A Comparative Evaluation of Deep Learning based Transformers for Natural Language to SQL query translation

Master Thesis

Author:
Anshu Daur

Examiner and Supervisor:
Prof. Dr. rer. nat. habil. Gunter Saake

Supervisor:
MSc. Gabriel Campero Durand

Magdeburg, 15.03.2021
Anshu Daur
A Comparative Evaluation of Deep Learning based Transformers for Natural Language to SQL query translation
Abstract

NL2SQL, or Natural Language to SQL, is the task of translating questions in natural language to an SQL query which can be executed in a traditional database, such that results are produced and returned to the user. An interface based on NL2SQL could allow users to communicate with database systems as if they were communicating with another human being. In recent years machine learning methods and especially deep learning, have been used to support NL2SQL. A promising group of research work uses word embeddings, a technique that represents textual information as dense numerical vectors. These vectors are formed in such a way that semantically similar words appear closer in a high dimensional space.

In recent years the NLP community has gained ample success from a novel class of models using attention mechanisms to generate contextualized embeddings, based on Transformer architectures. This approach shows benefits for tasks like classification, question-answering, text-summarization and others. Furthermore, a large number of these models are already available as pre-trained models to the public, so as to speed up the research progress.

Motivated by the recent progress in the NLP community we study in this work the role of transformer models to support 2 approaches to NL2SQL, one based on grammar and one based on parsing and multiple classifiers.

Specifically, we replace BERT with GPT2 and RoBERTa transformers on 2 cross-domain datasets (Spider and Sparc) over a grammar based model(IRNet). We also implement the use of GPT2 over a parsing-based model on an existing WikiSQL dataset for comparative evaluation. We report the performance over the 4 versions of modified NL2SQL models implemented as i.e. IRNet+BERT, IRNet+GPT2, IRNet+RoBERTa and SQLova+BERT. We also report the results after hyper-parameter optimization of these models. Hence our core contributions are a better understanding on the role of transformers and parameters on the performance of NL2SQL variant models.

We also perform generalization on a new Sparc dataset over grammar based model(IRNet). As, Sparc is meant for conversational agent but has lot of better revised queries than Spider, so we modify its objectives from multi-turn conversation NL2SQL task to single turn NL2SQL task by adding the turn count as part of schema information and report its performance.

We find that GPT2 is equally competitive with BERT model and better for Sparc when used with large sized contextual vector, large enough to capture the output from the pretrained transformer model. We also find that, using better activation functions like SELU without layer normalization, GELU and Leaky ReLu with layer normalization can lead to equally better and competitive results. In future work we propose that using larger transformers model will give better performance especially when it is tuned properly with the right hyper-parameters.
Acknowledgements

First and foremost, I would like to express my sincere gratitude to my advisor Gabriel Campero Durand, for providing expert guidance, persistent support, encouragement and patience, right from the inception of the thesis topic to the completion. His unwavering enthusiasm for this area of research motivated me immensely throughout the difficult phases of this thesis.

Besides his immense knowledge and work, he proved that I could not have imagined having a better mentor than him. I am thankful to constructive discussions and support during the pandemic time.

I would also like to extend my gratitude to Prof. Dr. rer. nat. habil. Gunter Saake for providing approval to conduct my thesis research under his authority and his valuable understanding throughout the thesis. The excellent research environment and the relevant ongoing research at his Database and Software Engineering Workgroup always urged me to write my thesis under his chair.

I would also like to express my gratitude to the entire NLP community and hugging face\textsuperscript{1} team for their incessant and astounding research and API releases which we could utilize during this thesis.

Finally, and most importantly, I would like to thank my life partner, Prativesh, for constant motivation, emotional support and care. I am incredibly grateful to my family and all my friends for their support that I will never forget.

\textsuperscript{1}https://huggingface.co/
Statement of Authorship

I hereby declare that I am the sole author of this master thesis and that I have not used any sources other than those listed in the bibliography and identified as references. I further declare that I have not submitted this thesis at any other institution in order to obtain a degree.

Signature

Place, Date
# Contents

<table>
<thead>
<tr>
<th>List of Figures</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Tables</td>
<td>6</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>7</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>7</td>
</tr>
<tr>
<td>1.2 Main Contributions</td>
<td>9</td>
</tr>
<tr>
<td>1.3 Thesis Structure</td>
<td>9</td>
</tr>
<tr>
<td><strong>2 Theoretical Background</strong></td>
<td>11</td>
</tr>
<tr>
<td>2.1 SQL</td>
<td>11</td>
</tr>
<tr>
<td>2.1.1 SQL History</td>
<td>11</td>
</tr>
<tr>
<td>2.1.2 SQL standard</td>
<td>12</td>
</tr>
<tr>
<td>2.1.3 Schema relationship with SQL</td>
<td>12</td>
</tr>
<tr>
<td>2.1.4 SQL Syntax</td>
<td>13</td>
</tr>
<tr>
<td>2.2 NL2SQL History and Challenges</td>
<td>15</td>
</tr>
<tr>
<td>2.2.1 Challenges</td>
<td>20</td>
</tr>
<tr>
<td>2.3 An introduction to deep learning concepts used in NL2SQL</td>
<td>21</td>
</tr>
<tr>
<td>2.3.1 Recurrent Neural Networks</td>
<td>21</td>
</tr>
<tr>
<td>2.3.2 Sequence to Sequence model using RNN</td>
<td>25</td>
</tr>
<tr>
<td>2.3.3 Neural Machine Translation as Seq2Seq problem using Encoder-Decoder Architecture</td>
<td>26</td>
</tr>
<tr>
<td>2.3.4 Attention</td>
<td>30</td>
</tr>
<tr>
<td>2.3.5 Pointer Networks</td>
<td>32</td>
</tr>
<tr>
<td>2.3.6 Word Embeddings</td>
<td>33</td>
</tr>
<tr>
<td>2.3.7 Transformers</td>
<td>34</td>
</tr>
<tr>
<td>2.4 NL2SQL approaches</td>
<td>45</td>
</tr>
<tr>
<td>2.4.1 SQLova</td>
<td>46</td>
</tr>
<tr>
<td>2.4.2 IRNet</td>
<td>50</td>
</tr>
<tr>
<td>2.5 Summary</td>
<td>53</td>
</tr>
<tr>
<td><strong>3 Design</strong></td>
<td>54</td>
</tr>
<tr>
<td>3.1 Research Questions</td>
<td>54</td>
</tr>
<tr>
<td>3.2 Pipeline</td>
<td>55</td>
</tr>
<tr>
<td>3.3 NL2SQL Models with Transformers</td>
<td>55</td>
</tr>
<tr>
<td>3.3.1 Grammar-based model with Transformer models</td>
<td>56</td>
</tr>
<tr>
<td>3.3.2 Parsing-based model with Transformers</td>
<td>58</td>
</tr>
<tr>
<td>3.4 Evaluation of NL2SQL models</td>
<td>59</td>
</tr>
<tr>
<td>3.4.1 Exact Match Accuracy</td>
<td>60</td>
</tr>
<tr>
<td>3.4.2 Sketch Accuracy</td>
<td>60</td>
</tr>
<tr>
<td>3.4.3 Execution Accuracy</td>
<td>61</td>
</tr>
</tbody>
</table>
4 Experimental Setup
4.1 Datasets Preprocessing .................................................. 63
4.1.1 Preprocessing for IRNet .............................................. 64
4.1.2 Preprocessing for SQLova .......................................... 65
4.2 Model Hyper-parameters ................................................ 66
4.3 Experimental Environment ............................................. 67
4.4 Summary ........................................................................ 68

5 Evaluation and Results
5.1 Research Questions .......................................................... 69
5.2 RQ1: Hyper-parameter Tuning on Parsing-based model (SQLova) .... 69
5.2.1 Vector Size & Network width ....................................... 70
5.2.2 Activation Functions ................................................... 71
5.2.3 Dropout ...................................................................... 74
5.3 RQ2: Generalization on cross-domain dataset (Sparc) and Hyper-parameter Tuning on Grammar-based Model (IRNet) .... 75
5.3.1 Vector Size & Network width ....................................... 75
5.3.2 Activation Functions ................................................... 77
5.3.3 Learning rate, decay rate and clip gradient ...................... 81
5.3.4 Dropout ...................................................................... 84
5.3.5 Layer Normalization .................................................... 85
5.4 RQ3: Word Contextualization using Transformer Models ............ 87
5.4.1 SQLova with transformer model: SQLova+GPT2 .............. 87
5.4.2 IRNet with transformer model ...................................... 88
5.5 Summary ........................................................................ 89

6 Related Work
6.1 Transformers comparison .................................................. 92
6.2 Related Work : NL2SQL Approaches ................................... 93

7 Conclusion and Future Work
7.1 Conclusion ..................................................................... 95
7.2 Future Work ................................................................... 96

Bibliography ........................................................................ 97

A Appendices
A.1 SQLova+GPT2 : Training and validation graph for sub modules ...... 109
A.2 IRNet with different transformers: Training graph for input vector size and embedding size over Sparc dataset ...................... 112
A.3 IRNet with different transformers: Training graphs for learning rate and decay over Sparc dataset ................................. 112
A.4 IRNet with different transformers: Training graphs for dropout over Sparc dataset .................................................. 112
List of Figures

