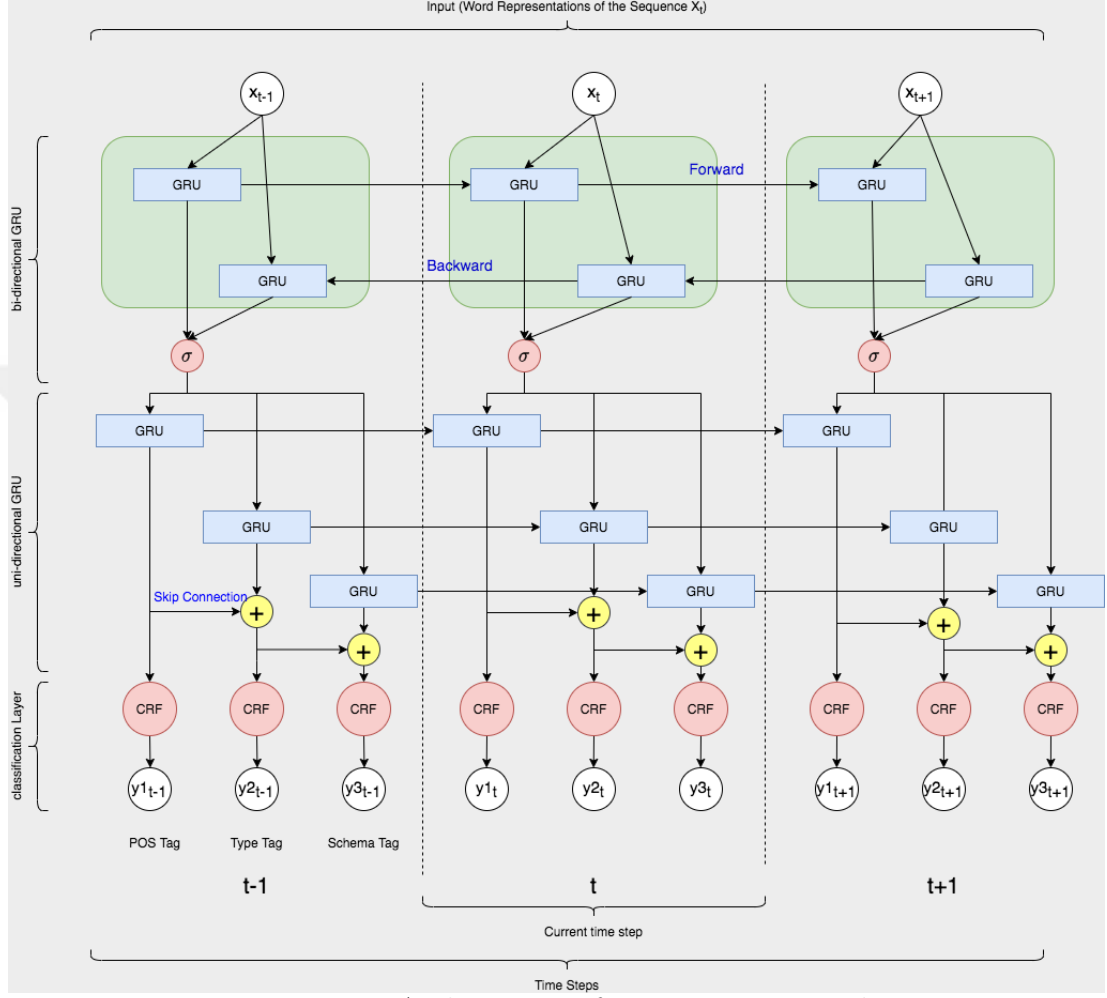


Figure 4.5: Architecture of DBTagger Network

define tag scores as;

$$s(X, Y) = \sum_{i=1}^n A_{y_i, y_{i+1}} + \sum_{i=1}^n G_{i, y_i} \quad (4.6)$$

where  $A$  is a transition matrix in which  $A_{i,j}$  represents the score of a transition from the  $i^{th}$  tag to the  $j^{th}$  tag. After finding scores, we define probability of the sequence  $Y$ :

$$p(Y|X) = \frac{e^{s(X,Y)}}{\sum_{\bar{Y} \in Y_x} e^{s(X,\bar{Y})}} \quad (4.7)$$

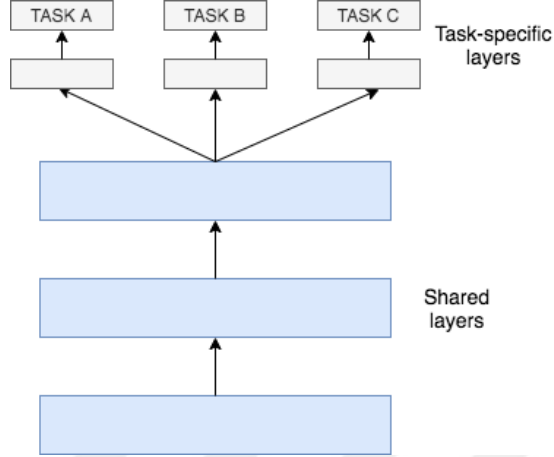


Figure 4.6: Hard parameter sharing in multi-task learning

where  $\bar{Y}$  refers to any possible tag sequence. During training we maximize the log-probability of the correct tag sequence and for the inference we simply select the tag sequence with the maximum score.

In our architecture, we utilize *Multi-task learning* by introducing two other related tasks; POS and type levels (shown in Figure 4.5). The reason we apply multi-task learning is to try to exploit the observation that actual database tags of the tokens in the query are related to POS tags. Besides, multi-task learning helps to increase model accuracy and efficiency by making more generalized models with the help of shared representations between tasks [104].

For multi-task learning in the context of deep learning, there are mainly two different ways to implement multi-task learning; which are *hard* or *soft* parameter sharing [104]. In the architecture of DBTagger, we utilize hard parameter sharing technique. It generally refers to having hidden layers shared across all tasks while also having task specific tasks appended to the hidden layers. The general architecture implementing hard parameter sharing as a multi-tasking approach in deep learning tasks is depicted in Figure 4.6.

POS and Type tasks are trained with schema task to improve accuracy of schema (final) tags. For each task, we define the same loss function, described above. During backpropagation, we simply combine the losses as follows;

$$\begin{aligned}
L_{total} &= \sum_{i=1}^3 w_i \times L_i \text{ subject to} \\
\sum_{i=1}^3 w_i &= 1
\end{aligned}
\tag{4.8}$$

where  $w_i$  represents the weight of  $i^{th}$  task and  $L_i$  represents the loss calculated for the  $i^{th}$  task similarly.

Another technique we integrate into the neural architecture is *skip-connection*. Skip connection is used to introduce extra node connections between different layers by skipping one or more layers in the architecture. With skip connections, the model provides an alternative for gradient to back propagation, which eventually helps in convergence. Also, in a recent work [105], it was shown by experimental evidence that skip connection helps improving the training by eliminating singularities.

The technique has become compulsory component in many neural architectures deployed in computer vision community, such as the famous architectures ResNet [106] and DenseNet [107]. In the architecture of DBTagger, for each task except the first one (POS), we additionally feed the output of uni-directional GRU layer of previous task into CRF layer of the next task ( $i + 1^{th}$  task). With these connections, we further carry the information of previous tasks to later tasks and eventually to the final task, schema tagging.

In most applications, skip connections use outputs of preceding layers as input for future layers. The term is also referred as *residual connections* in the literature, where residual connection means using an output of a layer by adding or concatenation into any layer directly, without applying non-linear activation functions to allow gradients to flow without exploding or vanishing. We use the output of uni-directional GRU of previous task, without applying any non-linearity by concatenating with the output of uni-directional GRU of future task as an implementation of a residual connection. In a different study [108] in which a deep architecture is utilized, the authors implement both multi-tasking and skip

Table 4.1: An example natural language query with its associated tags corresponding to each word in three different levels

NL query	POS tags	Type tags	Schema tags
who	WP	O	O
acted	VBD	TABLEREF	cast
John	NNP	VALUE	cast.role
Nash	NNP	VALUE	cast.role
in	IN	COND	cond
the	DT	O	O
movie	NNN	TABLE	movie
A	DT	VALUE	movie.title
Beautiful	JJ	VALUE	movie.title
Mind	NN	VALUE	movie.title

connections, similar to our architecture. They named the particular architecture as *cross-skip connections*.

## 4.3 Annotation Scheme

In our problem formulation, every token (words in the natural language query) associates three different tags; namely part-of-speech (POS) tag, type tag and schema tag. In the following subsections, we explain how we extract or annotate each of them in detail.

### 4.3.1 POS Tags

To obtain the POS tags of our natural language queries we used the toolkit of Stanford Natural Language Processing Group named Stanford CoreNLP[109]. We use them as they are output from the toolkit, without doing any further processing since the reported accuracy for POS Tagger (97%) is sufficient enough.



### 4.3.2 Type Tags

In each natural language query, there are keywords (words or consecutive words) which can be mapped to database schema elements such as table, attribute or value. We divide this mapping into two levels; type tagging and schema tagging. Type tags represent the type of the mapped schema element to be used in the SQL query. In total we have seven different type tags;

- **TABLE:** NLQs contain nouns which may inhibit direct references to the tables in the schema, and we tag such nouns with *TABLE* tag. In the example NL query given in Table 4.1, noun *movie* has a type tag as *TABLE*, which also supports the intuition that schema labels and pos tags are related.
- **TABLEREF:** Although the primary sources for table references are nouns, some verbs contain references to the tables most of which are relation tables. *TABLEREF* tag is used to identify such verbs. Revisiting the example given Table 4.1, the verb *acted* refers to the table *cast*, and therefore it is tagged with *TABLEREF* to differentiate better the roles of POS tags in the query.
- **ATTR:** In SQL queries, attributes are mostly used in *SELECT*, *WHERE* and *GROUP BY* clauses. Natural language queries may contain nouns that can be mapped to those attributes. We use *ATTR* tag for tagging such nouns in the natural language queries.
- **ATTRREF:** Like *TABLEREF* tag, *ATTRREF* tag is used to tag the verbs in the natural language query that can be mapped to the attributes in the SQL query.
- **VALUE:** In NLQs, there are many entity like keywords that need to be mapped to their corresponding database values. These words are mostly tagged as *Proper noun-NNP* such as the keyword *John Nash* in the example query. In addition to these tags, it is also likely for a word to have a *noun-NN* POS tag with a *Value* tag corresponding to schema level. In order to handle these cases having different POS tags, we have *Value* type tags (e.g., *Mind* keyword in the example query is part of a keyword that needs to be mapped as *value* to

*movie.title*). Keywords with *Value* tags can later be used in the translation to determine "where" clauses in SQL.

- **COND**: After determining which keywords in the query are to be mapped as values, it is also important to identify the words that imply which type of conditions to be met for the SQL query. For that purpose, we have the *COND* type tag.
- **O (OTHER)**: This type of tag represents words in the query that are not needed to be mapped to any schema instrument related to the translation step. Most stop words in the query (e.g., the) fall into this category.

### 4.3.3 Schema Tag

Schema tags of keywords represent the database mapping that the keyword is referring to; name of a table, or attribute. Tagging a keyword with a type tag is important yet incomplete. To find the exact mapping the keyword refers to, we defined a second level tagging where the output is the name of the tables or attributes. For each entity table (e.g. *movie* table in Figure 4.1) and for each non-PK or non-FK attribute (attributes which have semantics) we define a schema tag (e.g *movie*, *people*, *movie.title*, etc., referring to Figure 4.1). We complete possible schema tags by carrying *OTHER* and *COND* from type tags. We use the same schema tag for attributes and values (e.g *movie.title*), but differentiate them at the inference step by combining tags from both type tags and schema tags. If a word is mapped into *Value* type tag as a result of the model, its schema tag refers to the attribute in which the value resides.

In order to annotate queries, we annotate each word in the query for three different levels mentioned above. While POS tags are extracted automatically, we manually annotate the other two levels. Annotations were done by three graduate and three undergraduate computer science students who are familiar with database subject. Although annotation time varies depending on the person, on the average it took a week to annotate tokens by a single person for two levels

(type and schema) for a query log with 150 NL questions, which we believe is practical to apply in many domains.



## Chapter 5

# SQL Query Recommendation in Databases (Conquer)

In this chapter we present *Conquer*, CONtextual QUery Recommendation system which utilizes distributed representations learned through Graph Convolution Networks. We first give detailed information about how we learn local embeddings to extract distributed representations for database tuples. Following that in the next section, we mention how we utilize these representations to generate a recommendation.

### 5.1 Embedding Relational Database into Latent Space Using Graph Neural Networks

A relational database is composed of two main components: namely **Entities** and **Relations** between these entities [110]. Therefore, an ideal representation of tuples should reflect these components. The tuples residing in the same table (Entity) having the same values should have similar vector representations. Also, the tuples residing in different tables which are connected through foreign keys (Relation) should have similar vector representations.

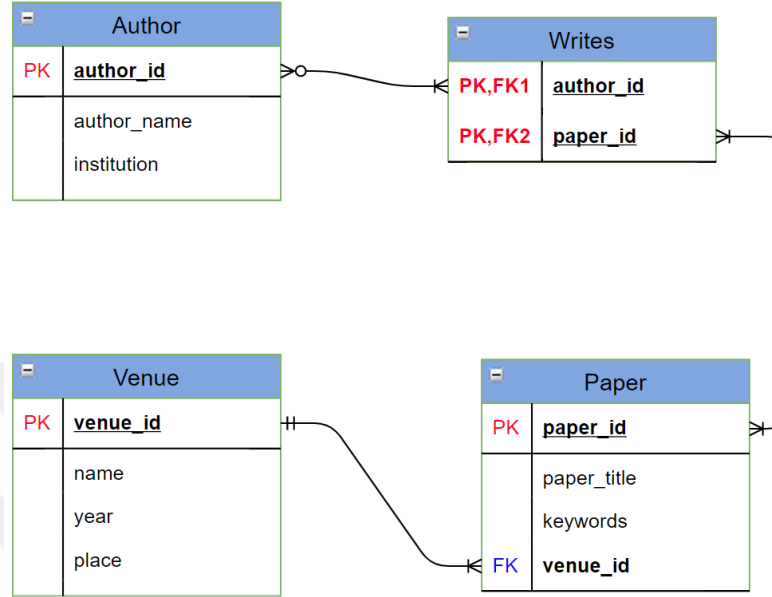


Figure 5.1: A sample ER Diagram for Scholar Schema

An example schema of a relational database is shown in Figure 5.1 along with synthetic instances of a small size for the schema in Figure 5.2. In the given schema; *Author*, *Paper* and *Venue* are examples for entities whereas *Writes* is a relation. After them, come the attributes of these tables such as *author\_id*, *venue\_id*, *name*, *paper\_title*, some of which are special in terms of primary and foreign key constraints (e.g., *author\_id* and *venue\_id* in *Paper* table, respectively). These two sets of instruments are what makes a relational database structured to efficiently store and retrieve actual values such as *AuthorA*, *PaperC*, *database* shown in Figure 5.2. Considering the values given in Figure 5.2, the tuples having *p1* and *p2* as the value for primary key should be more similar to each other than the tuple with *p3* as primary key due to having the same value as *keywords*. Similarly, in author table the tuples with values *a1* and *a2* should be close to each other in representation space.

The other component, the relations between the entities, has to be considered as well when generating distributed representations. Connections through foreign keys form relation aspect of the relational database. These connections need to be reflected into embeddings. There are two relations depicted in the ER diagram in Figure 5.1, which are between Author-Paper and Venue-Paper, which

Table 5.1: Author Table

author_id	author_name	institution
a1	AuthorA	UniA
a2	AuthorB	UniA
a3	AuthorC	UniB

Table 5.2: Paper Table

paper_id	paper_title	keywords	venue_id
p1	PaperA	database	v1
p2	PaperB	database	v2
p3	PaperC	ML	v2

Table 5.3: Venue Table

venue_id	name	year	place
v1	VLDB	2020	Tokyo
v2	Sigmod	2019	Amsterdam

Table 5.4: Writes Table

author_id	paper_id
a1	p1
a1	p2
a2	p2
a3	p3

Figure 5.2: Example instances for Scholar Schema

have *many-to-many* and *1-to-many* cardinalities, respectively. Depending on the cardinality, there are two approaches to represent a relation when converting them into psychical tables . One is to have a separate table (e.g., Writes) which includes primary keys of corresponding tables as foreign keys (e.g., Author and Paper) whereas the other one is to include only the primary key of one table inside the other as a foreign key (e.g., venue\_id inside Paper). Note that, separate table approach is a must for  $M - M$  relations, while being optional for  $1 - M$  ones.

In an ideal embedding, the tuples from different tables connected through foreign keys should be close to each other in latent space. For instance, the pair of tuples  $a2, p2$  and  $p1, v1$  should be close to each other in terms of their representations. In addition to that, considering relations between the pair of tuples,  $a1, a2$  and  $p2, p3$  are also more similar to each other than the other tuples in their tables due to interacting with the same particular tuple,  $p2$  in Writes and  $v2$  in Paper tables, respectively.

As mentioned earlier, in this thesis, we utilize graph neural networks to train local database embeddings. In order to do that, we first need to extract a graph data out of the database. Regarding the properties mentioned above, we propose two different graph structures on which we employ deep learning to generate representations. In the following subsections, we provide the algorithms we used to create these two different graphs and mention about deep learning methodologies we utilize to train embeddings on these graphs.

### 5.1.1 Classifying Tables as Entity or Relation in Databases

In our solution, we first need to automatically differentiate (i.e., classify) entity and relation tables from each other. Because, we will create nodes for each tuple in an entity table and create edge in between these nodes (i.e., tuples) for a relation tuple or a foreign key connection in the output graph. The pseudo-code for this operation is given in Algorithm 1.

We assume that each table in a relational database is an entity table unless it has the following properties:

- The table has attributes referencing other tables as foreign key.
- The set of attributes as primary key has overlap with the set of attributes referencing other tables as foreign key.

### 5.1.2 Tuple-Tuple-Value (TTV) Graph

As the first graph structure, we propose Tuple-Tuple-Value (TTV) graph in which we store nodes in two different types. First, we store each unique tuple that has a connection with another tuple in a different table (i.e., through primary-foreign key connection) as *Tuple* nodes. This idea of utilizing connections between tuples

```

SetTableType ( $D$ )
|   Input : relational database  $D$ 
|   Output: list of tables  $T$ 
 $T \leftarrow \text{getAllTables}(D)$ ;
foreach table  $t_i \in T$  do
|    $pks \leftarrow \text{getPKAttributes}(t_i)$ ;
|    $fks \leftarrow \text{getFKAttributes}(t_i)$ ;
|    $table\_size \leftarrow \text{length}(t_i)$ ;
|   if  $fks$  is Empty then
|   |    $t_i.type \leftarrow \text{"entity"}$ ;
|   |   break;
|   if  $fks = pks$  then
|   |    $t_i.type \leftarrow \text{"relation"}$  ; // in cases for  $M\_M$  relation tables
|   |   break;
|   foreach attribute  $a_i \in pks$  do
|   |   if  $fks$  contain  $a_i$  then
|   |   |    $t_i.type \leftarrow \text{"relation"}$  ; // in cases for  $1\_M$  relation
|   |   |   tables
|   |   |   break;
|   end
|    $t_i.type \leftarrow \text{"entity"}$ ;
end
return  $T$ ;

```

**Algorithm 1:** Differentiate Entity and Relation Tables in a Database

helps us with exploiting the structural information the graph exhibits. However, the textual information residing in tuples is also important to consider when generating representations. Therefore, we also create another node type, *Value* nodes, for each unique value residing in cells inside tuples. For multi-word values, we simply split the value into multiple tokens each of which becomes a value node in the graph.

The edges between tuple nodes are straightforward. We create an edge for each pair of tuples that has a primary-foreign key relation. For each value node we create, we also draw an edge between that value node and the tuple node in which the value resides in. The example TTV graph extracted from the sample set of instances provided in Figure 5.2 is illustrated in Figure 5.3. Orange nodes represent *tuple* while blue ones represent *value*, which is a unique token in a



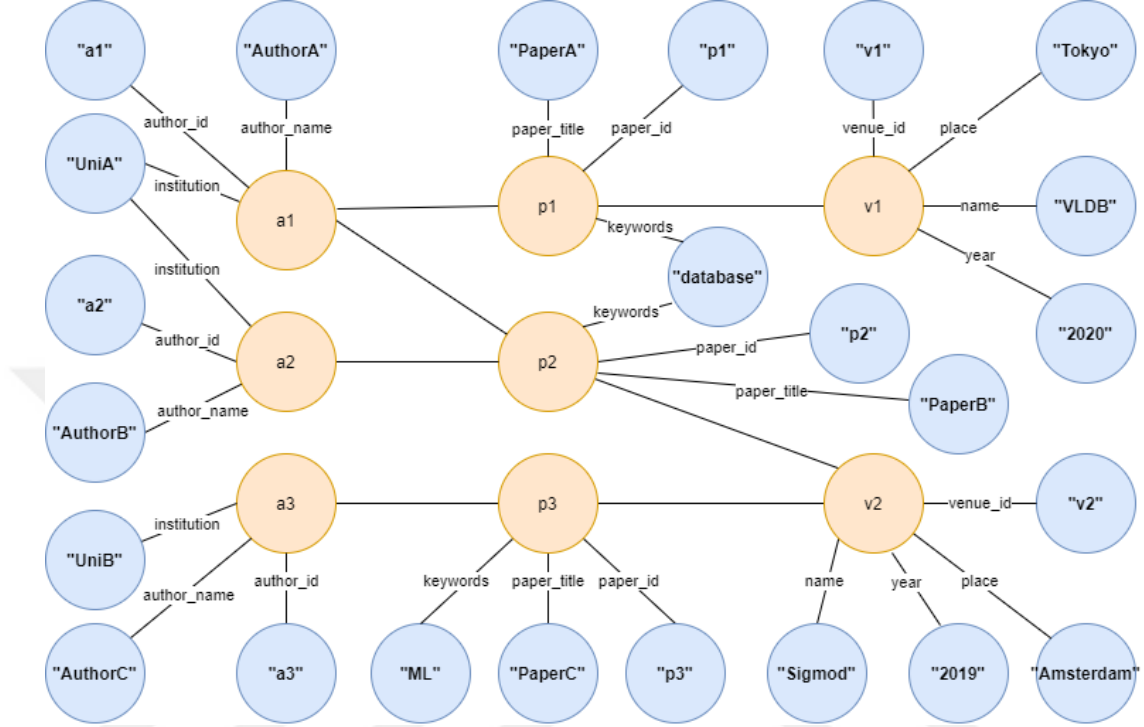


Figure 5.3: Example TTV graph extracted from sample database instances in Figure 5.2

database value. The labels on the edges between blue and orange nodes are to show the attribute that value comes from.

### 5.1.3 Tuple-Tuple (TT) Graph

Similar to the previous one, we propose another graph structure called *Tuple-Tuple (TT)* in which we only store edges for connections between tuples through primary-foreign key relations. As opposed to TTV graph, we do not explicitly utilize textual information (i.e., value nodes) in this graph, yet since textual information is also important we try to store textual signals as node features this time. Instead of a 2-dimensional node features in TTV graph, we first learn node features offline by another state-of-the-art deep learning methodology by only utilizing values residing in the tuples.

After learning textual features for tuples, we feed them into graph neural model

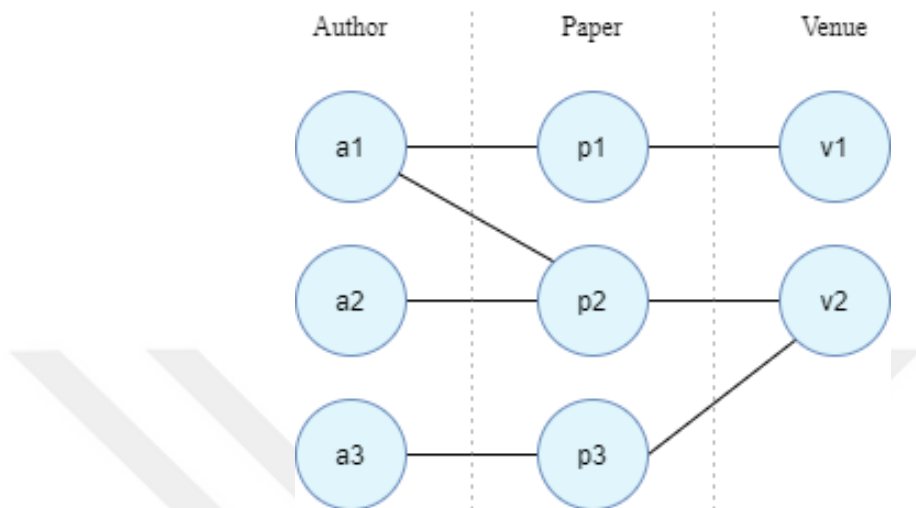


Figure 5.4: Sample TT graph extracted from sample database instances in Figure 5.2

as node features to be used during information propagation among neighbour nodes. In other words, the textual information of the tuples will be stored as vectors to help with training during learning in graph convolutional networks. The example TT graph extracted from the sample set of instances provided in Figure 5.2 is illustrated in Figure 5.4. The graph in a sense becomes a  $k$ -partite graph, where  $k$  refers to the number of entities (i.e., non-relation tables) in the database. The state-of-the-art deep learning techniques we employ for extracting textual representations of tuples to be utilized as node features are discussed later in sub-section 5.1.5.

#### 5.1.4 Node Embeddings with Graph Convolutional Networks

After extracting the graph, we utilize graph convolutional networks for node embeddings as mentioned before. Instead of an unsupervised learning such as [35], we choose to train embeddings in supervised fashion. In order to train a machine learning model in a supervised fashion, we need actual target labels to use during training. For that purpose, we use name of the tables for tuple nodes as target labels as an application of *self supervision*. For words, we do not associate

any target, since a unique token can appear in many tuples from different tables. Therefore, our training setup can be considered as *semi-supervised learning*, since we only use tuple nodes that have actual target labels during training, calculating the loss and back-propagating it through the layers to update weights.

In graph convolutional networks, there are also node features that can be utilized during the convolve operation among neighbourhood nodes. Since we have 2 different type of nodes, we choose to represent node type as node features. We vectorize features in a 2-dimensional vector following the 1-hot encoding approach. For tuple and value nodes we have [10] and [01] feature vectors, respectively.

Formally, the problem of learning representations of nodes in a graph  $G = (V, E)$  where  $V$  and  $E$  represent vertices (i.e., nodes) and edges (i.e., connections) is to learn a function of features on the graph [34]. In a graph neural networks, the input has the following:

- A feature vector  $x_i$  for each node  $i$ ; stored in a matrix  $X$  of dimensions  $N \times D$  where  $N$  represents the number of nodes in the graph and  $D$  refers to the dimension of the feature vector.
- An adjacency matrix or list ( $A$ ) that stores the connections between nodes in the graph.

After training, the GNN model outputs a matrix  $Z$  of dimensions  $N \times F$  where  $F$  represents the dimension of the node representation. Since the model can have multiple layers, every network layer can be written as follows

$$H^{l+1} = f(H^l, A) \quad (5.1)$$

with  $H^0 = X$  and  $H^L = Z$ ,  $L$  representing the number of network layers. The graph neural networks basically differ from each other by employing specific  $f$  functions to propagate information through layers. We utilize a simple propagation rule provided in the original GCN study [34] which is defined as follows

$$f(H^l, A) = ReLU(AH^lW^l) \quad (5.2)$$

where  $W^l$  represents the weight matrix for the layer  $l$  and  $ReLU$  is a non-linear activation function performed on top of the matrix operations.

In our model, we employ 2 layer convolution, which means that during propagation we process all the neighboring nodes that are up to 2 hops away. On top of the last layer, we append a Softmax layer to normalize probabilities for target classes for classification. For training, we use cross-entropy loss which is evaluated as follows

$$L = - \sum_{l \in Y_L} \sum_{k=1}^K Y_{l,k} \ln Z_{l,k} \quad (5.3)$$

where  $Y_L$  represents the set of node indices used in training along with their target labels, and  $Z_{l,k}$  refers to the output representation for a particular node  $l_i$  on which soft-max activation is applied for class (i.e., entity)  $k$ .

### 5.1.5 Hybrid Approach: Combining Graph Neural Networks with External Deep Learning Techniques

In this thesis, we propose two different graph structures, namely TTV and TT. Although TTV graph has textual information integrated into the graph through value nodes and their respective edges, TT graph only has edges between tuples storing neither node nor edge that corresponds to text in the database. In order to compensate it, we propose a hybrid approach which learns textual features of graph nodes externally through another deep learning methodology.