2.1 Phrase dependency graph in NLQ/A (Phrase dependency graph for input question PDG1: input question "whose is the author of Homo Deus?") and PDG2 : knowledge graph ........................................ 18
2.2 Answer graph generated by NLQ/A (Answer dependency graph for input question PDG1:" whose is the author of Homo Deus?" .................................................. 18
2.3 Match found by Athena ontology annotator for input question 'who is the author of Homo Deus?' .............................................. 19
2.4 Standard RNN architecture : folded model (in left) and unfolded model in time (in right)[Elm90] .......................................................... 22
2.5 Repeating module in an LSTM [co] .................................................... 23
2.6 GRU Cell : Variant of LSTM [co] ....................................................... 25
2.7 Seq2Seq model architecture .................................................................. 26
2.8 Neural Machine Translation model scheme ........................................... 27
2.9 NMT Inference .................................................................................... 29
2.10 Bahdanau Attention in RNN [BCB14, Arb19] ....................................... 31
2.11 Pointer Networks [VFJ17] .................................................................. 33
2.12 Transformer architecture proposed by[VSP+17a] .................................. 35
2.13 Transformer encoder-decoder model[Ala19] ........................................ 36
2.14 BERT-base Encoder blocks (3D representation) [Ala18] ...................... 38
2.15 BERT input representation introduced in [DCLT18] .............................. 39
2.16 BERT language modeling with masked token prediction simplified by [Ala18]. ........................................................... 40
2.17 BERT language modeling with Next Sentence prediction simplified by [Ala18]. ........................................................... 41
2.18 GPT-2 transformer architecture simplified by [Ala19] ............................ 42
2.19 Self-Attention vs. Masked Self-Attention.[Ala19] ................................. 42
2.20 GPT2 Decoder-only transformer blocks : English to German translation [Hug21] .......................................................... 43
2.21 SQLova shallow encoder layer [HYPs19] ........................................... 47
2.22 SQLova Decoder layer [HYPs19] ....................................................... 47
2.23 SQLova NL2SQL layer [HYPs19] ....................................................... 48
2.24 Context free grammar adapted by IRNET : SEMQL [GZG+19] .......... 51
2.25 IRNet Architecture [GZG+19] ........................................................... 51
3.1 NL2SQL pipeline .................................................................................. 55
3.2 Modified IRNET with Transformer Models GPT2 or RoBERTa to encode schema and Natural language question, adapted from IRNET paper [GZG+19] ............................... 56
3.3 SQLova with GPT2 ............................................................................... 58
5.1 300 vector size : Loss on training set .................................................. 71
5.2 300 vector size with GPT2 : Loss on dev set ........................................ 71
5.3 768 vector size : Loss on training set .................................................. 71
5.4 768 vector size : Loss on dev set .......................... 71
5.5 Relu : Loss on training set ........................................ 73
5.6 Relu : Loss on dev set ............................................. 73
5.7 GELU : Loss on training set ........................................ 73
5.8 GELU : Loss on dev set ............................................. 73
5.9 SELU : Loss on training set ......................................... 73
5.10 SELU : Loss on dev set ............................................. 73
5.11 Dropout of 0.5 : Loss on training set for SQlova model over wikisql dataset .... 74
5.12 Dropout of 0.5 : Loss on dev set for SQlova model over wikisql dataset .... 74
5.13 IRNet with different transformers: Training loss on input vector size and hidden layer dimensions ........................................ 76
5.14 IRNet+BERT: Training loss on activation functions over Spider dataset .... 78
5.15 IRNet+GPT2: Training loss on activation functions over Spider dataset .... 79
5.16 IRNet+RoBERTa: Training loss on activation functions over Spider dataset .... 79
5.17 IRNet+GPT2: Training loss graph for learning rate, weight decay and gradient clipping ................................................ 81
5.18 IRNet+BERT: Training loss graph for learning rate, weight decay and gradient clipping ................................................ 82
5.19 IRNet+RoBERTa: Training loss graph for learning rate, weight decay and gradient clipping ................................................ 83
5.20 IRNet with different transformers: Training loss graph for dropouts .... 85
5.21 IRNet with Layer Normalization : Training loss graph ................................. 86
5.22 SQLova+GPT2 : Loss on WikiSQL training set ............................ 87
5.23 SQLova+GPT2 : Loss on dev set ...................................... 87
5.24 SQLova+GPT2 : Execution accuracy on WikiSQL dev set ...................... 88
5.25 SQLova+GPT2 : Logical Form accuracy on WikiSQL dev set .................... 88
5.26 SQL Query Translation : Comparison of IRNet with different transformers on Spider’s dev set(1034 NL questions) ......................... 89
5.27 SQL Query Translation : Comparison of IRNet with different transformers on Sparc’s dev set(1201 NL questions) ................................. 90

A.1 Select column : training set ....................................... 109
A.2 Select column : validation set ..................................... 109
A.3 Aggregation operator : training set .................................. 109
A.4 Aggregation operator: validation set .................................. 109
A.5 Where Column : training set ....................................... 110
A.6 Where column: validation set ..................................... 110
A.7 Number of conditions in where clause : training set ...................... 110
A.8 Number of conditions in where clause: validation set ...................... 110
A.9 Where Operator : training set ..................................... 110
A.10 Where Operator: validation set .................................... 110
A.11 Start Index of where value : training set ............................ 111
A.12 Number of conditions in where clause: validation set ...................... 111
A.13 Start Index of where value : training set ............................ 111
A.14 Number of conditions in where clause: validation set ...................... 111
A.15 IRNet with different transformers: Training graph for input vector size and embedding size over Sparc dataset ........................................ 111
A.16 IRNet with different transformers: Training graphs for learning rate and decay over Sparc dataset ................................................. 112
A.17 IRNet with different transformers: Training graphs for dropout over Sparc dataset ................................................................. 113
List of Tables

5.1 Effect of hidden context vector-size tuning on different modules of SQLova Model. .................................................. 70
5.2 Effect of vector-size tuning on SQLova Model with transformers over WikiSQL dataset based on execution accuracy. .................................................. 72
5.3 Effect of tuning dropout on 6 modules of SQLova model over wikisql dataset .................................................. 75
5.4 Effect of vector-size tuning on IRNet Model with transformers over Spider dataset based on Sketch Accuracy and Exact Match accuracy .................................................. 77
5.5 Generalization over Sparc dataset and effect of vector-size tuning on IRNet Model with different transformers based on Sketch Accuracy and Exact Match accuracy .......................... 77
5.6 Effect of activation function tuning on IRNet Model with transformers over Spider dataset based on Sketch Accuracy and Exact match accuracy .................................................. 80
5.7 Generalization and performance tuning using different activation function on IRNet Model with transformers over Sparc dataset based on Sketch Accuracy and Exact match accuracy .................................................. 80
5.8 Performance tuning using Learning Rate, Decay rate and Clip gradient on IRNet Model with different transformers over Spider’s dev dataset based on Sketch Accuracy and Exact match accuracy .................................................. 83
5.9 Performance tuning using Learning Rate, Decay rate and Clip gradient on IRNet model with different transformers generalized over Sparc’s dev dataset based on Sketch Accuracy and Exact match accuracy .................................................. 84
5.10 Effect of dropout tuning on IRNet with transformers over Spider dataset based on Sketch Accuracy and Exact match accuracy for 100 epochs .................................................. 85
5.11 Effect of adding Layer-Normalization on IRNet model for different transformers over Spider dataset .................................................. 87
5.12 Performance of SQLova+GPT2 transformer over WikiSQL test set for 50 epochs .................................................. 88
5.13 Summary of Improvement in Results for NL2SQL models .................................................. 90
5.14 Summary of generalized Results on Sparc dataset .................................................. 91
6.1 Transformer comparison .................................................. 92
1 Introduction

1.1 Motivation

Natural everyday language provides the most convenient way for regular users to interact with software systems. Even though standard domain-specific query languages, like SQL, are powerful and practical for their applications, they still still require from users certain technical knowledge for employing the language effectively, and hence leveraging software systems. This problem is important as it creates a drawback for more users to employ a given software system, and can slow down decision making. Therefore it is important for the technology community to strive towards building better interaction approaches for users. Within this context, using natural language would enable users to carry out the majority of their operations over text and voice, without specialized query languages or APIs. Smart voice-based interaction will also help users who are blind.

At present, there are already many successful use-cases such as Google voice assistant, Apple Siri, Amazon Alexa, Microsoft cortona and some voice-based interaction systems in cars built by Nuance communication installed in BMW, Audi, etc which allows to adjust settings in car, inform about the traffic, perform gesture recognition to identify where the user is looking and provide information, etc. Such systems handle limited commands but show a great potential in language based interfaces.

Natural Language to SQL (NL2SQL) is the problem to automate the generation of SQL queries for questions asked by users in natural language, serving as a novel application interface to information saved in database system. In order to access the information in relational databases, traditionally users need to have knowledge about database schema, relations, entities along with syntax and semantics of SQL. Since these formal languages like SQL are more machine friendly, it is difficult to learn for most people, hence the need for it’s automation.