In order to train textual features of the graph nodes, we treat each tuple as a sentence and handle the problem of learning distributed representation as sentence embedding (see Chapter 3). Each tuple is converted to a sentence in

Entity Table	Entity Tuple Label	Sentence
Author	a1	AuthorA UniA
	a2	AuthorB UniA
	a3	AuthorC UniB
Paper	p1	PaperA database
	p2	PaperB database
	p3	PaperC ML
Venue	v1	VLDB 2020 Tokyo
	v2	Sigmod 2019 Amsterdam

Table 5.5: Sample tuple sentences used for textual deep learning for node features

which values of corresponding attributes are appended to each other to form the sentence. While processing the values of the tuples, we discard *NULL* values and the values for foreign key attributes which are already processed in their respective referenced attribute used as primary key. Example sentences extracted from the sample database provided in Figure 5.2 are provided in Table 5.5.

In what follows, we list the state-of-the-art deep learning techniques we employ to learn tuple representations externally and summarize their characteristics:

- **word2vec:** As the first technique, we apply the well known word2vec [22] with SkipGram approach. After extracting the sentences we train the neural model from scratch solely on the set of database tuples. After training is done, for each sentence (i.e., tuple), we look at embeddings for each word (i.e., token of a database value) and calculate average of the embeddings of all words inside the sentence to generate a single vector.
- **doc2vec:** We apply doc2vec [30] as another technique to train sentence representations directly from the sentences generated. doc2vec can be considered as an extension to the word2vec in a sense that, during training another input token is fed into embedding lookup to accompany token embeddings, which eventually will represent the original sentence.
- **fastText:** Both of the above two studies are similar to each other in the way that they train to learn mapping functions for each token or sentence appearing in the training corpus. Therefore, they fail to generate embeddings for not-seen (i.e., OOV) tokens. In order to address that, fastText [58]

tries to exploit sub-word (e.g., character n-grams) information and therefore can generate an embedding for a variable length text, which is applicable to generate sentence embeddings. Similar to the previous studies, we train the model from scratch on database sentences only.

- **BERT:** Transformers [85] are state-of-the art architectures employed in many NLP tasks to encode textual input. One of the most popular studies utilizing transformer architecture is BERT [86]. Using a language modelling objective, BERT encodes a given textual input (e.g., word or sentence), which can be later used in another deep learning as an application of *transfer learning*. In this thesis, we exercise pre-trained BERT models trained on large corpus to generate tuple embeddings. Unlike the previous methods, we do not train the neural model from scratch using our database tuple for BERT. We just simply generate embeddings by applying a forward pass (i.e., inference step) on encoder of the pre-trained model.
- **USE:** Similar to the previous methodology, we also adapt another transformer structure which can generate sentence embeddings, named *Universal Sentence Encoder (USE)* [111]. For training, USE employs multi-task learning on 3 different tasks including both unsupervised next and previous sentence prediction similar to SkipThought [31], an input response task which tries to answer an input query with an answer among possible candidates, and supervised natural language inference problem which is a classification problem of a hypothesis between pair or premises.

## 5.2 SQL Query Recommendation Based on Tuple Representations

In order to generate query recommendations, one needs to define a similarity score between SQL queries. This leads to the question of finding a representation for a single query to perform similarity between queries. As noted in [70], there are mainly 3 different approaches that construct these representations, which are

as follows:

1. The first one is feature-based approach, which focuses on the query structure such as the tables in from clause or predicates in where clause to create representation for a query.
2. Another approach is called witness-based, which basically tries to represent a query in terms of the witnesses (i.e., database tuples) returned in result set of the query.
3. The other approach is access area-based which tries to estimate user's interest in the data space to represent a query.

In this thesis, we follow a witness-based approach focusing on the tuples returned as a result of a query issue. Assume that the result set of tuples  $T$  is returned for a query  $q$ .  $T$  can have as many as  $n$  tuples as a result. Representation of a query  $Z_q$  is a  $d$  dimensional vector in latent space. In order to calculate  $Z_q$ , we use representations of tuples  $t \in T$ . The calculation of  $Z_q$  is as follows;

$$Z_q = \sum_{i=1}^n Z_{t_i} \quad (5.4)$$

where  $Z_{t_i}$  represents embeddings for tuple  $t_i$  in result set  $T$  for query  $q$ . Each tuple embeddings are extracted offline after training the neural model, as explained in the previous sections. Since we have embeddings for each tuple and therefore representation for each query, we store each query and its representation as a pair,  $\langle q, Z_q \rangle$  to serve most similar queries to the current query as a recommendation.

In Figure 5.5, the flowchart of the recommendation algorithm, Conquer, is illustrated. In order to serve recommendations, we store two different look-up tables; the first one is query representations and the second one is tuple representations. Assume that there are  $k$  number of queries issued by the users in the past. For each query, we store its representation, a  $d$  dimensional vector,

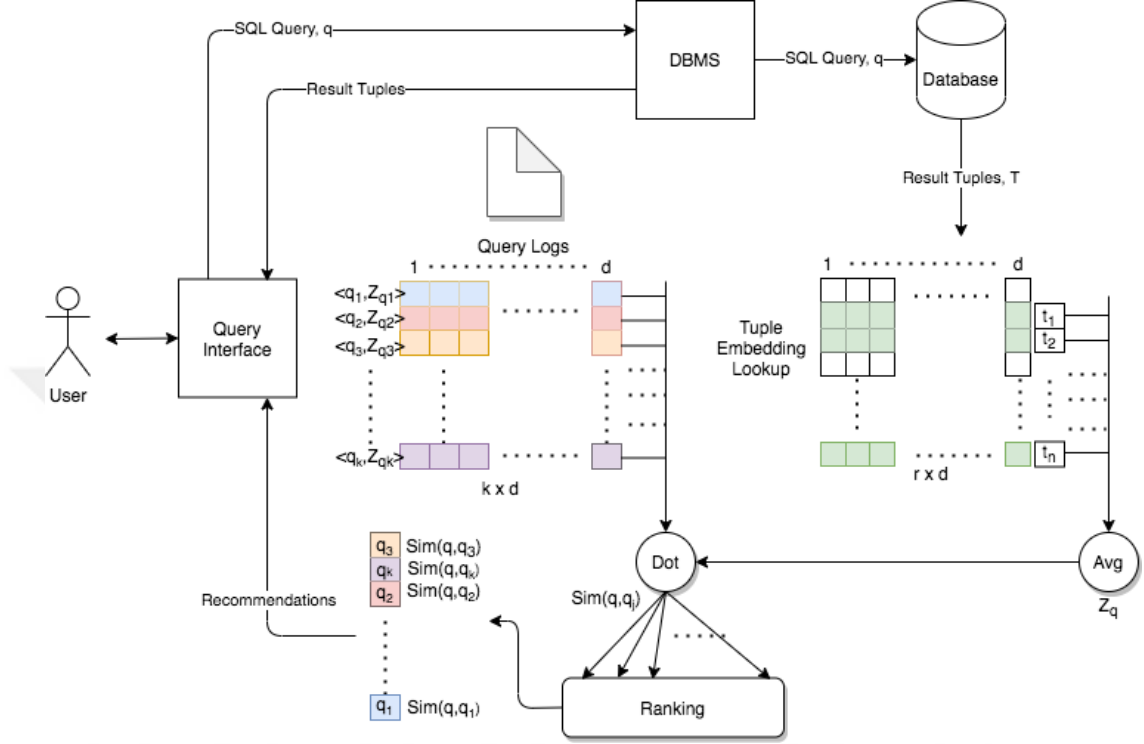


Figure 5.5: Flowchart diagram of the recommendation algorithm, Conquer

therefore the first look-up table is of dimensions  $k \times d$ . For the tuple embeddings, we store a look-up table with dimensions  $r \times d$ , where  $r$  represents the number of tuples residing in entity tables, and again  $d$  represents the number of dimensions in the representation vector. We do not store representations for tuples that are in relation tables whose only purpose is to store the connection between actual tuples in entity tables. You can refer to Section 5.1.1 for further details of how we differentiate entity and relation tables.

After the user issues the query  $q$ , first the returned tuples  $T$  are retrieved from the database. Assume that  $T$  has  $n$  number of tuples. In Figure 5.5, the highlighted rows in the look-up table with green color represent the tuples returned. For each returned tuple  $t_i$ , we find its corresponding embedding from the look-up table (depicted at the right side of Figure 5.5). We then calculate the representation of query  $Z_q$  by averaging all the embeddings for tuples  $t_i \in T$  (see Equation 5.4).

After finding  $Z_q$ , we perform a cosine similarity between vectors  $Z_q$  and  $Z_{q_i}$ ,



corresponding to the user query  $q$  and a past query  $q_i$  from query logs respectively, using the look-up table for query representations. Cosine similarity is a measure of similarity function heavily utilized in information retrieval for finding similarities between pairs of textual inputs [112]. Using the cosine similarity, the similarity value between queries  $q$  and  $q_i$  is calculated as follows;

$$Sim(q, q_i) = \frac{Z_q \bullet Z_{q_i}}{\|Z_q\| \|Z_{q_i}\|} \quad (5.5)$$

where  $\|Z_q\|$  is the length of the vector  $Z_q = (z_1, z_2, \dots, z_d)$ . The cosine similarity basically measures the angle between the vectors, and if the two vectors are close to each other in latent space, the angle gets closer to 0, hence the similarity value becomes closer to 1. After finding the similarity value between queries  $q$  and  $q_i$  where  $i = 1, 2, \dots, k$ , we lastly rank the queries according to their similarity values in descending order and serve the user as recommendations.

# Chapter 6

## Experimental Evaluation

### 6.1 Keyword Mapping in NLIDB

#### 6.1.1 Datasets

In our experiments we used *yelp*, *imdb* [12], and *mas* [10] datasets which are heavily used in many NLIDB related works by the database community [10, 11, 12, 24]. In addition to these datasets, we also used different schemas from the *Spider* dataset [25]; which are *academic*, *college*, *hr*, *imdb*, and *yelp*. Spider is comprised of approximately 200 schemas from different domains, however, there are only handful (around 10) of schemas with more than 100 NL questions. Number of questions is important for our deep learning based solution, since it requires certain number of training data to effectively train. Each schema we picked from the Spider dataset is among the schemas with most number of NL questions, having over 100 queries to work with.

Another limitation of the Spider dataset is that, most of the schemas having enough number of query logs do not have representative number of rows present. For instance, there are no database rows available for *academic*, *yelp*,

Table 6.1: Statistics of the public relational databases used for keyword mapping problem in NLDBs

Properties (#)	Database		
	imdb	mas	yelp
entity tables	6	7	2
relation tables	11	5	5
total tables	17	12	7
total attributes	55	28	38
nonPK-FK attributes	14	7	16
total tags	31	19	20
queries	131	599	128
tokens in queries	1250	4483	1234

Table 6.2: Statistics of the schemas used from the Spider dataset for keyword mapping problem in NLDBs

Properties (#)	Spider				
	academic	college	hr	imdb	yelp
entity tables	7	5	6	6	2
relation tables	8	2	1	11	5
total tables	15	7	7	17	7
total attributes	42	43	35	55	38
nonPK-FK attributes	18	29	21	14	16
total tags	26	36	30	31	20
queries	181	164	124	109	110
tokens in queries	2127	2130	2099	1012	1035

imdb schemas (in Spider), which makes them not applicable for experiments especially for unsupervised approaches that make use of database content such as NALIR or word2vec. In addition to that, college and hr schemas have only around 100 rows, which are not enough to make fair comparison or analysis. Therefore, due to the lack of sufficient database values (many schemas do not have database rows or have few number of rows), we used the Spider dataset only on supervised setup.

The statistics about each public dataset for which annotation is done is shown in Table 6.1. Similarly, statistics for schemas for which annotation is done in the Spider dataset are provided in Table 6.2. In Table 6.1 and 6.2 (referring to Figure

Table 6.3: Accuracy results of neural models with different task weights for fine tuning

Model	task weights			Database		
	pos	type	schema	yelp	imdb	scholar
Single Task	0.0	0.0	1.0	0.9577	0.9061	0.9424
2-Channel Task	0.0	0.1	0.9	0.9500	0.9183	0.9557
	0.0	0.2	0.8	0.9423	0.9265	0.9457
	0.0	0.3	0.7	0.9500	0.9346	0.9502
	0.1	0.0	0.9	0.9576	0.8979	0.9502
	0.2	0.0	0.8	0.9576	0.8979	0.9413
	0.3	0.0	0.7	0.9538	0.9102	0.9457
	0.33	0.33	0.33	0.9538	0.9020	0.9491
3-Channel Task	0.1	0.3	0.6	<b>0.9615</b>	0.9183	0.9413
	0.05	0.2	0.75	0.9500	<b>0.9388</b>	0.9435
	0.05	0.15	0.8	0.9576	0.9142	<b>0.9558</b>

4.1), entity tables refer to main tables (i.e., Movie), relation tables refer to hub tables that store connections between entity tables (i.e., cast, written\_by), nonPK-FK attributes refer to attributes in any table that is neither PK nor FK (i.e., gender in People table), and finally total tags refer to unique number of taggings extracted from that particular schema depending on the above mentioned values. Final schema tags of a particular database are determined by composing table names and name of the nonPK-FK attributes in addition to COND and OTHER. In the last two rows of the both Tables 6.1 and 6.2, we show annotated number of NL questions, referred to as queries, and the number of total words inside these queries, referred to as tokens.

### 6.1.2 Experimental Setup

We first split the datasets into train-validation sets with 5 – 1 ratio, respectively to be used for tuning task weights.

After determining the optimum values for training the model, the last part of

Table 6.4: F1 results of neural models with different task weights for fine tuning

Model	task weights			Database		
				yelp	imdb	scholar
	pos	type	schema			
Single Task	0.0	0.0	1.0	0.9132	0.8169	0.9120
2-Channel Task	0.0	0.1	0.9	0.9209	0.8504	0.9222
	0.0	0.2	0.8	0.9132	0.8476	0.9148
	0.0	0.3	0.7	0.9174	0.8720	0.9234
	0.1	0.0	0.9	0.9082	0.7942	0.9165
	0.2	0.0	0.8	0.9223	0.8207	0.8997
	0.3	0.0	0.7	0.9041	0.8212	0.9105
	0.33	0.33	0.33	0.9090	0.8246	0.9200
3-Channel Task	0.1	0.3	0.6	<b>0.9358</b>	0.8279	0.9126
	0.05	0.2	0.75	0.9181	<b>0.8815</b>	0.9095
	0.05	0.15	0.8	0.9132	0.8356	<b>0.9288</b>

parameter tuning is performed to identify optimal weights for each task. In order to understand the impact of each task for training, we analyzed performance of the models having different weight values for all datasets.