The most basic approach adapted initially for such interactions were keyword-based systems to identify keywords enhanced with the fixed dictionary to general SQL systems that were judged as limited[BJK+12, SKI08, ZZM+09, DTB08, BH15]. Keyword based system were then enhanced with pattern-based systems and added more trigger words to identify grammatically enhanced NL keywords and build more complex queries then simple 'Select * from table' statements[DTB08] and could handle where clauses. Some of these systems used user feedback and used domain knowledge to build knowledge graphs[ZCZ+17]. But faced problems of ambiguous relationships in the graph with the increased user-interaction. This was addressed by parsing-based systems which resolved ambiguity problem in dependency graphs.
Semantic parsing maps natural language (NL) to a formal language which can then be mapped to a structured machine executable SQL query for an end system. In parsing, the NL sentence is formatted to identify which of the NL words are important to fill the slots of the algorithm used for generating SQL. Most of the earlier systems used slot-filling mechanisms to parse NL statements and convert them to SQL queries [SFS+16, LYJ07, KBZ06, LJ14]. But such systems faced the ordering problem over different sub-clauses (where clause, orderby, groupby) in queries. This problem was solved by grammar based systems to parse the NL to a format which conforms to rules that keep the syntactical and structural order, hence leading to semantically similar SQL [DDS+16, Fer16].

Some systems also relied on using statistical methods which used training corpus of question-query pairs to provide supervision to the algorithm to optimize the training objective. Most of the statistical systems used supervised machine learning algorithms. Such systems learned lexical mappings given the data by scoring different identified SQL components and then finally built a query with the best identified components [ZC05, PL15, BL15, YYH+17]. But in such systems the dataset was either too simple or too small to extract meaningful features to capture natural language semantics. Thus, predefined features by traditional systems and feature extraction using statistical systems were not able to generalize better on unseen examples. This is a commonly occurring machine translation problem, reappearing in many domains.

Looking beyond NL2SQL, one popular state of the art approach to machine translation is based on deep learning, modelling the task as a sequence to sequence mapping (Seq2Seq), and proposing an encoder-decoder architecture to support it [SVL]. In general, similar models have already been applied to solve semantic parsing [DL16a] with good results.

The progress of language models [DCLT19, RW+19a, LOG+19a] and transfer learning [SET17] in NLP community led to integration of trained language models to extract better contextual word embeddings and fine tuning [Liu19, AHH19] generating better feature vector with seq2seq model. This has lead to state-of-the-art results in NLP community solving problems like summarization, machine translations, hence leading to solving NL2SQL problem with language models using attention mechanism [XLS17, ZXS17, YLZ+18, DL18, GZG+19, HYP19, WHP+18, BGB19].

Technological support for applying Seq2Seq models is currently growing. This support is in the form of libraries, datasets (eg. webpages [ZXS17], books and conversations [YZY+, YZY+19]), new model developments (eg. using attention mechanisms [VSP+17b]), but most importantly in the form of pre-trained models being made publicly available for further research.

Considering the developments for machine translation with Seq2Seq models, we identify that the NL2SQL task is very likely to benefit from them. In fact, there exist already some proposals for using these models for NL2SQL, but these proposals still leave many aspects about the role of the transformer models selected, and the hyper-parameters not studied. Therefore we take as the topic of this work the study on the role of parameters and transformer choices to improve the performance of NL2SQL.

In this thesis, we study modifications to the architecture of grammar based and parsing-based neural models, with the intention of improving the model performance. We study
the impact of different hyper parameter configurations to train and report the outcome of the modified models. And we also contribute by generalizing over grammar-based model for a cross-domain dataset and replicate the hyper-parameter search. Lastly, based on our experiments and evaluations, we propose future works in this area of research.

1.2 Main Contributions

1. We experiment with different configurations of hyper parameters to train parsing based and grammar based NL2SQL models and study their impact on its performance.

2. We replace BERT in grammar-based NL2SQL models with competing transformer models GPT2 and RoBERTa to report the performance with better contextualized embeddings obtained from such language models.

3. We also report performance on parsing-based NL2SQL model with GPT2 transformer

4. We generalize grammar-based model over a novel complex cross-domain dataset. And we also perform hyper-parameter optimization to report the impact in performance.

Overall, we work with 4 versions of modified NL2SQL models i.e. IRNet+BERT, IRNet+GPT2, IRNet+RoBERTa and SQLova+BERT.

1.3 Thesis Structure

The remainder of this thesis is structured as follows:

- In Chapter 2, we explain some fundamental concepts of NL2SQL task and challenges faced by such systems. We review word embedding and deep learning architectures. More importantly, we study transformer approaches to address the bottleneck problem. We review some of the essential works in the field of Natural Language Interfaces.

- Chapter 3 formulates the goals of this thesis. We cover our specific research questions and propose our design, including data pre-processing, encoder-decoder models and evaluation choices.

- In Chapter 4, We review such essential components of the experimental setup like datasets selected, configurations, and programming frameworks.

- Chapter 5 is dedicated to reviewing and discussing evaluation results obtained by the four transformers architectures, and performing a comparison of different classifiers.

- In Chapter 6, we revisit some recent deep learning models implemented for NL2SQL task.
• Finally, we conclude this thesis and outline some promising directions for future research in Chapter 7.
2 Theoretical Background

This chapter focuses on the theoretical background to provide an overview of the essential topics covered in the thesis, to create a solid foundation of understanding of the research area. For convenience, we will be using the abbreviated form "NL2SQL" for Natural Language to SQL translation task. This chapter is structured as follows:

- **Section 2.1**, provides an overview of SQL query language, its history, ISO standards, relationship of SQL with database schema and its syntax followed by some examples.

- **Section 2.2**, provides an overview of Natural language to SQL translation (NL2SQL) as a task and different types of approaches developed through time to address the problem using Natural Language Interfaces (NLI) designed to communicate with the databases. Eventually, we present the major challenges faced by the NLI systems task.

- In **Section 2.3**, we first discuss about some of the concepts of deep learning used to solve NL2SQL problem. It includes recurrent neural networks with lstm and gru cells, followed by state of the art Sequence to Sequence architecture (encoder-decoder), understanding neural machine translation and finally investigating word embeddings, which became the crucial part of applying deep learning methods on natural language to SQL translation. We also explain the concept of attention mechanism which forms the basis of **Subsection 2.3.7**

- In **Subsection 2.3.7**, we discuss transformer models in detail and transfer learning, which inspired this thesis. Also, we provide an in-depth overview of the state-of-the-art transformer architectures to be used in this research for comparative evaluation. We end the section with our reasoning of why we choose these three transformers.

- In **Section 2.4**, we explain the grammar based and parsing based NL2SQL models which we investigate during this thesis in detail.

- In **Section 2.5**, we summarize this chapter.

2.1 SQL

2.1.1 SQL History

SQL was one of the first languages developed for the relational model which is the most widely used database language, supported by all the relational Database Management Systems (DBMS). It originated from relational algebra and tuple calculus and became very popular to get/modify/update/create information in database management systems [Cod70] in 1970. Relational algebra operation used are projection, selection, set...
operations which where the basic operations to be performed with SQL. SQL which is a
descriptive language is powerful because it can manipulate multiple records in multiple ta-
bles with a single SQL query. SQL was called SEQUEL initially. It was renamed and first
standardized in 1986. The most recent standard of SQL is published in 2019. Its syntax
and semantics were kept similar to natural language for developers to easily adapt to the
language[LJ14]. But soon, it’s complexity increased with the complexity of the problem
to be solved and hence the need for NL2SQL systems[LY07, LJ14, FDKZ+18, XLS17]
is required. Recent works about such systems also called as NL2SQL systems. Before
going into detail of NL2SQL systems, let’s first shortly explain SQL standard, syntax and
relationship of SQL query with database schema.

2.1.2 SQL standard

The first initial standard of SQL was released in 1987 called as ISO/SEC 9075 also called
as SQL:86¹ which established standard of SQL in terms of objects, descriptors, scope, ses-
sion, implementation and different data-types to be allowed in the first-most and oldest
version. The later standards like SQL-89 included integrity constraints like UNIQUE,
NOT NULL, DEFAULT and CHECK. The next major version SQL:92 added most im-
portant functionality of providing JOIN syntax and its various types like NATURAL
JOIN, LEFT JOIN, RIGHT JOIN, FULL JOIN, CROSS JOIN. It also provided set
function like union, intersection and difference, except, ALTER, DROP functions for ta-
bles, views. Various data-types were also included like VARCHAR, DATE, TIME, BIT,
TIMESTAMP etc. The size of these standards increased significantly to make sure that
major databases implement it similarly.

use of analytical functions to analyze trends, window functions, set up report and work-
flows with access to different users were provided. They also included JSON file format
for non-relational databases to handle huge amounts of non-relational data especially
useful for distributed environments². The most recent one is SQL:2019³ to provide mul-
tidimensional support for both relational and non-relational databases, intelligent query
processing, memory optimized TempDB metadata, accelerated database recovery, high
availability solutions⁴, database integration, index enhancements especially for columnar-
storage databases and many more.

2.1.3 Schema relationship with SQL

Now for this given relational database D and a natural language question Q_{NL}, nat-
ural language to SQL (NL2SQL) agents will translate/find SQL statement mostly
called as SQL query Q_{SQL} to answer Q_{NL}. This database is a storage engine saving data
and its relationships into tables. The metadata information of how this data is saved in
tables called as schema is also saved in databases.

¹https://www.iso.org/standard/63555.html
²https://learnsql.com/blog/history-of-sql-standards/, last visited: 1 Feb 2021
⁴https://www.sqlshack.com/overview-of-sql-server-2019-general-availability-and-installation/, last vis-
ited : 14 Feb 2021

12
The tables in each column store value which together form a tuple which is the basic units of information saved in a table. The schema of a table is composed of columns, datatype, relationships with other table's columns in the form of primary-key, foreign-key etc. SQL query is processed by SQL query engine of database which compiles and divides into separate clauses. Each clause is processed separately to find which tables to query and what columns to look. More complex queries are formed using join clauses like inner join, outer join, left join etc. as per needed between multiple tables.

In general, each table can consist of 1 or 1 million tuples/records. And while extracting information from these tables in distributed environments where databases themselves are geographically distributed, it becomes easy for NL2SQL systems to process schema information to break down the question to clauses which can be used to finally generate SQL queries instead of looking at all of the data using a smarter agent like SQL query engine.

2.1.4 SQL Syntax

A typical SQL query is also divided into 4 groups according to its functions such as:

- **Data definition language (DDL)**: SQL queries with CREATE/DROP/ALTER commands are for structure organization and management within the databases. With these queries, tables/views/columns can be created, modified or deleted temporarily or permanently form the database.

- **Data Query Language (DQL)**: It is responsible for fetching relevant information with SELECT statements and is described in the following section in more details to cover the relational/algebraic functions that come along with it.

- **Data Manipulation Language (DML)**: These queries are responsible for manipulation of data within the schema using INSERT/UPDATE/DELETE/MERGE/-COMMIT/ROLLBACK/SAVEPOINT.

- **Data Control Language (DCL)**: DCL based SQL queries controls the user’s access rights to schema by using REVOKE and GRANT. This is decided by superuser having access to all admin rights (all rights) of the database.

We will explain DQL statements below, as current NL2SQL systems target on "Select" queries to allow to communicate to databases to get information, because accessing data is a much researched and easier problem to solve first. DQL based SQL statement while fetching information consists of select statement and are written as:

"SELECT COLUMN FROM TABLE TABLE_NAME [WHERE CONDITION]"

From clause in SQL query refers to table/s to be queries and select clause can consist of single or multiple columns. It also uses special sign * to get the all of the information at once. Where clause like other clauses JOIN, ORDER BY, GROUP BY is optional. Addition of these clauses leads to getting more specific information from multiple tables as explained in the following examples.

1. **SELECT**: to get information from column/s of same or different tables
2. **TABLE**: to query information from, also called as schema.

3. **WHERE**: to satisfy condition/s using conditional clauses. like "COLUMN >= or < or == or LIKE VALUE/COLUMN-value". It can handle multiple conditions using conditional operators.

4. **DISTINCT**: to get unique output more formally called as tuples/record

5. **AVG/MAX/MIN/COUNT/SUM**: These are aggregation operations performed on numerical columns.

6. **LIKE** predicate: It performs pattern matching on strings and is contained within the where clauses. it does so by using 2 operators % and _. For string followed by %, it returns zero or more matching characters. And for string which is followed by _ then the query value exactly 1 arbitrary character, if more _ (underscores) are part of the query, then exactly that many arbitrary character are allowed. % and _ forms a kind of regular expression for pattern matching on the strings. For example, "SELECT name from employee where name LIKE 'P%E_H'". Here we want to know name of all employees whose name starts with 'P', followed by 0 or more alphabets and ends with E, exactly one arbitrary character and H which can return a string 'PRATIVESH'.

7. **AND/OR/BETWEEN/NOT/IN** predicates: These are conditional operators used in queries to filter out information based on conditions/clauses mentioned in WHERE clause. For example, "SELECT GAME from TOURNAMENT WHERE TOURNAMENT_YEAR = '2020' AND COUNTRY = 'GERMANY'"

8. **JOIN**: INNER JOIN to get information from multiple table with same column names, OUTER JOIN to get information from both tables on the basis of equality conditions mentioned on specified columns in the SQL query.

9. **Complex query**: query inside query to get more specific information like 'SELECT * FROM PLAYERS WHERE GAME IN (SELECT GAME from TOURNAMENT WHERE TOURNAMENT_YEAR = '2020')'. Here we want to know 'all the players playing games in 2020 tournament'.

There are certain rules associated with usage of these clauses like with COUNT, DISTINCT keyword can be used to count unique values of columns of tables/s. These are most basic of SQL clauses mentioned to fetch information using SELECT clause. In general, SQL also modifies information like it creates and deletes schema/tables using Data Manipulation Language (DML), Data Control Language (DCL) using INSERT, TRIGGERS, functions, stored procedures etc.

The syntax base of SQL stays pretty much the same from different databases owned by different organizations like Oracle\(^5\), IBM DB2\(^6\), MySQL\(^7\), SAP HANA\(^8\), Sybase\(^9\), Teradata\(^10\) etc. SQL is not redundant-free i.e. some queries can be written in many forms. For examples, queries consisting of multiple AND, OR operators with different orders can

\(^5\)https://www.oracle.com/database/, last visited : 15 Dec 2020
\(^6\)https://www.ibm.com/products/db2-database, last visited : 15 Dec 2020
\(^7\)https://www.mysql.com/, last visited : 15 Dec 2020
\(^8\)https://www.sap.com/products/hana.html, last visited : 15 Dec 2020
\(^9\)http://www.sybase.com/, last visited : 15 Dec 2020
\(^10\)https://www.teradata.com/, last visited : 15 Dec 2020
give same results. In a distributed environment where data is fetched from databases that are located at different location, this ordering matters which leads to another problem of optimising the query requests so as to fetch records/data faster. SQL optimization is a separated area of research and is out of scope of this thesis.

2.2 NL2SQL History and Challenges

Even though there have been many developments done over the years to improve the natural language interfaces to improve access to information in the database, NL2SQL is a long standing research problem to be solved and there have been many Natural Language interfaces (NLI) [OWE00, BJK+12, WKW72, SFS+16, LYJ07, BJK+12, BKK05, DTB08, ZZM+09, KBF07, BH15, ZCZ+17, DTB08, KBZ06, LJ14] designed in the past to facilitate querying for users. With the increasing data, machine learning based statistical and neural approaches are becoming mainstream. These NLI systems target only “DQL” statements because select queries as information retrieval is most researched and hence an easier problem to solve. Also, since most of the external users communicating to database in a scaled-up environment via a chatbot will be anonymous and will be accessing in an object-oriented environment using web-services, their access will be handled by proprietor of the information. So, DML statements via chatbot will require custom model’s built on top of NL2SQL models/NLI systems can be implemented at the proprietor’s level. These NLI systems uses rule-based, statistical methods or neural [SMNNA15, DTB08] approaches for SQL query generation and can be classified into 4 categories based on the technical approach [ASB19] as follows:

1. **Keyword-based systems.** Keyword based methods which cover the simplex natural language statement to answer simple queries where the model tries to match the keywords like select, order from with the inverted index of base data and metadata [BJK+12]. Most of the simple question are easily formed with keywords. In order to process the user query and interpret keywords, the question input is first preprocessed using stopword removal, identify synonym and part of speech then performing stemming and lemmatization. Examples of such systems are:

   - **SODA [BJK+12]** is keyword-based NLI which used five steps to translate input question into query. These five steps involve first, the look up in inverted index of the database for each of the keywords identified in the user questions and then a score is assigned for each lookup step and types of matches i.e. fuzzy match or an exact match. Then out of the metadata information available table name and relationships are identified by finding the foreign-keys and then filters conditions are collected to form query. As part of the last the final query is created. Hence, SODA identifies keywords, meta-data information and uses domain knowledge to generate executable query. But it lacks in identifying the comparison operator for filter conditions i.e. ‘to get all books published before 2020’, the query has to be written like this ‘select count (book) from Book where year<2020 group by(year)’. But the question will not identify table ‘Book’ for ‘all books’ keyword in the input and the filter condition has to be explicitly written like ‘year<2020’ to get the relevant output.
NLP-Reduce[KB07] uses natural language processing technologies like stopword removal, punctuation removal and stemming to enhance the user input and then it performs similar steps like SODA to generate SPARQL query. It is better as compared to soda as it can answer input question which do not have the keywords expected by SODA NLI system. but it cannot detect the comparison operators to form where condition for the query.

Precis[SK08] supports multiple operators AND, OR and NOT by transforming the input to Disjunct-normal form(DNF). Since it uses DNF, it can handle brackets with the multiple AND, OR and NOT operators in the where conditions by performing an additional look up for the DNF in the inverted index. Unlike SODA, it does not perform lookup for metadata and hence although it form the query almost perfectly, it cannot identify relevant table names if the name of the tables are not there in the questions explicitly.

QUICK[ZZM+09] is user-interaction based NLI system where the user is guided to improve the question to generate meaningful queries. The system suggests the possible semantic interpretation of the input question from which the user can select one of the suggested SQL query. Although it generated more meaningful queries as compared to SODA and NLP-Reduce, it is limited in generating multiple conjunctions for where conditions.