We train our deep neural models using the backpropagation algorithm with two different optimizers, namely Adadelta [113] and Nadam [114]. We start the training with Adadelta and continue it with Nadam. We found that using two different optimizers resulted better in our problem. For both shared and unshared bi-directional GRUs, we use 100 units and apply dropout [115] with the value of 0.5 including recurrent inner states as well. For training, the batch-size is set to 32 for all datasets.

Parameter values chosen are similar to that reported in the study [50] (the state-of-the-art NER solution utilizing deep neural networks), such as the dropout and batch size values. We measure the performance of each neural model by applying cross validation with 6-folds. All the results reported are the average test scores of 6-folds. During inference, we discard POS and Type task results and only use Schema (final) tasks to measure scores.

Table 6.5: Accuracy scores of unsupervised baselines for relation and non-relation matching

Baseline	Database		
	imdb	mas	yelp
tf-idf	0.594-0.051	0.734-0.084	0.659-0.557
NALIR	0.574-0.103	0.742-0.476	0.661-0.188
word2vec	0.625-0.093	0.275-0.379	0.677-0.269
TaBERT	NA-0.251	NA-0.094	NA-0.114
DBTagger	0.908-0.861	0.964-0.950	0.947-0.923

Tables 6.3 and 6.4 present the results in *accuracy* and *f1* scores, respectively. We included the F1 metric as well, because it is highly preferred to be used in similar multi-class classification problems, such as POS tagging and NER in NLP. The first entry in the table represents the single task where only schema tag (actual label) is used in training. *2-Channel Task* represents models learned only on two tags, either pos or type tag in addition to schema tag. *3-Channel Task* finally presents models trained utilizing all three tags as proposed.

We can see that addition of the other tasks (pos and type) improves keyword mapping performance of the models. Regarding *2-Channel Task* models on *imdb* dataset, we observe an improvement especially when we utilize type tag (i.e., the third entry). On the contrary, we see that pos tags are more useful than type tags to improve performance on the *yelp* dataset. We also see consistent improvement on each dataset when we add another task as the second on top of the single task model during training, which supports the idea of utilizing multi-task learning in this problem.

For models trained on multiple tasks, we used  $0.1 - 0.2 - 0.7$  as tuned weights for POS, Type and Schema tasks, respectively. We chose the aforementioned task weights, because they perform well for all schemas in both metrics, accuracy and F1.

### 6.1.3 Comparison with Unsupervised Baselines

We implemented the unsupervised approaches utilized in the state-of-the art NLIDB works for the keyword mapping task as baselines to compare with DB-Tagger. We implemented 3 different unsupervised approaches which are utilized in state-of-the-art NLIDBs in the database community. We implemented sql querying over database column approaches (regex and full text search), which is preferred in NALIR [10]. We implemented a well known tf-idf baseline for exact matching by constructing an inverted index over unique database values present, as in the work ATHENA [11]. We also implemented a semantic similarity matching approach in which pre-defined word embeddings are used. This approach is exercised by Sqlizer [12]. In addition to these unsupervised conventional solutions, we also implemented TaBERT [43], a neural network solution to compare with our proposed solution. TaBERT is also one of the more recent works published by the NLP community working on the NLIDB problem.

- **tf-idf:** Similar to ATHENA [11], for each unique value present in the database, we first create an exact matching index, and then perform tf-idf for tokens in the NLQ. In case of matches to multiple columns, the column with the biggest tf value is chosen as matching. In order to handle multi word keywords, we use n-grams of tokens up to  $n = 3$ . For relation matching, we used lexical similarity based on the Edit Distance algorithm.
- **NALIR:** NALIR [10] uses WordNet, a lexical database in which synonyms are stored for relation matching. They calculate similarity for tokens present in the NLQ over WordNet, and determine a matching if similarity is bigger than a manually defined threshold. For non-relation matching, for each token present in the NLQ, it utilizes regex or full text search queries over each database column whose type is text. In case of matches to multiple columns, the column which returns more rows as a result is chosen as matching. For fast retrieval, we limit the number of rows returned from the query to 2000, as in the implementation of NALIR.

- **word2vec:** For each unique value present in the database, cosine similarity over tokens in the NLQ is applied to find mappings using pre-defined word2vec embeddings. The matching with the highest similarity over a certain threshold is chosen.
- **TaBERT:** TaBert [43] is a transformer based encoder which generates dynamic word representations (unlike word2vec) using database content. The approach also generates column encoding for a given table, which makes it an applicable keyword mapper for non-relation matching by performing cosine similarity over both encodings. For a particular token, matching with maximum similarity over a certain threshold is chosen.

We categorize the keyword mapping task as *relation matching* and *non-relation matching*. The former mapping refers to matching for table or column names and the latter refers to matching for database values. For fair comparison, we do not apply any pre or post processing over the NL queries or use external source of knowledge, such as a keyword parser or metadata extractor. Results are shown in Table 6.5. Each pair of scores represents token wise accuracy for relation and non-relation matching. For TaBERT, we only report for non-relation matching, because the approach is not applicable for relation matching.

DBTagger outperforms unsupervised baselines in each dataset significantly, by up to 31% and 65% compared to best counterpart for relation and non-relation matching, respectively. For relation matching, results of all approaches are similar to each other except the word2vec method for the mas dataset. The main reason for such poor performance is that the mas dataset has column names such as *venueName* for which word2vec cannot produce word representations, which radically reduces chances of semantic matching.

tf-idf gives promising results on the yelp dataset, whereas it fails on the imdb and mas datasets for non-relation matching. This behavior is due to presence of ambiguous values (the same database value in multiple columns) and not being able to find a match for values having more than three words. For the *imdb* dataset, none of the baselines performs well for non-relation matching. The *imdb*



dataset has entity like values that are comprised of multiple words such as movie names, which makes it impossible for semantic matching approaches to generate meaningful representations to perform similarity. NALIR’s approach of querying over database has difficulties for the imdb and yelp datasets since the approach does not solve ambiguities without user interaction.

TaBERT is a recent work that can be applied in many downstream tasks in NLIDB problem, since it generates dynamic encodings for natural language utterances considering the database content as well, which then can be used as input to the various neural models. It is a transformer (BERT) based neural architecture which is trained in an unsupervised fashion. For a given natural language query, it makes a one pass over a particular table to find out most relevant rows (content snapshot) to the given query by using exact matches over n-grams. After finding the most relevant rows, these rows are appended into input to feed the transformer for training. Finally, the model outputs encodings for both the NL utterances and the database columns (called context encoding). In keyword mapping problem, we used these encodings to perform similarity to find a candidate matching over database columns. TaBERT performs poorly for all datasets, which we believe is due to couple of reasons as follows:

- TaBERT has its own tokenizer which is based on BERT base. The tokenizer tries to deal with tokens which are out of vocabulary by breaking the token into sub-words that have representations. This approach might be useful for a language model, but it is problematic in keyword mapping setup, since the values present in the databases are domain specific, which are likely to not occur in the general corpus data used to train such transformers. Also, databases such as imdb, have many entity like values such as "Eternal Sunshine of the Spotless Mind" which is comprised of several words. Such keywords appearing in the natural language query are therefore divided by the tokenizer into pieces, which eventually leads to unrelated word representations and therefore non-predictive similarity calculation.
- The other limitation is using cosine similarity. Such an approach requires a manually defined threshold which is not easy to come up with. When

we decrease the similarity threshold, we increase our chances to find a true positive, however the model becomes prone to generate false positives as well for keywords that are not related to database elements such as stop words, sql specific words (i.e., return, find, minimum, etc.).

We argue that unsupervised baselines may perform reasonable for relation-matching, whereas they fail to answer the challenges raised by non-relation matching. This is due to ambiguity present in the imdb database such as having values that occur in multiple tables (i.e., "Matt Damon") and domain specific values that are not covered in word embeddings (such as word2vec and TaBERT) trained on general corpus data.

#### 6.1.4 Translation Accuracy

The ultimate goal of DBTagger is not to produce tags for POS or Type tasks. We use these tasks to utilize multi task learning to further enhance accuracy of schema (final) tags. During inference, we discard outputs for POS and Type tasks and are only interested in the schema tags. One of the works studied by database community for keyword mapping problem in NLIDB is TEMPLAR [24]. We believe comparison with Templar in terms of translation improvement is not fair, since TEMPLAR is not a standalone keyword mapper but rather an enhancer over an existing NLIDB, which is why we believe they reported the improvement over mapping and translation on their experiments. The biggest difference between TEMPLAR and DBTagger is that, TEMPLAR requires multiple preliminaries to perform enhancement using query logs unlike our solution. DBTagger additionally outputs schema tags for relation matching such as table and column names, which are not covered in TEMPLAR.

On the other hand, we carried out an experiment to further prove efficacy of predicted tags output by DBTagger. We implemented a simple translation pipeline, similar to methodology in TEMPLAR [24]. The pipeline generates join paths for SQL translation using shortest length path over schema graph to cover

Table 6.6: Comparison of Pipeline utilizing tags output by DBTagger with state-of-the-art translation solutions

NLIDB System	Database		
	imdb	mas	yelp
NALIR	0.383	0.330	0.472
TEMPLAR (on NALIR)	0.500	0.402	0.528
DBTagger Pipeline	0.564	0.551	0.461

all the mappings output by DBTagger. We count inaccurate, if the algorithm cannot output a joining path. We compare our pipeline with a state-of-the-art system, NALIR[10], and TEMPLAR[24], which is an enhancer over an existing NLIDB system.

We excluded the Pipeline baseline in TEMPLAR [24] study, since authors hand-parsed the keywords and the associated metadata to perform enhancement. These two preliminaries are the most important challenges yielded by the keyword mapping problem. Parsing keywords require detecting multi-word entities. For a particular parsed keyword, associated metadata includes matched database column along with the sql predicate (i.e., select, where). They assume that given a natural language query, Pipeline system outputs this pair correctly for this baseline. Then, TEMPLAR performs semantic similarity between these pairs with candidate query fragments extracted from query logs to find mappings. These two important challenges are addressed in DBTagger implicitly during training without any external interference. In fact, the authors of [24] state that their scope of the paper is different than ours such that they focus on utilizing query logs to enhance mapping for non-relation matching and translation, and therefore in order to ensure effective experimental setup they provide hand parsed keywords and metadata into Pipeline not to deal with any parser-related performance issues outside the scope of their work.

The results are presented in Table 6.6. The pipeline over DBTagger tags outperforms both systems in imdb and mas datasets, up to 66% and 37% compared to NALIR and TEMPLAR, respectively. For queries which do not include nested or group by constraints such as simple select-join queries, our pipeline produces 67%,

77% and 53% translation accuracy for imdb, mas and yelp datasets, respectively. Considering the simplicity of the translation algorithm, results demonstrate the efficacy of predicted outputs of DBTagger.

### 6.1.5 Impact of DBTagger Architecture

In this experimental setup, we perform keyword mapping in a supervised fashion with different neural network architectures along with a non-Deep Learning (DL) baseline to evaluate architectural decisions.

- **CRF:** As a non-DL baseline, we use vanilla CRF. Semantic word representations of the NLQ are fed as input to the model.
- **ST\_Uni:** We create a two layer stack of uni-directional GRUs, followed by CRF as the classification layer. This model is trained on only a single task, schema tags.
- **ST\_Bi:** Different than the previous architecture, we use bi-directional GRUs instead of uni-directional GRUs. Classification is done on the CRF layer.
- **MT\_Seq:** In this model, training is performed on all three tasks. However, each task is trained separately. The predicted tag of the previous task is fed into the next task. To do that, 1-hot vector representations of predicted tags are concatenated with semantic word representations. We stack a bi-directional GRU with a uni-directional GRU to encode the sentence and feed the output vector to the CRF layer.
- **DBTagger:** This model represents the DBTagger architecture where all tasks are used during training concurrently. DBTagger also has cross-skip connections between tasks as depicted in Figure 4.5.

Table 6.7: Performance of Neural Models with Different Architectures in accuracy-F1 metrics for public databases

Model	Database		
	yelp	imdb	mas
CRF	0.934-0.890	0.907-0.850	0.955-0.932
ST_Uni	0.939-0.883	0.905-0.805	0.961-0.938
ST_Bi	0.947-0.908	0.917-0.832	0.964-0.941
MT_Seq	0.938-0.886	0.921-0.853	0.964- <b>0.943</b>
DBTagger	<b>0.968-0.938</b>	<b>0.935-0.878</b>	<b>0.965-0.941</b>

Table 6.8: Performance of Neural Models with Different Architectures in accuracy-F1 metrics for schemas in the Spider dataset

Model	Spider				
	academic	hr	college	imdb	yelp
CRF	<b>0.974-0.956</b>	<b>0.881-0.748</b>	0.878-0.721	0.866-0.821	0.880-0.827
ST_Uni	0.962-0.945	0.844-0.642	0.854-0.692	0.848-0.751	0.865-0.803
ST_Bi	0.966-0.952	0.877-0.689	0.872-0.720	0.882-0.811	0.891-0.841
MT_Seq	0.964-0.952	0.835-0.685	0.886-0.714	0.896-0.837	0.895-0.838
DBTagger	0.965-0.954	0.861-0.735	<b>0.904-0.761</b>	<b>0.898-0.855</b>	<b>0.897-0.854</b>

For all the models, the same hyper parameters are used for fair comparison during training, as explained in Section 6.1.2. The results are shown in Tables 6.7 and 6.8. Each pair of scores represents the accuracy and F1 measures, respectively. DBTagger performs better than the other supervised architectures for six different datasets in accuracy and in terms of F1. Especially for the yelp and college datasets the performance improvement is remarkable, which is up to around 4.5% and 5%, respectively. Vanilla CRF performs well among all (best in two datasets), which signifies its role in the architecture for the sequence tagging problem. ST\_Bi performs better than ST\_Uni in all datasets, which shows the positive impact of bi-directional GRUs. Compared to single task models, multi task models perform better for all datasets. Except the *mas* dataset for the F1 metric, DBTagger produces better tags compared to the other multi task model, MT\_Seq, in which tasks are trained separately.

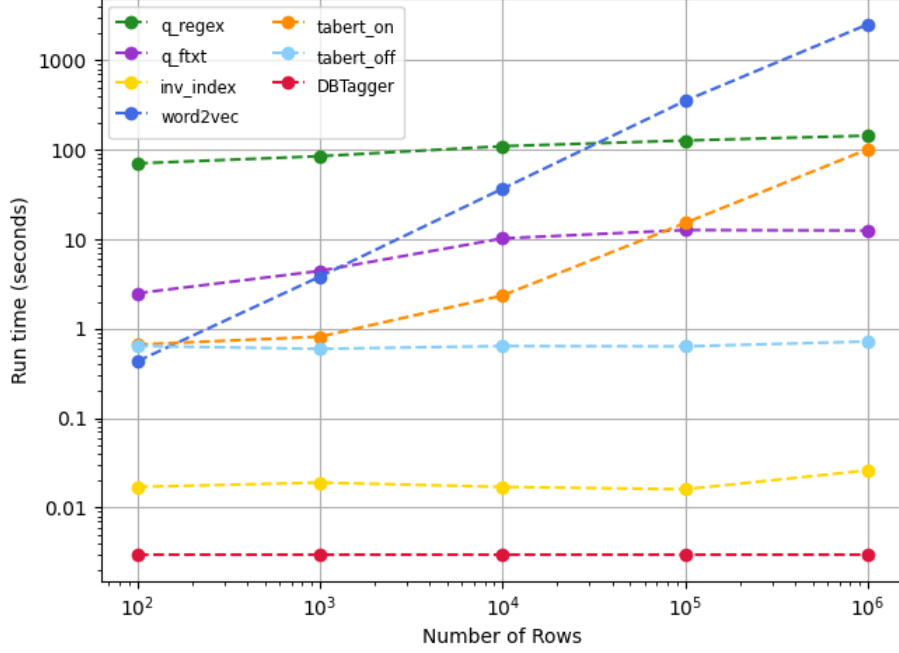


Figure 6.1: Run time comparison of DBTagger with unsupervised state-of-the-art keyword mapping approaches

### 6.1.6 Efficiency Analysis

Efficiency is one of the most important properties of a good keyword mapper to be deployable in online interfaces. Therefore, run time performance of keyword mapping approaches mentioned in Section 3.3 is also evaluated.

- **NALIR:** We analyze both querying over database column approaches used in NALIR[10], named as *q\_regex* and *q\_ftext*, which use *like* and *match against* operators, respectively. The *match against* operator queries over database columns to perform mapping. NALIR [10] prefers this approach for bigger tables with more than 2000 rows. Downside of this approach is the requirement to create full text indexes on each database column with text type. Both query based approaches are used in NALIR [10].

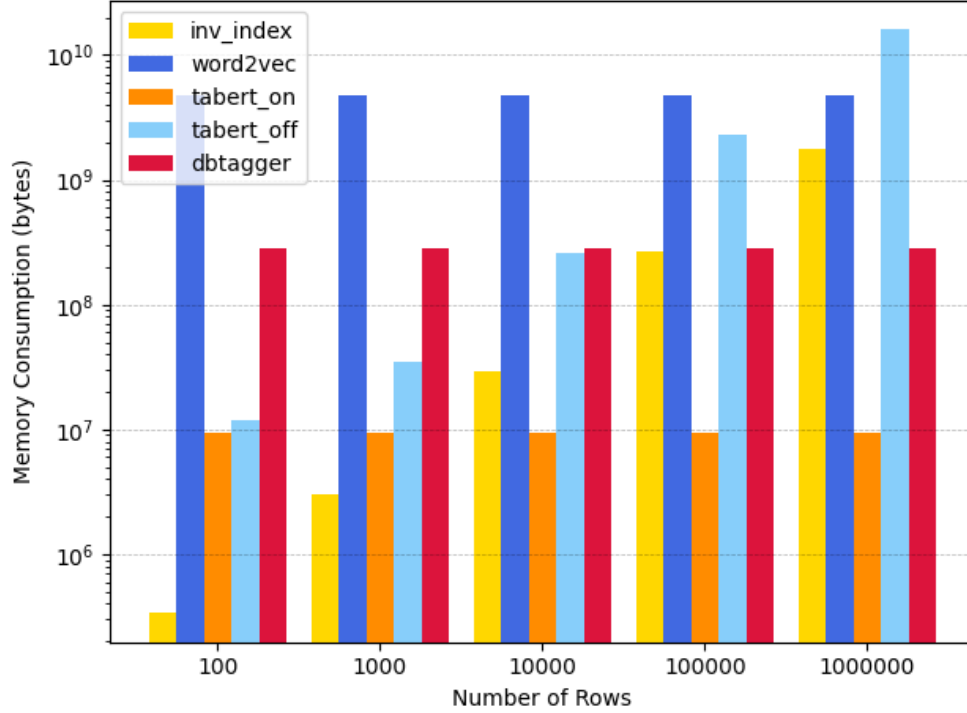


Figure 6.2: Memory usage comparison of DBTagger with unsupervised state-of-the-art keyword mapping approaches

- **tf-idf**: Similar to ATHENA [11], we created an exact matching index, using inverted index named as *inv\_index*, beforehand to avoid querying over database. Inverted index stores each unique value present in the database along with their frequency in each candidate collection (i.e., database columns).
- **word2vec**: Many works such as Sqlizer [12] make use of pre-trained word embeddings to find mappings, which requires keeping the model in the memory to perform similarities.
- **tabert\_on**: TaBert requires database content (content snapshot) to generate encodings for both NL tokens and columns. We call this setup tabert online, where the model generates the content snapshot to perform mapping when the query comes.
- **tabert\_off**: We also use TaBert in offline setup. For each table, database content is generated beforehand to perform encodings. In this setup, we

keep the content in the memory to serve the query faster.

We measured the time elapsed for a single query to extract tags and the memory consumption needed to perform mapping for each approach. We also run each experiment with different number of row values to capture the impact of the database size. Figures 6.1 and 6.2 present run time and memory usage analysis of keyword mappers, respectively. DBTagger outputs the tags faster than any other baseline and it is scalable to much bigger databases. However, `q_regex`, `q_ftext`, `tabert_on` and `word2vec` do not seem applicable for bigger tables having more than 10000 rows. The `tf-idf` technique has nice balance between run-time and memory usage, but it is limited in terms of effectiveness (Table 6.5). `tabert-off` performs the tagging in a reasonable time, yet it requires huge memory consumption especially for bigger tables. Although, query over database approaches do not occupy memory too much compared to other approaches, they fail to perform in reasonable time when number of rows in the database gets bigger than 1000.

## 6.2 SQL Query Recommendation in Databases

### 6.2.1 Datasets

In our experiments, we used 2 different datasets; which are namely *geography* [116] and *college* schema from Spider [25] dataset. These datasets have been utilized widely in many research studies in the database community [20, 25, 39]. Statistics of the datasets used in the experiments are provided in Table 6.9. Tuple nodes and word nodes represent the number of nodes for entity tuples in the database and the tokens residing inside these tuples, respectively in the graph. Note that word nodes are utilized only in the TTV graph structure. Last column of the table represents the number of entity tables, which refer to actual target classes used for nodes in supervised learning for the node classification problem.



Table 6.9: Statistics of the public datasets used for the query recommendation algorithm

Dataset	tuple nodes	word nodes	total nodes	total edges	entity tables
geography	704	1193	1846	4825	6
college	2420	4327	6747	49046	8

### 6.2.2 Experimental Setup

For supervised classification of nodes, we first need to split the datasets into train-test splits for which we used 1 – 9 ratio, respectively. For training, we further split the test split into validation and test sets with 1 – 1 ratio. Similar to the parameters provided in the original GCN paper [34], we constructed graph neural network with 2 layers whose unit sizes are determined as 64 and 32. For both layers, we used *Relu* activation function on the units. We apply dropout to the input features for each layer with 0.5 as value. We train our GCN model with back-propagation algorithm with *Adam* [117] optimizer on the *categorical cross-entropy* loss with a learning rate of 0.001. For the classification, we append a dense layer along with a softmax layer to produce class probabilities. Overall architecture of the GCN model is schematically illustrated in Figure 6.3.

In order to implement the GCN model, we use a public framework called *Stellar Graph* to train graph neural networks on graph data. Stellar Graph uses a well-known high level API named *Keras* with *tensorflow* as back-end. In order to implement the other baselines which train sentence embeddings for database tuples to be utilized in the hybrid approach with TT graph (see Section 5.1.5 for further details), we use *Gensim* framework for conventional word embeddings which are word2vec, doc2vec and fasttext. For BERT implementation, we use a public repository implementing a state-of-the-art study [118] and for USE we implement embeddings using their pre-trained models available at *tensorflow-hub*, an official public repository where the authors of the studies working on language models publish their pre-trained model for other researchers.

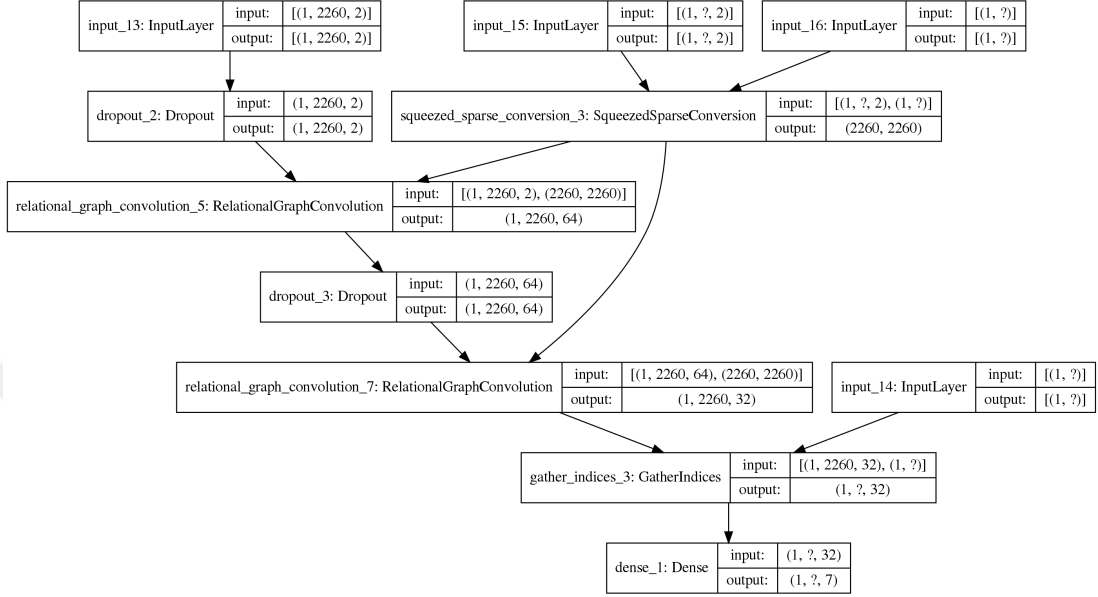


Figure 6.3: Graph Convolutional Network Model Architecture

### 6.2.3 Tuple Classification

As the first experiment, we try to determine how effective tuple embeddings are for tuple classification with target labels indicating their tables they appear in, similar to study [119]. We evaluate both graph structures we propose, TT and TTV (i.e., tuple-tuple and tuple-tuple-value graphs, respectively) in the classification of tuples. For node features in TT graph, as explained in Section 5.1.5, we implement 4 different baselines which are word2vec, doc2vec, fastText, and USE. For the first 3 methods, we first train the embeddings on the database tuples (see Section 5.1.5 for details about how to extract tuple sentences out of the database) from scratch. For the latter, that is based on transformer architecture [85], we simply load the pre-trained model from tensorflow-hub to encode tuple sentences into vector representations.

For the baselines, we calculate the accuracy and F1 metrics using Logistic Regression classifier, similar to classification setup in [119]. In order to make fair comparison with proposed graph structures, we used the same ratio, 1 – 9, and employ 10 fold cross validation. Reported results for both metrics are average scores across all folds. For word2vec, doc2vec and fastText, the models are trained

on 100 epochs, after which the gaining is observed to be minimal.

In what follows, we give details about the parameters chosen for the baselines mentioned for node feature extraction:

- **word2vec:** We use skip-gram approach of the word2vec model. As a window size, we use 3 to determine neighbouring tokens for a target word. For dimension of the embeddings, we employ 500 to make it close to the dimension USE extracts for fair comparison.
- **doc2vec:** Similar to the parameters mentioned above, we apply doc2vec with the *PV-BOW* (distributed bag of words) approach.
- **fastText:** Window and dimension parameters are determined as with the previous baselines. For training, we employ the *CBOW* (continuous bag of words) approach. For character n-grams, minimum and maximum n-gram characters (i.e., sub-words) are determined to be 3 and 6, respectively.
- **USE:** As we mentioned earlier, we do not train a model for this baseline, we simply encode tuples utilizing a pre-trained model made available by the authors. We choose a model that produces embeddings in 512 dimensions.

After extracting tuple representations for the aforementioned approaches, we employ logistic regression to classify tuples, similar to the work presented in [119]. We later utilize these embeddings as node features in the graph convolutional network (gcn) model for TT graphs. All the results for 3 different setups (i.e., baselines, TT and TTV graphs) in 2 datasets are shown in Table 6.10. For each dataset, we report results in accuracy and F1 metrics, which are broadly preferred in a multi-class classification problem [119]. In the Table 6.10, baselines and TT graphs are combined into rows, since in TT graph setup we utilize initial embeddings extracted by a particular baseline as node features, which is indicated by *+gcn*.

In the geography and college datasets, the best performing models are our

Table 6.10: Accuracy-F1 results of tuple classification using tuple embeddings for datasets along with baselines

Dataset	geography		college	
	accuracy	f1	accuracy	f1
word2vec	0.5591	0.1513	0.9309	0.4007
+ gcn (TT)	0.5710	0.5244	<b>0.9541</b>	0.9617
doc2vec	0.5539	0.1500	0.8652	0.2770
+ gcn (TT)	0.5647	0.5128	0.8641	0.8870
fastText	0.5482	0.1180	0.9006	0.3017
+ gcn (TT)	0.5931	0.5231	0.9376	0.9598
USE	0.5708	0.1936	0.9022	0.3540
+ gcn (TT)	0.6418	0.6496	0.9513	<b>0.9630</b>
gcn (TTV)	<b>0.8810</b>	<b>0.7455</b>	0.9513	0.9374

proposed graphs; which are TTV and TT respectively. While the baselines perform close to our proposed solutions in the college dataset, they perform poorly in the geography dataset for which the TTV approach outperforms counterparts by up to 60% and 570% in accuracy and F1 metrics, respectively. The baselines perform poorly in both datasets in terms of the F1 metric, which is especially important in setups where class distribution is not balanced. The class distribution of database tuples is naturally not balanced, which further signifies the results in the F1 metric for the models.