QUEST[DTB08] and SINA[SMNNA15] uses machine learning techniques combines with semantic techniques to perform database lookup to identify keywords. Both use Hidden Markov Models(HMM) where each state of HMM is the step performed on the input question. For QUEST, keywords are identified as part of the heuristic rules and then with the user feedback all elements like table names, column names for the queries are identified. Then a ranking is done on the most informative join path of the table and tuples with user’s feedback. QUEST is constantly learning from user’s feedback and generating semantically rich queries but it cannot handle table names combined with adjectives. For example if the question contains 'Good Songs' then though it is easy for it to identify just the 'Song' table, it still cannot interpret 'Good Songs' as it does not know what does good refers to, rating of songs, singers of songs etc. For SINA, input question is preprocessed using tokenization, stemming, lemmatization and keywords are extracted. Like NLP-Reduce, it also generates SPARQL queries which are then translated back to SQL queries. It also cannot handle multiple conjunction in the SPARQL queries.

AQQU[BH15] generates SPARQL queries by identifying the keywords and part of speech(PoS) tags like Noun (NN) or Proper nouns(NNP) to identify where values, table names and column names. With the help of verb and adjectives it identifies abstract relationships with the help of machine learning classifier such as binary Random Forest(RF).

Keyword-based NLI systems cannot work on complex user questions as they need good parser and part of speech(PoS) tagger to identify correct “select action”, 'where operator' required by the natural language question. These systems also lack in working on the nested and more complex queries.
2. Pattern-based systems. Pattern-based NLI systems are built on top of keyword-based systems[ASB19] and hence can cover more complex questions including mapping of keywords like most, least to aggregations. It relies on identifying trigger words like “by” to generate “group by” like queries and this poses the problem of identifying every possible synonym for such trigger words. They use knowledge graph to generate queries in a greedy fashion by identifying dependent and independent phrases. Where independent phrases are identified with a dictionary of operators, aggregations, modifiers, variables. And dependent phrases are identified after preprocessing step like stopword removal and using n-gram approach which are extended with synonym dictionary. These are then mapped to a knowledge graph based on edit distance (string similarity) to identify candidate mappings. These mappings are then extended to build dependency graph. But this requires the user to resolve the ambiguity in the dependency graph and hence, is useful for an efficient user interaction process to guide the system to make intelligent edges between the dependency graph that is build to generate a query. We explain two such pattern-based systems below:

- **NLQ/A**[ZCZ+17] uses knowledge graph to generate SQL query to avoid too many interactions with the users. It uses four steps to generate SQL queries where it first preprocesses input, removes prepositions and identifies dependent and independent phrases. It then matches the phrases to an existing dictionary of aggregations, operators, variables, stopwords. After this the input is enhanced with dictionary of synonyms to generate all candidate mapping which helps in enhancing the meaning of input question. Phrase dependency graph(PDG) is created where each of the candidates become the nodes of the graph and the edges become the mapping between phrases identified for each of the candidates as shown in figure. Once the PDG is created, the users feedback is taken to resolve the ambiguities in the multiple candidates identified and then the highest similarity between the phrases to generate the answer graph. The system is simple but it lacks because of multiple user-interaction required to resolve various ambiguous candidate mappings.

- **QUEST**[DTB08] used to query ontologies in three steps key concept identification, context collector and relationship identification. First it identifies all classes, properties, values and then identifies patterns like NLQ/A. After this as one of the most important step, it identifies all semantic relationships between all ontologies in the previous steps and the results are shown after the query is executed. The query generation requires lot of domain knowledge to work with new and already identified ontology.

Although this method is easy and simple to implement but with increasing interaction multiple feedback from user is required to resolve many ambiguous response by the system.

3. Parsing-based systems. Parsing Based methods are better as compared to the previous two methods as the input is parsed to generate structure of the question which is useful for long sentences and hence long range dependencies which cannot be covered by simple questions. It identifies metadata to find relevant table and columns, translation index to identify variations over matching data, identifies numeric and time range expressions like number can be written as string in
**Figure 2.1:** Phrase dependency graph in NLQ/A (Phrase dependency graph for input question PDG1: input question (“whose is the author of Homo Deus?”) and PDG2: knowledge graph)

**Figure 2.2:** Answer graph generated by NLQ/A (Answer dependency graph for input question PDG1: ‘whose is the author of Homo Deus?’)

database or finding and finding dependencies in tokens of input questions. Example of such systems are ATHENA [SFS+16], NaLiX [LYJ07] (natural language to XML) as explained in detail below:

- **ATHENA** [SFS+16] generate intermediate query using ontology driven NLI to handle long question input. It first identifies metadata using inverted index and extent it by finding synonyms from translation index. Input expression, time ranges and token dependencies are extracted from input question. These identified ontology elements are then ranked using an interpretation tree (iTree) and then classified into different types of constraints like weak-connectedness, inheritance. These constraints or interpretations are then ranked and an intermediate query called as Ontology Query language (OQL) is generated which is similar to SQL. OQL consists of identified select clause, from clause, groupby clause, orderby and where clause. Athena queries are better than keyword-based and pattern-based systems as they can handle certain level of nesting in the queries but they cannot handle negation operator in the where conditions.
• Querix[KBZ06] is a dialog based NLI systems where the user is asked to correct the question as per the feedback of ontology driven system. A query skeleton is generated consisting of noun, verb, wh-pronouns(where, who, when) and conjunctions to identify properties, objects and subjects in the question using synonyms dictionary. For this, it first parses the input using Stanford Parser and Wordnet and generated the skeleton of the question. For example,’ Who(Q) is(V) the author(N) of(P) "Homoe Deus"(N) ?’ Such Xqueries are generated using the syntax tree and user feedback is taken to select the best query. The NLI system querix simplicity is advantage and disadvantage as they have to confirm to syntax and can successfully generate simpler queries but cannot work with nested and complex queries.

• NALIR(Natural Language Interface for Relational Databases)[LJ14] is an improved NLI system over NaLIX which uses Stanford parser but it generates SQL queries instead of Xqueries. User feedback is taken initially when the question is posted by user instead of at a later stage to generate a valid and well-adjusted parse tree. This prevents constant user involvement until the final query is generated and results are returned to user but the weakness of depending on an intermediate parse tree still exists.

4. Grammar-based systems. Grammar Based methods identify rules or grammar that define questions and can be better used by systems to generate SQL. They enhance the quality of questions by ensuring grammar generation which does not give error during generation of SQL. Natural language questions which are grammar based can also work recursively and can generate nested queries without error in the systems. The queries generated by such systems never give execution error but the disadvantage is that the grammar is highly domain dependent. We discuss 2 of the latest grammar based NLI systems :

• AskNow[DDS+16] uses Normalized query structure(NQS) which is less prone to the structure conformity of the user input question. The input question goes through POS tagger and Named Entity Recognition(NER) libraries to mark each word in the question to tags and identify named entities. After that a SPARQL query is generated which confirms to NQA templates and confirm to a knowledge base. It also enhances the SPARQL query with synonyms if required by matching it to the keywords of knowledge base. Since it uses intermediate query generation which is then converted to SQL query, it can
also support subqueries but the only disadvantage is identification all correct
POS tags which only allows certain kinds of questions. It is also difficult to
identify all kinds of relationships using NER and POS tagger and limits the
generation of nested queries with joins.

- **SPARKLIS**[Fer16] is semantic based NLI tool to guide users in writing better
SPARKLIS queries which guides users in three stages Faceted Search, Query
builder and NLI. SPARKLIS queries are similar to SPARQL queries justs that
they are written in natural language which is easy for users to understand. It
assists user to form queries with no syntax error by only allowing table names,
column names, operators after taking in the tokens from user’s question. The
final SPARKLIS query generated if error free is sent to the query engine to
eexecute else sent to the user for revision with suggestion to modify the query.
It’s advantage of always generating executable query is also its disadvantage
as it bothers the users until the final perfect SPARKLIS query is generated to
be executed in the system.

### 2.2.1 Challenges

- **Cross domain or Out of domain words or Lexical problem**: All grammar
  based and parsing-based NLI systems are domain dependent as they work with
domain dependent rules to generate final executable query. We need better gener-
alization to generate domain-independent SQL queries.

- **Natural language uncertainty and interpretability**: Most of the NLI systems
discussed above deal with natural language input question by using preprocessing
steps, POS tagging, NER, grammatical conformity and then generating intermediate
or final query with either the user in loop or not. But they face the problem of
dealing with problem of intent where the intent of the question might be different
with the token involved expressing it. It is a common problem which is addressed
by researcher working on Natural Language Processing(NLP domain). Even though
there are some NLI systems [BH15, KBZ06, LYJ07, DDS+16, KBF07, BJK+12]
that have synonym mapping as one of the sub-steps but they do not always cover
conjunction with keywords like none, not or group by as part of the input which
comes from domain knowledge of writing SQL queries. To interpret natural language
questions which can be written in many ways and still mean the same is still an
open problem in NLP and hence in NLI systems.

- **Handle nested queries**: NLI systems [SFS+16] can generate only upto single level
  of nesting in SQL query while other NLI systems do not report generating nested
sub-queries. Nested sub-queries are powerful as they can lead to an optimized
query which could save time to retrieve information or processing time of the query
especially in distributed systems. Generation of nesting within SQL query is another
important challenge to be addressed which needs to be handled by fully working NLI
systems.

- **Number of User Interaction**: NLI systems with user feedback to enhance the
  quality of query generation so that the query is executable often end up giving very
  basic instructions to users[BJK+12]. Another disadvantage is only few questions
can be answered at a time in a dialogue based setting for such rule based NLI systems [DAC10].

- **Unknown words Problem**: Humans tend to use different words from different context/dialects in different manners in their natural language. This interchangeability and adaptability of words across different natural languages poses the problem of having to deal with unknown words for which the older semantic-parsers failed by removing the words entirely. This problem is addressed using modern NL2SQL [GZG+19]

We are focusing on solving Out of Domain words problem and handling nested queries.

### 2.3 An introduction to deep learning concepts used in NL2SQL

Current Research focuses more towards deep learning approaches to implement NLI systems which treats inputs and outputs as sequences and learns over the conditional distribution of tokens in the sequence. Since we are dealing with interpretation and generation of sequences i.e. sequence to sequence problem [SVL14a] where the output is SQL query. Since most of the techniques use neural networks under deep learning domain to solve NL2SQL problem, it comes under Neural Machine Translation (NMT) problem. Recent Deep Learning approaches uses recurrent neural networks like LSTM [GSC99], GRU [DS17] in an encoder-decoder architecture to solve end to end machine translation problems. Encoder-decoder deep learning models that derive motivation from parsing-based NLI systems frames the problem of NL2SQL as a Neural Semantic parsing task [ZXS17, HYPS19, YLZ+18, XLS17] which can be formed as a supervised machine learning problem. Deep Learning based NL2SQL problem takes in NL question as input in encoder and generate SQL queries as output label from decoder in a supervised learning setting. The seq2seq model first learns weights for its neural networks model and takes in the sequence based inputs and generates sequence based output while reducing the loss for every wrong prediction that it makes to generalize over unseen data. More recent methods also use attention [DCLT19, RWC+19a, KR18, SHB16, Kud18] based encoder-decoder models [DL18, GZG+19, XLS17, YLZ+18, HYPS19, DL18, DL16b] to mitigate the problem of vanishing gradients for long term dependencies which is faced by deep learning encoder-decoder models. For simplicity, we will using NL2SQL term throughout this thesis as we focus on using attention based language models in a seq2seq or SEQ2SQL problem domain using recurrent neural networks in an encoder-decoder architecture.

#### 2.3.1 Recurrent Neural Networks

Recurrent Neural Networks (RNN) [Elm90] are like feed forward NN with connection between all neurons of input, hidden and output layers but also had a loop connection between hidden layers of NN to learn over current and also previous inputs seen already by the network. With this feedback connection, RNNs learn over variable length of input sequence (e.g. sequence of words) by modifying the weights of the network for input words of sequence seen in the past to predict the next words. RNN achieves state of
Figure 2.4: Standard RNN architecture: folded model (in left) and unfolded model in time (in right)[Elm90]

the art performance on important tasks that include language modeling[Mik12], speech recognition[GMH13], and machine translation[KB13], sentiment analysis[LFL98], gesture recognition, time series prediction as they are good at capturing temporal dependencies.

RNN architecture can be explained better in the form of equation 2.1 and 2.2 where, weight matrices $W_{xh}$, $W_{hh}$, $W_{hy}$ represent weights matrix between input and hidden layer, hidden-hidden layer and hidden and output layer respectively as shown in figure 1.4. The output at time-step t depends on input provided to RNN at time-step t, on the weight matrices and also on all previous inputs. F and G are the non-linear activation functions like tanh, relu applied over inputs. Weight matrix $W_{hh}$ is called as the memory or state matrix[Elm90] connecting the state from the previous time-step to the state in the current time-step, as its weights are modified to capture the temporal dependencies according to the inputs seen by the network. The current output is a simple linear combination of the current state elements with the corresponding weight matrix.

\[
\begin{align*}
    h_t &= F (W_{xh}x_t + W_{hh}h_{t-1} + b_h) \\
    y_t &= G (W_{hy}h_t + b_y)
\end{align*}
\] (2.1) (2.2)

The weights matrices are learned using back-propagation technique[Guo13] while minimizing the loss by calculating the partial derivative of the loss function with respect to weight matrices and accumulating all contributions to the change in the error from each state. In terms of text based inputs, we feed the one-hot vectors to the RNNs, and the network will output value close to 1 when the text is detected else will output value close to zero using a softmax function in the output layer. Since the size of weight matrices and networks remains unchanged, over time accumulating gradients the gradients become too small which is called as 'Vanishing Gradients' problem where the gradients stemming back from many time-steps become negligible and so the partial derivative of the error also become negligible, and the network stops learning. Therefore temporal dependencies that after multiple time-steps are effectively discarded by the network. RNNs also suffers from exploding gradient problem, in which the value of the gradient grows uncontrollably
and solution for the exploding gradient problem is Gradient Clipping. Long Short-Term Memory (LSTM) [GSC99] and Gated Recurrent Units (GRU) [DS17] RNNs were designed to specifically Vanishing Gradient problem.

The gates in GRU and LSTMs have fuzzy analogy to each other in the sense that there are no memory cells in GRU and hence they are not exactly performing the same functions. Bigger difference is GRUs need less memory and hence are computationally more efficient as compared to LSTMs and make it easier to retain information for the long term. There have been many variants of GRUs and LSTMs proposed over the year but there is not fixed evidence where LSTMs or GRUs performs consistently better than the other.

**Long Short-Term Memory (LSTM)**

LSTM cells for each neuron [GSC99] are good at capturing long-term dependencies by allowing certain inputs to be latched or stored for longer period of time without forgetting them. In RNN, the next state was calculated through a simple activation function (tanh or relu) over linear combination of inputs and corresponding weight matrices. The output was also calculated as linear combination. LSTMs have more complicated computations in a single neuron and the connection between input, outputs and hidden neurons are same as seen in previous RNN. It uses multiple gates to separate information to forget about the past inputs and learn new information. LSTMs are useful at learning long-term dependencies described as follows:

![Figure 2.5: Repeating module in an LSTM](image)

Single LSTM cell as shown in Figure 2.5 consists of 4 operations using sigmoid function, hyperbolic tangent (Tanh) function, addition and multiplication where each of the functions are fully differentiable. The three sigmoid functions acts as a gates which decides what goes into the cell, what retains into the cell and what passes to the output by outputting numbers between zero and one (1: allow information to pass through, 0: not allow).

- **Forget Gate**: uses sigmoid function to forget the information summary carried over by the previous state’s weight matrices. The activation output from these multiplied with the previous state and current input both and added together to decide what
to forget and what to keep. The values near to 1 are retained and values near to zeros are forget as shown in the equations:

\[ f_t = \sigma(W_f \cdot h_{t-1} + W_f \cdot x_t + b_f), \] (2.3)

- **Input or Learn Gate**: uses both sigmoid and tanh activation function, where sigmoid’s role is similar as discussed. It modulates the amount of the current input that is going to be stored in the memory over two computation paths: similarly to the forget gate, the input gate takes the current input \( x_t \) and the output of the previous step \( h_{t-1} \) and uses a sigmoid layer to generate an activation output zero and one. Tanh layer updated state \( \tilde{c}_t \) between \([-1, 1]\) as shown in the following equations:

\[ i_t = \sigma(W_i \cdot h_{t-1} + W_i \cdot x_t + b_i), \]
\[ \tilde{c}_t = \text{Tanh}(W_c \cdot h_{t-1} + W_c \cdot x_t + b_c). \] (2.4)

- **Remember Gate**: \( c_t \) and the previous state \( c_{t-1} \) fully determine the state at time \( t \) with forget gate \( f_t \) and input gate \( i_t \). In short it takes long-term memory coming out of the forget gate and short-term memory coming out of the learn gate and simply combines them together.

\[ c_t = f_t \cdot c_{t-1} + i_t \cdot \tilde{c}_t \] (2.5)

- **Output or Use Gate**: output of LSTM at time \( t \) passes through tanh non-linearity to generate output and sigmoid layer determines the next state of output gate. It takes useful long-term information from forget gate and useful short-term information from learn gate and multiplies them together to generate output.

\[ o_t = \sigma(W_o \cdot h_{t-1} + W_o \cdot x_t + b_o), \]
\[ h_t = o_t \cdot \text{tanh}(c_t). \] (2.6)

**Gated Recurrent Unit (GRU)**

GRUs[14] are variation of LSTM RNN which uses less gates to forget past information as shown in the figure and hence need less memory with less computations performed each iteration. It has merge the two gates in learn or input gates of LSTM and combined them by performing addition over the weight matrices coming from current input, long term information from forget gate and short term information from previous state and passed it through sigmoid gate to learn the new information.

\[ z_t = \sigma(W_z \cdot h_{t-1} + W_z \cdot x_t), \]
\[ r_t = \sigma(W_r \cdot h_{t-1} + W_r \cdot x_t), \]
\[ \tilde{h}_t = \text{tanh}(W_o \cdot r_t + h_{t-1} + W_o \cdot x_t), \]
\[ h_t = (1 - z_t) \cdot h_{t-1} + (z_t) \cdot \tilde{h}_t. \] (2.7)
z is the update gate which balances what parts of hidden states \( h \) are updated and preserved from the previous state \( h_{t-1} \) and current input at time \( t \) respectively. \( r \) is the reset gate which controls which parts of previous hidden state \( h_{t-1} \) is used to compute next output. \( h_t \) selects useful vector information from reset gate by multiplying with previous hidden state and using current input to create updated hidden state corresponding to the new output.