Another important observation is that, for all baselines, TT graph models enhance the baseline embeddings for the classification problem for both datasets, which highlights the efficacy of graph usage for database tuples. The improvement in the F1 metric is much higher up to 220% (i.e., compared to the doc2vec baseline for the college dataset).

The reason why there is a performance difference between the datasets is we believe 2-folds. The geography dataset is more ambiguous such that multiple values occur in multiple tables which limits efficacy of the baselines and consequently the TT models, which depend on textual representations of the tuples. However, TTV model outperforms the best baseline + TT model by 37% in the geography dataset, which shows that it is better to resolve ambiguities. The other reason

why models perform better for the college dataset is that the number of tuples is higher than that in the geography dataset (Table 6.9).

#### 6.2.4 t-SNE Visualization of Database Tuples

Another indicator of quality of the embeddings output from deep learning models is visualization of the embeddings in latent space. Since, embeddings are high dimensional, a dimensionality reduction technique is applied before visualization such as t-SNE. Similar to the work [119], in this experimental setup we report t-SNE visualizations of the embeddings of the approaches mentioned in the previous section. Ideally embeddings of the tuples with the same target labels (i.e., tables in the databases) should be plotted close to each other while being separated from the other tuples with different target labels.

t-SNE plots of embeddings for each approach are depicted in Figures 6.4 and 6.5 for the geography and college datasets, respectively. It can be seen from Figure 6.4 that baseline approaches cannot differentiate tuples from each other effectively, while TT graph models on top of them performs better separating the tuples from each other in terms of the tables they reside in.

Reflecting the accuracy and F1 results provided in Table 6.10, both graph models, TT and TTV, produce embeddings which are visually well enough separated from each other to represent their target labels in a group. However, although the separation is evident, embeddings from baseline approaches are worse compared to both graph models in terms of visualization.

#### 6.2.5 Generated Recommendations

After extracting embeddings for tuples, we utilize these embeddings in a recommendation task. First, we find the tuples retrieved for a particular query and calculate query representation by averaging embeddings of tuples returned. For

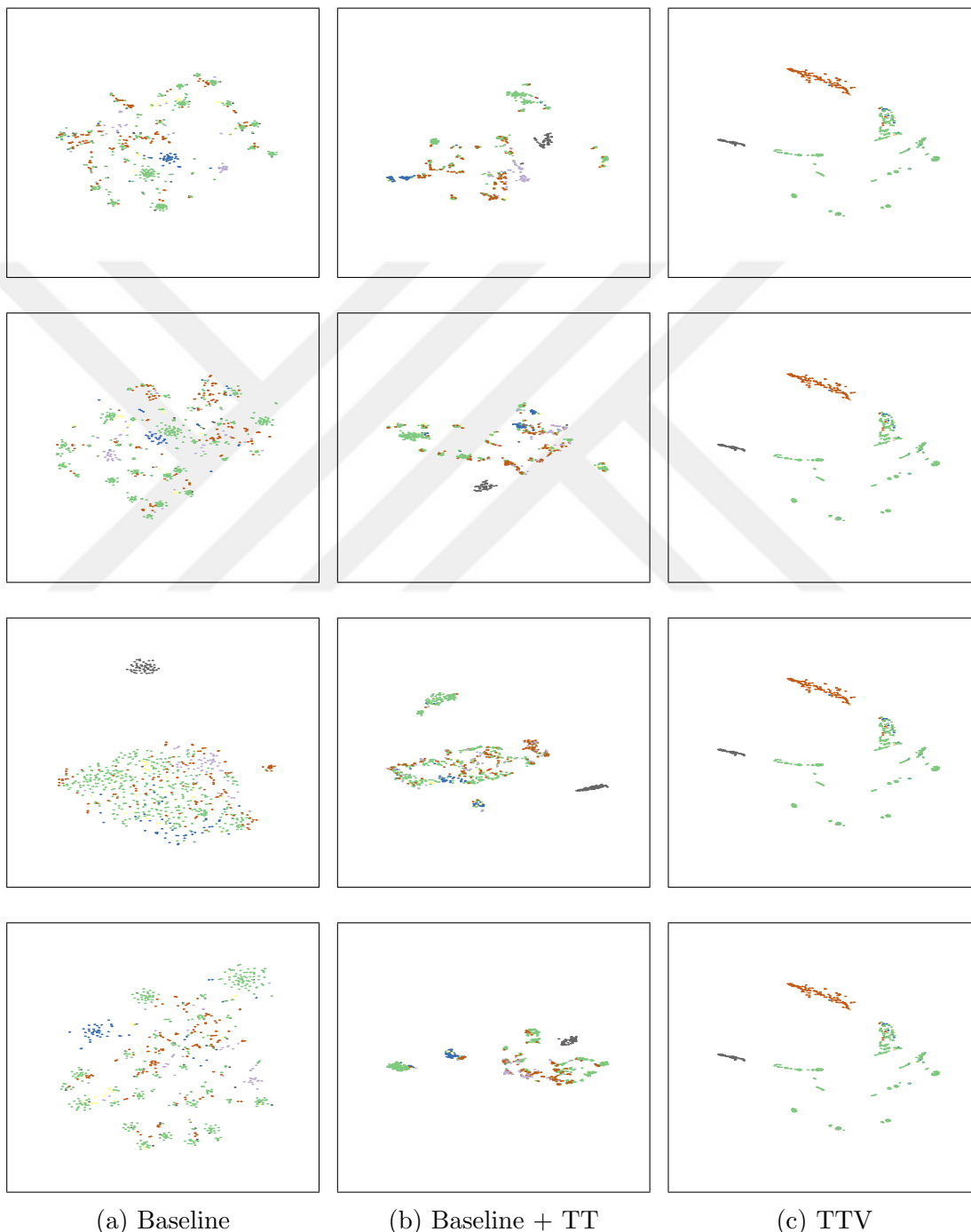
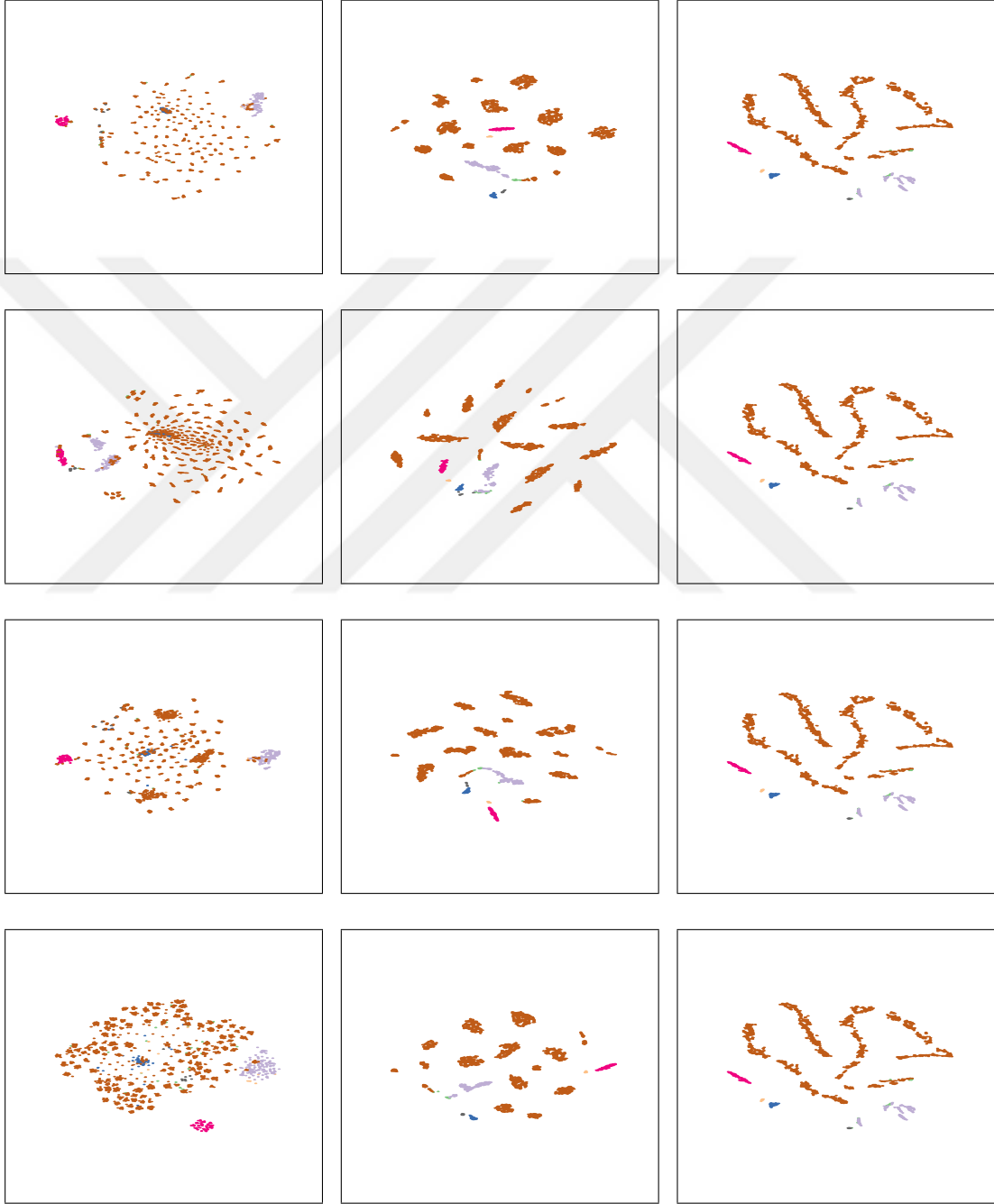


Figure 6.4: tSNE visualization of the models in the geography dataset. For each baseline, plots are depicted side-by-side. From top to bottom, each row of plots represents word2vec, doc2vec, fastText, and USE baselines, respectively



(a) Baseline

(b) Baseline + TT

(c) TTV

Figure 6.5: tSNE visualization of the models in the college dataset. For each baseline, plots are depicted side-by-side. From top to bottom, each row of plots represents word2vec, doc2vec, fastText, and USE baselines, respectively

this experiments, we again use geography and college datasets. For the embeddings, we perform experiments using TTV graph model for convenience. For geography and college datasets, we use 480 and 60 queries, respectively. In our experiments, we only process queries that do not include join operations.

First, we visualize the queries in the latent space using t-SNE dimensionality reduction technique along with all the accessed tuples from entire query logs processed. The visualizations for geography and college datasets are depicted in Figures 6.6 and 6.7, respectively. It is evident from the figures that queries are placed close to tuples retrieved in latent space, which shows that semantic representation of queries using tuple embeddings is promising to represent a query. Compared to college dataset, queries are distributed evenly in the latent space following the tuples retrieved.

In addition to visualizations of the queries, we also provide example pairs of most similar queries in both datasets in Table 6.11. Although there is still room for improvement in terms of query recommendations, it is evident from the examples that, embeddings are effective to capture query similarities.



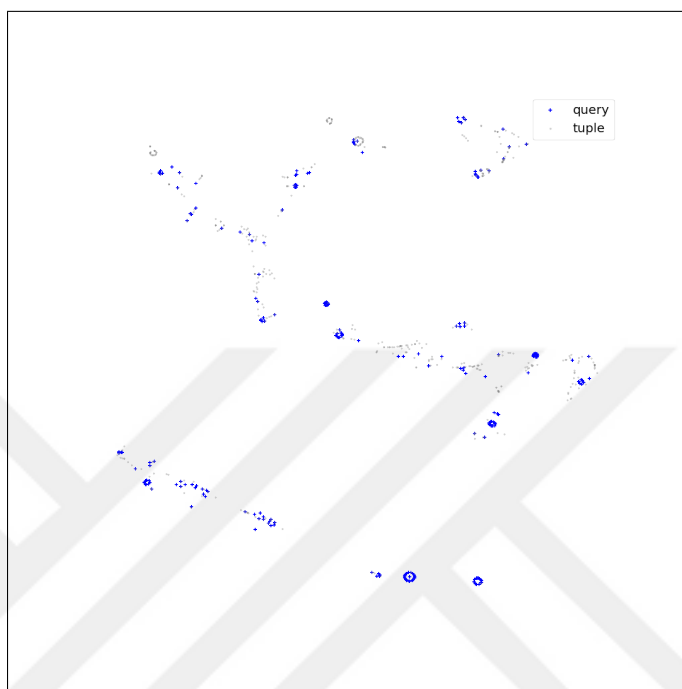


Figure 6.6: Visualization of the queries along with accessed tuples for geography dataset

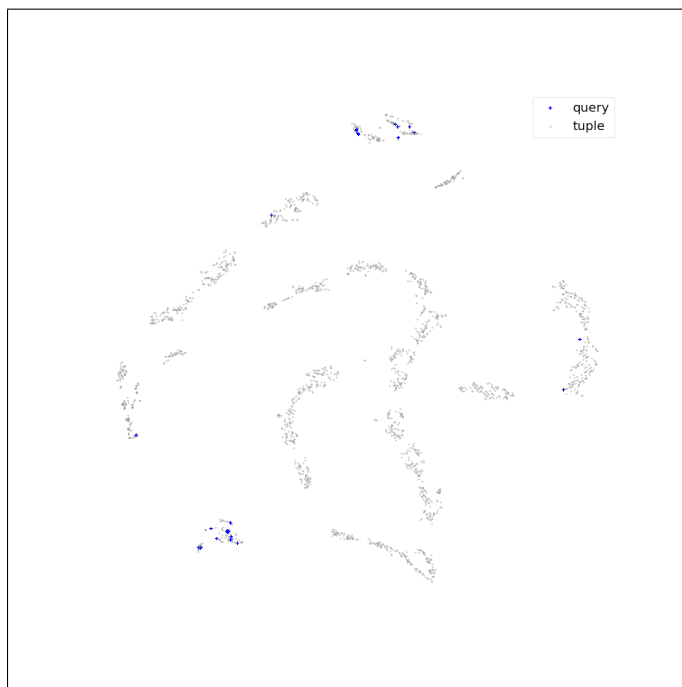


Figure 6.7: Visualization of the queries along with accessed tuples for college dataset

Table 6.11: Example pairs of most similar queries from both datasets

geography	SELECT traverse FROM river WHERE river_name = "ohio"	SELECT COUNT ( river_name ) FROM river WHERE river_name = "colorado"
	SELECT border FROM border_info WHERE state_name = "kentucky"	SELECT border FROM border_info WHERE state_name IN ( SELECT traverse FROM river WHERE river_name = "mississippi" )
	SELECT population FROM city WHERE city_name = "seattle" AND state_name = "washington"	SELECT state_name FROM city WHERE population = ( SELECT MAX ( population ) FROM city WHERE state_name = "montana" ) AND state_name = "montana"
college	SELECT count(*) FROM classroom WHERE building != "Lamberton"	SELECT DISTINCT building FROM classroom WHERE capacity > 50
	SELECT dept_name , building FROM department WHERE budget > (SELECT avg(budget) FROM department)	SELECT sum(budget) FROM department WHERE dept_name = "Marketing" OR dept_name = "Finance"
	SELECT name FROM student WHERE dept_name = "History" ORDER BY tot_cred	SELECT name FROM instructor WHERE dept_name = "Comp. Sci."