2.3.2 Sequence to Sequence model using RNN

For Sequence learning, RNN as discussed in Subsection 2.3.1 are used in Sequence to sequence architecture(Seq2Seq) was proposed for machine translation task in which a sentence in source language(source sequence) is translated to target language(target sequence) [BCB16, SVL14b, CvMG+14]. RNNs deals with fixed dimensionality of input data so as to generate output but does not perform well for output with variable sizes. To mitigate the problem of predefined fixed dimensionality, Google published a sequence to sequence architecture[SVL14b] in 2014. Core Idea of Seq2Seq model is to learn a probabilistic model from fixed length input in the form of text treated as sequence of characters or words and generate variable length output which is also a sequence of characters or words. Sequence to Sequence architecture are type of models that use 2 RNNs for encoding the input sequence and then learning over their conditional probabilities to output vector equivalent for output sequence. These 2 RNN models are called encoder and decoder. The encoder consists of LSTM or GRU units, initialized with random weights which takes as input a sequence of text and forms a fixed-size vector representation of it. The decoder consists of another set of LSTM or GRU units, initialized with the last hidden state of encoder. It uses the abstract representation produced by the encoder which is mapped to fixed size representation to generate output sentence. The fixed-size

![GRU Cell](image)

**Figure 2.6:** GRU Cell : Variant of LSTM [co]
vector representation captures relevant information from the input sequence and maps it to an arbitrary dimensional space where similar words or tokens are closer to each other. Figure 2.7 shows the seq2seq model architecture.

The paper proposes two RNNs stacked one after the other, where the first RNN is encoder which takes input size of variable length and map it to fixed size vector and second RNN is decoder which will decodes or generates variable length output by taking in the fixed dimensional word vectors and previous output predicted by the decoder. Authors used LSTM cells in both encoder and decoder so as to handle long range dependencies.

2.3.3 Neural Machine Translation as Seq2Seq problem using Encoder-Decoder Architecture

Seq2Seq is an example of conditional language model and was trained for Machine Translation(MT) task initially to translate English sentences to french sentences is a versatile model. It is termed similar to language model as decoder is predicting next words of the target sequence conditioned on source sequence. It involves 2 RNNs which acts as Encoder and decoder to which English and French pair of sentences as input are passed in parallel to estimate the conditional probability of french words given the english words. Encoder encodes the source sequence into hidden state and decoder learns a conditional view of translation by modeling the probability of a target sentence $w_{1:T}$ given a source sentence $x_{1:S}$ as

$$P(w_{1:T}|x) = \prod_{t=1}^{T} P(w_t|w_{1:t-1}, x; \theta) \quad (2.8)$$

Figure 2.8 presents the workflow of NMT model. The example shown is a translation(English to French) problem [LBZ17]. Both the encoder and the decoder are RNNs: LSTMs [SVL14c, LPM15a] or GRUs [LYD+17, SLL15]. To allow variable length sequences, the sentences are padded with 0 to reach the size of the longest sentence of the training set.

A source encoder recurrent neural network(RNN) maps each token or word or character from source encoder to respective output tokens and processes these to a sequence of hidden vectors $h_1, \ldots, h_S$. The target decoder combines encoder’s RNN hidden representation of previously generated tokens $(w_1, \ldots, w_{t-1})$ with hidden vectors to predict conditional score for each possible next token of the sequence. Decoder is also called as
Figure 2.8: Neural Machine Translation model scheme. (a) Source input refers to the input sequence tokenized to let the encoder read it token by token. (b) Target input refers to the decoder input that lets the decoder have the correct $t-1$ token. (c) Embedding layer refers to the feed-forward neural network language model that creates hidden vector representation. (d) Encoder refers to stacked LSTMs used to construct hidden vector i.e. an abstract representation of the input sequence. (e) Decoder refers to stacked LSTMs that produce the output sequence based on the abstract representation constructed by the encoder and on the previous target word. (f) Hidden layers refer to the number of stacked RNN layers. (g) Projection layer generated the probability distribution over the next output token or word which is used to calculate the loss. (h) Decoder output refers to the output sequence generated.[LBZ17]
language model which generates target sequence that is conditioned on encoding of source tokens.

The model is trained and output of decoder layer goes through softmax function to produce the output distribution \( P(w_t|w_{1:t-1}, x; \theta) \) over target dictionary of token or words and take argmax over it to get the topmost predicted token. Categorical cross-entropy loss function is calculated between the target and decoded distribution and the loss is back propagated through the network to minimize the error over predictions. Thus, the complete model is trained end-to-end to minimize the negative log-likelihood over the next word using back propagation approach.

**Inference in NMT model**

Once the model is trained, inference on NMT model is obtained by freezing the encoder weights and use just the decoder to generate new output tokens at each step giving a distribution over possible next tokens conditioned over all tokens generated till now[Neu17]. But, since the space of possible output tokens is exponentially large, heuristic search methods such as greedy decoding or beam search[FAO17, Gra12] must be used to find high-probability sequences.

**Greedy decoding** takes the most likely token at each \( t \) time-step with the highest probability using argmax function hence termed as greedy as it only looks at the best solution at that very moment. It discards tokens that lead to too low probability and takes only most probable word. This approach is computationally efficient but does not have great quality[Gra12, CCBL18] as it does not always produce optimal solution. Figure 2.9 shows the inference process using a greedy search. Greedy Search cannot backtrack to its previous decisions i.e. tokens selected which might lead to an overall low probability of output sequence.

Another way is to perform **Exhaustive search** by computing all possible sequences but then it means keeping track of all possible partial translations. It find the most optimal solution but the search space for decoder becomes very huge. So, if output vocabulary is \( V \) then the complexity to choose relevant tokens will be \( O(V^t) \) where \( t \) is the length of target sequence which is very expensive w.r.t memory and time. We need algorithm that approximates this bigger search space to decide among most relevant tokens. Beam search decoding is one such method to obtain a more effective solution for building predictions.

**Beam Search Decoding**

**Beam search decoding** is important algorithm used by the decoder while it has to predict the most probable token out of the tokens with higher conditional probabilities given the encoder context vector and the tokens already predicted by the decoder. This algorithm perform approximate search and it stops searching as soon as the exhaustive search goes beyond a certain predefined threshold value [FAO17]. This threshold value can be manually set or can be defined heuristically. Beam Search searches by keeping track of the \( K \) most probable partial translations which are also called as hypotheses until the number of possibilities exceeds a certain threshold. \( K \) is the integer and called as
**Figure 2.9:** Neural Machine Translation inference (a) *Source input* refers to the input sequence tokenized to let the encoder read it token by token. (b) *Embedding layer* refers to the feed-forward neural network language model that creates words’ vector representation. (c) *Encoder* refers to stacked LSTMs used to construct an abstract representation of the input sequence. (d) *Decoder* refers to stacked LSTMs that produce the output sequence based on the abstract representation constructed by the encoder and on the \( t - 1 \) output word. (e) *Hidden layers* refer to the number of stacked RNN layers. (f) The *Greedy search* takes the most probable output word at each time-step (g) *Decoder output* refers to the output sequence tokenized[LBZ17].

beam size which is equivalent to looking at the search space K times. Instead of selecting most likely token at each step, we require the model to store \( b \) the most likely predictions’ prefixes \( y_{b1}, \ldots, y_{bt} \). Using this approach, we do not choose the locally optimal token, but in general the optimal hypotheses considering other k tokens, which increases the chance
of building a correct prediction. It is important to notice, beam search still does not guarantee, that final translation is an optimal in $D^L$ space, but it broadens search space comparing to greedy-search.

Let us define $k$ hypotheses seen so far which is just the conditional log probabilities of new token given k hypotheses seen so far:

$$y_{<k>}^i := (y_1^i, \ldots, y_{k-1}^i), \text{ where } i - \text{ beam search branch index}$$  \quad (2.9)

Suggest we have already finished $k$ iterations of beam search. At that moment we find beam most likely $k$-long prediction hypotheses:

$$y_{<k+1}^1, \ldots, y_{<k+1}^{beam}$$  \quad (2.10)

Beam Search algorithms for $k + 1$ step is as follows:

1. For all $i \in [1, beam]$ beam search takes cumulative probabilities

   $$P_i = \{ p(y_{k+1}|y_{<k+1}, x, \theta)p(y_{<k+1}^i|\theta) | y_{<k+1}^i \in \mathcal{D} \}$$  and corresponding $y_{<k+1}$ tokens.