# Chapter 7

## Conclusion

As the first work of this thesis, we present DBTagger, a keyword mapper to be used in translation pipelines in NLIDB systems. DBTagger is a standalone system which does not require any processing or external knowledge such as parser or metadata preliminaries. Inspired by sequence tagging architectures used for well known problems such as part-of-speech (POS) in the NLP community, DBTagger utilizes a deep neural architecture based on bi-directional Gated Recurrent Units (GRUs). We try to exploit the observation that POS tags of keywords are related to schema tags by applying multi-task learning in our architecture. In addition to multiple tasks on which training is done, we also made use of the skip-connection technique, well known especially in architectures used in computer vision.

DBTagger provides the best accuracy results on three publicly available databases and five schemas in the Spider dataset, producing keyword tags with 92.4% accuracy on the average over all the datasets within 3 milliseconds, which is up to 10000 times faster than unsupervised approaches. Our results also show that DBTagger is scalable to large databases containing millions of rows.

We believe that DBTagger can be applied in existing NLIDB systems as the first step to improve translation, especially in pipeline-based systems. For deep learning based approaches, DBTagger can be utilized to be augmented on neural

network to enrich input query before feeding into network. DBTagger trains keyword mapper in a supervised fashion, which naturally depends on the quality of training data in terms of distribution of target labels. In addition to that, although it does not require heavy labor work to annotate training data, a handful of queries (i.e., more than 100 queries) is required, which can be stated as the limitations of the proposed solution.

As the second study, we propose Conquer, a contextual query recommendation algorithm utilizing graph neural networks for local embeddings of database tuples. In order to train representations, we extract two graph structures, Tuple-Tuple (TT) and Tuple-Tuple-Value (TTV), from a relational database. We employ node classification in the graphs to embed nodes (i.e., tuples of the database) into latent space. We follow a paradigm called self-supervised learning to associate database tuples with targets to train embeddings in supervised fashion. We introduce an algorithm to differentiate entity and relation tables from each other where entity tables are utilized as target labels in the training.

For comparison, we implement 4 different state-of-the-art baselines in NLP community to extract embeddings based on sentences. We evaluated our proposed graph learning approaches, TT and TTV, in 2 different datasets. Our results indicate that our graph models trained using graph convolutional networks outperform all baselines in both datasets by up to 60% and 570% in accuracy and F1 metrics, respectively. In addition to that, TT graph structure further improves embeddings extracted by the baseline approaches for both datasets, which indicates that our hypothesis that relational data can be exploited better in a graph holds. t-SNE visualizations of the output embeddings highlight that TTV is better to differentiate tuples from each other according to their tables. Query recommendations based on embeddings extracted by TTV graph are also promising, showing that semantic representation of the tuples is a good indicator for query similarity.

As a future work, we plan to integrate queries directly into the graph by introducing extra nodes and edges representing connections between the queries and the tuples retrieved by them. In addition to that, we hope to introduce

another objective other than table names as target labels to better extract tuple representations tailored for the recommendation task. Due to lack of publicly available datasets for recommendation task in databases, we plan to annotate queries available in the databases in terms of relevancy to better evaluate performance of Conquer. Quality of the recommended queries depends on availability of such queries in the log, which can be considered as the limitation of our approach.



# Bibliography

- [1] H. Li, *Learning to Rank for Information Retrieval & Natural Language Processing*. San Rafael, CA, USA: Morgan & Claypool, 2011.
- [2] A. Usta, I. S. Altingovde, I. B. Vidinli, R. Ozcan, and Ö. Ulusoy, “How k-12 students search for learning? analysis of an educational search engine log,” in *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*, pp. 1151–1154, 2014.
- [3] A. Usta, I. S. Altingovde, R. Ozcan, and Ö. Ulusoy, “Re-finding behaviour in educational search,” in *Proc. 23rd Int. Conf. Theory & Pract. Digit. Libraries*, pp. 401–405, Sept. 2019.
- [4] S. K. Karmaker Santu, P. Sondhi, and C. Zhai, “On application of learning to rank for e-commerce search,” in *Proc. 40th Int. ACM SIGIR Conf. Res. & Develop. Inf. Retrieval*, pp. 475–484, August 2017.
- [5] R. Baeza-Yates, C. Hurtado, and M. Mendoza, “Query recommendation using query logs in search engines,” in *Proceedings of the 2004 International Conference on Current Trends in Database Technology*, EDBT’04, (Berlin, Heidelberg), p. 588–596, Springer-Verlag, 2004.
- [6] S. Mohan, N. Fiorini, S. Kim, and Z. Lu, “Deep learning for biomedical information retrieval: Learning textual relevance from click logs,” in *BioNLP 2017*, (Vancouver, Canada,), pp. 222–231, Association for Computational Linguistics, Aug. 2017.

- [7] A. Gordo, J. Almazán, J. Revaud, and D. Larlus, “Deep image retrieval: Learning global representations for image search,” in *European conference on computer vision*, pp. 241–257, Springer, 2016.
- [8] A. Usta, I. S. Altingovde, R. Ozcan, and Ö. Ulusoy, “Learning to rank for educational search engines,” *IEEE Transactions on Learning Technologies*, vol. 14, no. 2, pp. 211–225, 2021.
- [9] L. Blunski, C. Jossen, D. Kossmann, M. Mori, and K. Stockinger, “Soda: Generating sql for business users,” *Proc. VLDB Endow.*, vol. 5, p. 932–943, June 2012.
- [10] F. Li and H. V. Jagadish, “Constructing an interactive natural language interface for relational databases,” *Proc. VLDB Endow.*, vol. 8, p. 73–84, Sept. 2014.
- [11] D. Saha, A. Floratou, K. Sankaranarayanan, U. F. Minhas, A. R. Mittal, and F. Özcan, “Athena: An ontology-driven system for natural language querying over relational data stores,” *Proc. VLDB Endow.*, vol. 9, p. 1209–1220, Aug. 2016.
- [12] N. Yaghmazadeh, Y. Wang, I. Dillig, and T. Dillig, “Sqlizer: Query synthesis from natural language,” *Proc. ACM Program. Lang.*, vol. 1, pp. 63:1–63:26, Oct. 2017.
- [13] S. Iyer, I. Konstas, A. Cheung, J. Krishnamurthy, and L. Zettlemoyer, “Learning a neural semantic parser from user feedback,” in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, (Vancouver, Canada), pp. 963–973, Association for Computational Linguistics, July 2017.
- [14] V. Zhong, C. Xiong, and R. Socher, “Seq2sql: Generating structured queries from natural language using reinforcement learning,” *arXiv preprint arXiv:1709.00103*, 2017.
- [15] X. Xu, C. Liu, and D. Song, “Sqlnet: Generating structured queries from natural language without reinforcement learning,” *arXiv preprint arXiv:1711.04436*, 2017.

- [16] T. Yu, M. Yasunaga, K. Yang, R. Zhang, D. Wang, Z. Li, and D. Radev, “SyntaxSQLNet: Syntax tree networks for complex and cross-domain text-to-SQL task,” in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, (Brussels, Belgium), pp. 1653–1663, Association for Computational Linguistics, Oct.-Nov. 2018.
- [17] N. Weir, P. Utama, A. Galakatos, A. Crotty, A. Ilkhechi, S. Ramaswamy, R. Bhushan, N. Geisler, B. Hättasch, S. Eger, U. Cetintemel, and C. Binnig, “Dbpal: A fully pluggable nl2sql training pipeline,” in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’20, (New York, NY, USA), p. 2347–2361, Association for Computing Machinery, 2020.
- [18] F. Özcan, A. Quamar, J. Sen, C. Lei, and V. Efthymiou, “State of the art and open challenges in natural language interfaces to data,” in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’20, (New York, NY, USA), p. 2629–2636, Association for Computing Machinery, 2020.
- [19] Y. Li and D. Rafiei, “Natural language data management and interfaces: Recent development and open challenges,” in *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD ’17, (New York, NY, USA), p. 1765–1770, Association for Computing Machinery, 2017.
- [20] A.-M. Popescu, O. Etzioni, and H. Kautz, “Towards a theory of natural language interfaces to databases,” in *Proceedings of the 8th International Conference on Intelligent User Interfaces*, IUI ’03, (New York, NY, USA), p. 149–157, Association for Computing Machinery, 2003.
- [21] G. A. Miller, “Wordnet: A lexical database for english,” *Commun. ACM*, vol. 38, p. 39–41, Nov. 1995.
- [22] T. Mikolov, G. Corrado, K. Chen, and J. Dean, “Efficient estimation of word representations in vector space,” in *Proceedings of the International Conference on Learning Representations*, ICLR’13, pp. 1–12, 2013.



- [23] S. Yavuz, I. Gur, Y. Su, and X. Yan, “What it takes to achieve 100% condition accuracy on WikiSQL,” in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, (Brussels, Belgium), pp. 1702–1711, Association for Computational Linguistics, Oct.-Nov. 2018.
- [24] C. Baik, H. V. Jagadish, and Y. Li, “Bridging the semantic gap with sql query logs in natural language interfaces to databases,” in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pp. 374–385, April 2019.
- [25] T. Yu, R. Zhang, K. Yang, M. Yasunaga, D. Wang, Z. Li, J. Ma, I. Li, Q. Yao, S. Roman, Z. Zhang, and D. Radev, “Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task,” (Brussels, Belgium), pp. 3911–3921, Association for Computational Linguistics, Oct.-Nov. 2018.
- [26] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa, “Natural language processing (almost) from scratch,” *Computing Research Repository - CORR*, vol. 12, 03 2011.
- [27] G. E. Hinton, J. L. McClelland, and D. E. Rumelhart, *Distributed Representations*, p. 77–109. Cambridge, MA, USA: MIT Press, 1986.
- [28] C. dos Santos and V. Guimarães, “Boosting named entity recognition with neural character embeddings,” in *Proceedings of the Fifth Named Entity Workshop*, (Beijing, China), pp. 25–33, Association for Computational Linguistics, July 2015.
- [29] Y. Kim, Y. Jernite, D. Sontag, and A. M. Rush, “Character-aware neural language models,” in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI’16, p. 2741–2749, AAAI Press, 2016.
- [30] Q. Le and T. Mikolov, “Distributed representations of sentences and documents,” in *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML’14, p. II–1188–II–1196, JMLR.org, 2014.

- [31] R. Kiros, Y. Zhu, R. Salakhutdinov, R. S. Zemel, A. Torralba, R. Urtasun, and S. Fidler, “Skip-thought vectors,” in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, NIPS’15, (Cambridge, MA, USA), p. 3294–3302, MIT Press, 2015.
- [32] L. Logeswaran and H. Lee, “An efficient framework for learning sentence representations,” 2018.
- [33] W. Wang, M. Zhang, G. Chen, H. V. Jagadish, B. C. Ooi, and K.-L. Tan, “Database meets deep learning: Challenges and opportunities,” *SIGMOD Rec.*, vol. 45, p. 17–22, Sept. 2016.
- [34] T. N. Kipf and M. Welling, “Semi-supervised classification with graph convolutional networks,” in *International Conference on Learning Representations (ICLR)*, 2017.
- [35] R. Cappuzzo, P. Papotti, and S. Thirumuruganathan, “Creating embeddings of heterogeneous relational datasets for data integration tasks,” *SIGMOD ’20*, (New York, NY, USA), p. 1335–1349, Association for Computing Machinery, 2020.
- [36] G. G. Hendrix, E. D. Sacerdoti, D. Sagalowicz, and J. Slocum, “Developing a natural language interface to complex data,” *ACM Trans. Database Syst.*, vol. 3, p. 105–147, June 1978.
- [37] M.-C. de Marneffe, B. MacCartney, and C. D. Manning, “Generating typed dependency parses from phrase structure parses,” in *Proceedings of the Fifth International Conference on Language Resources and Evaluation (LREC’06)*, (Genoa, Italy), European Language Resources Association (ELRA), May 2006.
- [38] F. Basik, B. Hättasch, A. Ilkhechi, A. Usta, S. Ramaswamy, P. Utama, N. Weir, C. Binnig, and U. Cetintemel, “Dbpal: A learned nl-interface for databases,” in *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD ’18, (New York, NY, USA), p. 1765–1768, Association for Computing Machinery, 2018.

- [39] T. Yu, Z. Li, Z. Zhang, R. Zhang, and D. Radev, “TypeSQL: Knowledge-based type-aware neural text-to-SQL generation,” (New Orleans, Louisiana), pp. 588–594, Association for Computational Linguistics, June 2018.
- [40] P.-S. Huang, C. Wang, R. Singh, W.-t. Yih, and X. He, “Natural language to structured query generation via meta-learning,” (New Orleans, Louisiana), pp. 732–738, Association for Computational Linguistics, June 2018.
- [41] J. Guo, Z. Zhan, Y. Gao, Y. Xiao, J.-G. Lou, T. Liu, and D. Zhang, “Towards complex text-to-SQL in cross-domain database with intermediate representation,” (Florence, Italy), pp. 4524–4535, Association for Computational Linguistics, July 2019.
- [42] B. Bogin, J. Berant, and M. Gardner, “Representing schema structure with graph neural networks for text-to-SQL parsing,” (Florence, Italy), pp. 4560–4565, Association for Computational Linguistics, July 2019.
- [43] P. Yin, G. Neubig, W.-t. Yih, and S. Riedel, “TaBERT: Pretraining for joint understanding of textual and tabular data,” in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, (Online), pp. 8413–8426, Association for Computational Linguistics, July 2020.
- [44] K. Affolter, K. Stockinger, and A. Bernstein, “A comparative survey of recent natural language interfaces for databases,” *The VLDB Journal*, vol. 28, no. 5, pp. 793–819, 2019.
- [45] H. Kim, B.-H. So, W.-S. Han, and H. Lee, “Natural language to sql: Where are we today?,” *Proceedings of the VLDB Endowment*, vol. 13, no. 10.
- [46] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa, “Natural language processing (almost) from scratch,” *J. Mach. Learn. Res.*, vol. 12, p. 2493–2537, Nov. 2011.
- [47] K. Yao, B. Peng, Y. Zhang, D. Yu, G. Zweig, and Y. Shi, “Spoken language understanding using long short-term memory neural networks,” in *2014 IEEE Spoken Language Technology Workshop (SLT)*, pp. 189–194, 2014.

- [48] A. Graves, A. Mohamed, and G. Hinton, “Speech recognition with deep recurrent neural networks,” in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 6645–6649, 2013.
- [49] Z. Huang, W. Xu, and K. Yu, “Bidirectional lstm-crf models for sequence tagging,” *ArXiv*, vol. abs/1508.01991, 2015.
- [50] G. Lample, M. Ballesteros, S. Subramanian, K. Kawakami, and C. Dyer, “Neural architectures for named entity recognition,” in *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, (San Diego, California), pp. 260–270, Association for Computational Linguistics, June 2016.
- [51] X. Ma and E. Hovy, “End-to-end sequence labeling via bi-directional LSTM-CNNs-CRF,” in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, (Berlin, Germany), pp. 1064–1074, Association for Computational Linguistics, Aug. 2016.
- [52] T. Young, D. Hazarika, S. Poria, and E. Cambria, “Recent trends in deep learning based natural language processing [review article],” *IEEE Computational Intelligence Magazine*, vol. 13, no. 3, pp. 55–75, 2018.
- [53] M. Ebraheem, S. Thirumuruganathan, S. Joty, M. Ouzzani, and N. Tang, “Distributed representations of tuples for entity resolution,” *Proc. VLDB Endow.*, vol. 11, p. 1454–1467, July 2018.
- [54] S. Mudgal, H. Li, T. Rekatsinas, A. Doan, Y. Park, G. Krishnan, R. Deep, E. Arcaute, and V. Raghavendra, “Deep learning for entity matching: A design space exploration,” *SIGMOD ’18*, (New York, NY, USA), p. 19–34, Association for Computing Machinery, 2018.
- [55] M. Günther, “Freddy: Fast word embeddings in database systems,” in *Proceedings of the 2018 International Conference on Management of Data, SIGMOD ’18*, (New York, NY, USA), p. 1817–1819, Association for Computing Machinery, 2018.

- [56] R. Castro Fernandez, E. Mansour, A. A. Qahtan, A. Elmagarmid, I. Ilyas, S. Madden, M. Ouzzani, M. Stonebraker, and N. Tang, “Sleeping semantics: Linking datasets using word embeddings for data discovery,” in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pp. 989–1000, 2018.
- [57] J. Pennington, R. Socher, and C. Manning, “Glove: Global vectors for word representation,” in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, (Doha, Qatar), pp. 1532–1543, Association for Computational Linguistics, Oct. 2014.
- [58] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, “Enriching word vectors with subword information,” *Transactions of the Association for Computational Linguistics*, vol. 5, pp. 135–146, 2017.
- [59] R. Bordawekar, B. Bandyopadhyay, and O. Shmueli, “Cognitive database: A step towards endowing relational databases with artificial intelligence capabilities,” *CoRR*, vol. abs/1712.07199, 2017.
- [60] D. Bahdanau, K. Cho, and Y. Bengio, “Neural machine translation by jointly learning to align and translate,” 2016.
- [61] B. Perozzi, R. Al-Rfou, and S. Skiena, “Deepwalk: Online learning of social representations,” in *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD ’14, (New York, NY, USA), p. 701–710, Association for Computing Machinery, 2014.
- [62] R. Baeza-Yates, C. Hurtado, and M. Mendoza, “Query recommendation using query logs in search engines,” in *Proceedings of the 2004 International Conference on Current Trends in Database Technology*, EDBT’04, (Berlin, Heidelberg), pp. 588–596, Springer-Verlag, 2004.
- [63] H. Cao, D. Jiang, J. Pei, Q. He, Z. Liao, E. Chen, and H. Li, “Context-aware query suggestion by mining click-through and session data,” in *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD ’08, (New York, NY, USA), pp. 875–883, ACM, 2008.

- [64] X. Yan, J. Guo, and X. Cheng, “Context-aware query recommendation by learning high-order relation in query logs,” in *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM ’11, (New York, NY, USA), pp. 2073–2076, ACM, 2011.
- [65] G. Chatzopoulou, M. Eirinaki, and N. Polyzotis, “Query recommendations for interactive database exploration,” *SSDBM 2009*, (Berlin, Heidelberg), pp. 3–18, Springer-Verlag, 2009.
- [66] N. Khoussainova, Y. Kwon, M. Balazinska, and D. Suciu, “Snipsuggest: Context-aware autocompletion for sql,” *Proc. VLDB Endow.*, vol. 4, pp. 22–33, Oct. 2010.
- [67] J. Akbarnejad, G. Chatzopoulou, M. Eirinaki, S. Koshy, S. Mittal, D. On, N. Polyzotis, and J. S. V. Varman, “Sql querie recommendations,” *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 1597–1600, 2010.
- [68] L. Blunshi, C. Jossen, D. Kossmann, M. Mori, and K. Stockinger, “Soda: Generating sql for business users,” *Proceedings of the VLDB Endowment*, vol. 5, no. 10, pp. 932–943, 2012.
- [69] K. Stefanidis, M. Drosou, and E. Pitoura, ““you may also like” results in relational databases,” 01 2009.
- [70] N. Arzamasova, K. Böhm, B. Goldman, C. Saaler, and M. Schaefer, “On the usefulness of sql-query-similarity measures to find user interests,” *IEEE Transactions on Knowledge and Data Engineering*, pp. 1–1, 2019.
- [71] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, “Distributed representations of words and phrases and their compositionality,” in *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS’13, pp. 3111–3119, 2013.
- [72] M. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, “Deep contextualized word representations,” in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1*

- (*Long Papers*), (New Orleans, Louisiana), pp. 2227–2237, Association for Computational Linguistics, June 2018.
- [73] C. dos Santos and V. Guimarães, “Boosting named entity recognition with neural character embeddings,” in *Proceedings of the Fifth Named Entity Workshop*, (Beijing, China), pp. 25–33, Association for Computational Linguistics, July 2015.
  - [74] C. N. Dos Santos and B. Zadrozny, “Learning character-level representations for part-of-speech tagging,” in *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML’14, p. II–1818–II–1826, JMLR.org, 2014.
  - [75] X. Zhang, J. Zhao, and Y. LeCun, “Character-level convolutional networks for text classification,” in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, NIPS’15, (Cambridge, MA, USA), p. 649–657, MIT Press, 2015.
  - [76] Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin, “A neural probabilistic language model,” *J. Mach. Learn. Res.*, vol. 3, p. 1137–1155, Mar. 2003.
  - [77] R. Socher, A. Perelygin, J. Wu, J. Chuang, C. D. Manning, A. Ng, and C. Potts, “Recursive deep models for semantic compositionality over a sentiment treebank,” in *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, (Seattle, Washington, USA), pp. 1631–1642, Association for Computational Linguistics, 2013.
  - [78] N. Kalchbrenner, E. Grefenstette, and P. Blunsom, “A convolutional neural network for modelling sentences,” in *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, (Baltimore, Maryland), pp. 655–665, Association for Computational Linguistics, June 2014.
  - [79] Y. Kim, “Convolutional neural networks for sentence classification,” in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, (Doha, Qatar), pp. 1746–1751, Association for Computational Linguistics, Oct. 2014.

- [80] H. Zhao, Z. Lu, and P. Poupart, “Self-adaptive hierarchical sentence model,” in *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI’15, p. 4069–4076, AAAI Press, 2015.
- [81] Z. Gan, Y. Pu, R. Henao, C. Li, X. He, and L. Carin, “Learning generic sentence representations using convolutional neural networks,” in *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, (Copenhagen, Denmark), pp. 2390–2400, Association for Computational Linguistics, Sept. 2017.
- [82] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, pp. 1735–80, 12 1997.
- [83] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, “Gated feedback recurrent neural networks,” in *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML’15, p. 2067–2075, JMLR.org, 2015.
- [84] T. Luong, H. Pham, and C. D. Manning, “Effective approaches to attention-based neural machine translation,” in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, (Lisbon, Portugal), pp. 1412–1421, Association for Computational Linguistics, Sept. 2015.
- [85] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, u. Kaiser, and I. Polosukhin, “Attention is all you need,” in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS’17, (Red Hook, NY, USA), p. 6000–6010, Curran Associates Inc., 2017.
- [86] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” 2019.
- [87] A. Radford and K. Narasimhan, “Improving language understanding by generative pre-training,” 2018.
- [88] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, “Language models are unsupervised multitask learners,” 2019.



- [89] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. Salakhutdinov, and Q. V. Le, “Xlnet: Generalized autoregressive pretraining for language understanding,” 2020.
- [90] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, “Albert: A lite bert for self-supervised learning of language representations,” 2020.
- [91] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, “Roberta: A robustly optimized bert pretraining approach,” 2019.
- [92] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, “Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter,” 2020.
- [93] R. Angles and C. Gutierrez, “Survey of graph database models,” *ACM Comput. Surv.*, vol. 40, Feb. 2008.
- [94] P. Hoff, A. Raftery, and M. Handcock, “Latent space approaches to social network analysis,” *Journal of the American Statistical Association*, vol. 97, pp. 1090–1098, 02 2002.
- [95] W. L. Hamilton, R. Ying, and J. Leskovec, “Representation learning on graphs: Methods and applications,” *IEEE Data Eng. Bull.*, vol. 40, no. 3, pp. 52–74, 2017.
- [96] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei, “Line: Large-scale information network embedding,” in *Proceedings of the 24th International Conference on World Wide Web, WWW ’15*, (Republic and Canton of Geneva, CHE), p. 1067–1077, International World Wide Web Conferences Steering Committee, 2015.
- [97] A. Grover and J. Leskovec, “node2vec: Scalable feature learning for networks,” 2016.
- [98] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, “Graph attention networks,” 2018.

- [99] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. van den Berg, I. Titov, and M. Welling, “Modeling relational data with graph convolutional networks,” 2017.
- [100] W. L. Hamilton, R. Ying, and J. Leskovec, “Inductive representation learning on large graphs,” in *Proceedings of the 31st International Conference on Neural Information Processing Systems, NIPS’17*, (Red Hook, NY, USA), p. 1025–1035, Curran Associates Inc., 2017.
- [101] H. Cai, V. W. Zheng, and K. Chang, “A comprehensive survey of graph embedding: Problems, techniques, and applications,” *IEEE Transactions on Knowledge & Data Engineering*, vol. 30, pp. 1616–1637, sep 2018.
- [102] A. Usta, A. Karakayali, and O. Ulusoy, “Dbtagger: Multi-task learning for keyword mapping in nlidbs using bi-directional recurrent neural networks,” *Proc. VLDB Endow.*, vol. 14, p. 813–821, Jan. 2021.
- [103] J. D. Lafferty, A. McCallum, and F. C. N. Pereira, “Conditional random fields: Probabilistic models for segmenting and labeling sequence data,” in *Proceedings of the Eighteenth International Conference on Machine Learning, ICML ’01*, (San Francisco, CA, USA), p. 282–289, Morgan Kaufmann Publishers Inc., 2001.
- [104] R. Caruana, “Multitask learning,” *Machine learning*, vol. 28, no. 1, pp. 41–75, 1997.
- [105] E. Orhan and X. Pitkow, “Skip connections eliminate singularities,” in *International Conference on Learning Representations*, 2018.
- [106] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.
- [107] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, “Densely connected convolutional networks,” in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2261–2269, 2017.

- [108] B. Jou and S.-F. Chang, “Deep cross residual learning for multitask visual recognition,” in *Proceedings of the 24th ACM International Conference on Multimedia*, MM ’16, (New York, NY, USA), p. 998–1007, Association for Computing Machinery, 2016.
- [109] C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky, “The Stanford CoreNLP natural language processing toolkit,” in *Association for Computational Linguistics (ACL) System Demonstrations*, pp. 55–60, 2014.
- [110] E. F. Codd, “A relational model of data for large shared data banks,” *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, 1970.
- [111] D. Cer, Y. Yang, S. yi Kong, N. Hua, N. Limtiaco, R. S. John, N. Constant, M. Guajardo-Cespedes, S. Yuan, C. Tar, Y.-H. Sung, B. Strope, and R. Kurzweil, “Universal sentence encoder,” 2018.
- [112] R. A. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. Reading, MA, USA: Addison-Wesley, 1999.
- [113] M. D. Zeiler, “Adadelata: An adaptive learning rate method,” *ArXiv*, vol. abs/1212.5701, 2012.
- [114] T. Dozat, “Incorporating nesterov momentum into adam,” in *ICLR Workshop*, JMLR.org, 2016.
- [115] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Improving neural networks by preventing co-adaptation of feature detectors,” *ArXiv*, vol. abs/1207.0580, 2012.
- [116] J. M. Zelle and R. J. Mooney, “Learning to parse database queries using inductive logic programming,” in *Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 2*, AAAI’96, p. 1050–1055, AAAI Press, 1996.
- [117] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” 2017.

- [118] N. Reimers and I. Gurevych, “Sentence-bert: Sentence embeddings using siamese bert-networks,” in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*, Association for Computational Linguistics, 11 2019.
- [119] L. Yao, C. Mao, and Y. Luo, “Graph convolutional networks for text classification,” 2018.