2. Let $P = \bigcup_{i=1}^{beam} P_i$

3. Beam search chooses beam with most highest probabilities in $P$ and its corresponding hypotheses or tokens $y_{<k+1}$

4. So we have $y_{<k+2}^1, \ldots, y_{<k+2}^{beam}$ $k + 1$ hypotheses or tokens on $k + 1$ step.

Finally after completing L iteration or after seeing the <END> or </s> token, we get $y_{<L+1}^1, \ldots, y_{<L+1}^{beam}$ hypotheses. Final translation is most likely $L + 1$ hypotheses among them. Different hypotheses may produce <END> tokens at different timestamps, hence selection of beam width has to be done carefully.

In summary, this method stores beam of the "most likely" hypothesis also called as token output by decoder at each iteration, in contrast to the naive approach, which greedily selects one locally optimal hypothesis at each iteration. The beam search is computationally more expensive than the greedy search and less expensive than exhaustive search, hence is the tool of choice in decoders. The trade-off beam search as the model learns a distribution that tends to carry more weights to singular tokens or shorter hypotheses. So, performance of beam search decreases for longer hypotheses as the scores keeps getting lower because of multiplications of conditional probability scores of tokens in sequences. One of the fix for this problem is to normalize the scores and select the token with highest score.

2.3.4 Attention

Neural Machine Translation (NMT) can achieve state-of-the-art performance in large-scale translation tasks such as from English to French but it is still challenging to handle long sequences because of fixed context vector size containing summarized information from all of the encoder’s hidden states.\cite{co, BCB16}. It leads to loss of information as all of the information about the sequences is forced into only one single context vector. NMT systems are also less interpretable and hence harder to debug.
In an encoder-decoder model, the importance of focusing on relationship between words is prioritized by weights mainly to help the decoder improve its prediction. Many research work has been published with better results using variety of attention mechanism [VSP+17b, LPM15b, SKFH18, KDHR17, LFdS+17] as solution to fixed context vector problem in NLP.

Attention [BCB14] is a recent technique proposed as solution to bottleneck problem of seq2seq based machine translation systems. The core idea of the attention mechanism is based on letting the model focus on the input sequence’s relevant parts as needed. It does so by using all of the encoder hidden states weighted by decoder’s state at current time step passed through softmax to get attention distribution over the source which focuses only on particular part of the source sentence or sequence. Attention is a technique to compute weighted sum over input values dependent on query [LPM15a, BCB16]. The query determines how to pay attention to the values which in essence are encoder hidden states.

As shown in Figure 2.11, all encoder hidden states $h_t$ is passed through attention layer which is a single feed-forwarded neural network and we get the attention scores $a_t$ which is passed to decoder hidden states $h_s$ at time $t$ as described in Eq. 2.11. These scores are also called as alignment scores as they have same dimension as encoder hidden states and they make decoder aware about at what parts of the input the decoder should focus at time $t$. Next, we take softmax over these attention scores to get attention distribution and normalize it so that the final scores sum up to 1.

$$a_t(s) = \text{attention}(h_t, \tilde{h}_s) \tag{2.11a}$$

$$= \frac{\exp(\text{score}(h_t, \tilde{h}_s))}{\sum_{s'} \exp(\text{score}(h_t, \tilde{h}_{s'}))} \tag{2.11b}$$

The attention score function score($h_t, \tilde{h}_s$) can be calculated in different ways in [LPM15a],

Figure 2.10: Bahdanau Attention in RNN [BCB14, Arb19]
as described in Equation 2.12.

\[
\text{score}(h_t, \bar{h}_s) = h_t^T \bar{h}_s \quad \text{dot – product} \quad (2.12a)
\]

\[
= h_t^T W_a \bar{h}_s \quad \text{multiplicative} \quad (2.12b)
\]

\[
= v_a^T \tanh(W_a[h_t^T; \bar{h}_s]) \quad \text{concat} \quad (2.12c)
\]

\[
= v_a^T \tanh(W_a h_t^T + W_b \bar{h}_s) \quad \text{additive} \quad (2.12d)
\]

In Equation 2.12b, \( W_a \) is a learnable parameter to get the best weight matrix to get the attention scores, this type of attention mechanism is called as multiplicative attention. Concatenation operation or concat attention mechanism applies simple linear transformation over concatenated encoder and decoder hidden states using a single feed-forwarded neural network to generate final attention scores which is nothing but a learnable weight matrix \( W_a \). Another type is additive attention, as mentioned in Eq. 2.12d where a linear transformation is applied to both encoder hidden states and decoder hidden states which are then added and passed through another non-linear transformation to be multiplied with another \( v \) weight vector to get a single score. So, we learn two weight matrices \( W_a \) and \( W_b \) to get additive attention score. Additive attention mechanism has one more hyper-parameter Lastly, we take attention output \( a_t \) concatenated with the decoder hidden states \( h_s \) and pass it through more fully connected layers to get decoded output sequence.

Since we are passing all the encoder hidden states to the decoder and not just the summarized information in one single context vector, also solves the bottleneck problem because of these attention scores which acts as skip connections provided to the decoder’s states. Attention mechanisms are thus great at mitigating the problem of vanishing gradient in seq2seq models for case of long sequences. It also significantly improves the performance as decoder can focus on particular word at time \( t \).

### 2.3.5 Pointer Networks

Pointer network are a variation of the attention based sequence-to-sequence model to generate ordered and variable length output sequence by decoder[VFJ17, GAN +16]. They can handle variable sized vector as inputs by generating a softmax attention score which is a pointer to select an input token from input vocabulary rather than generating a weighted fixed size attention vector over encoder hidden states.

\[
u^j_i = v_a^T \tanh(W_a h_t^T + W_b \bar{h}_s) \quad (2.13a)
\]

\[
p(C_i|C_1, ...C_{i-1}, P) = \text{softmax}(u^j) \quad (2.14a)
\]

As described in Equation 2.13a, linear transformation is applied to both encoder hidden states and decoder hidden states which are then added and passed through another non-linear transformation to be multiplied with another \( v \) weight vector to get a single score, where \( v, W_a \) and \( W_b \) are learnable parameters of the output model i.e. the decoder. This is also called as the additive attention. It outputs a conditional probability of next
integer input index to point to given sequence of previous integer indices c and soft-maxed additive attention output over encoder and decoder hidden states. Thus, it responds to only the position of the input sequences.

Pointer networks are useful when the output is a sequence of known vocabulary and the decoder output corresponds to the position in the input. Hence, they are also used for question answering tasks, where the answer is generated on the basis of question and given text. For NL2SQL task, many architectures solve the unknown word problem [LJ14] using pointer networks, by using argmax function over softmax to get index of the most probable word in the vocabulary.

2.3.6 Word Embeddings

Natural language is a complex system used to express meaning. In this system, words are the basic unit of meaning. As the name implies, word vectors are vectors used to represent words, and can also be considered as feature vectors or representations of words. The technique of mapping words to real number domain vectors is called as word embedding. In recent years, word embedding has gradually become the basic knowledge of natural language processing.

Embedding mathematically represents a mapping function, where the function such as f: X -> Y is injective (as each Y has only a unique X corresponding, and vice versa) and structure-preserving (Structural preservation, for example, X1 < X2 in the space where X belongs, then after mapping, the same is true in the space where Y belongs to Y1
Word embedding maps word vector to another space, where this mapping has the characteristics of injective and structure-preserving. Popular translation can be thought of as word embedding, that is, mapping words in the space of X to a multi-dimensional vector in the Y space, then the multi-dimensional vector is equivalent to embedding in the space of Y. Word embedding is to find a mapping or function to generate an expression in a new space. This expression is the new word representation. Word embeddings were mainly described as frequency-based and prediction-based. For example, frequency based embedding is a mapping method based on word frequency statistics. Count-Vector is one such algorithm, which is the simplest and most basic word frequency counting algorithm: for example, we have N documents, we count the number of different words in all the texts, and the result is a matrix. Then each column is a vector, representing the number of times the word appears in different documents. But there is a disadvantage of this method that it does not leverage co-occurrence statistic between words and is very high dimensional. This also lead to the problem of sparse vectors.

Another algorithm is Global vectors for word representation also called in short as GloVe, which was published by the Stanford NLP team in 2014. GloVe uses word co-occurrence information. It defines a co-occurring word matrix, where the element is the number of times the word appears in the context of another word i. It learns the log probability of co-occurrence of words which is equivalent to the vector difference in feature space of words. Owing to the fact that the logarithm of a ratio equals the difference of logarithms, this objective associates (the logarithm of) ratios of co-occurrence probabilities with vector differences in the word vector space. Because these ratios can encode some form of meaning, this information gets encoded as vector differences which is then consumed by machine learning models. Glove gives very informative word embeddings by learning a non-decreasing function, so as to ensure that the greater the number of co-occurrences of the vocabulary, the weight of the less frequent words do not decrease. And for words that are too frequent, the learned complex mapping should be able to give a relatively small value so that the frequently occurring words will not be over-weighted.

2.3.7 Transformers

The transformer is a deep learning architecture first introduced by [VSP+17a], which relies solely on attention’s power. While using LSTM and gated neural networks became common in sequence modeling problems such as machine translation, incorporating attention mechanism along with a neural network, gained much success in tasks like text summarization and machine translation. The authors of [VSP+17a] introduce a new architecture called Transformer, which completely abandons RNNs and is completely based on attention to derive global dependencies between input and output. Due to its architecture, transformers do not require that the sequential data be processed in order, leading to higher parallelization power and translation quality. As shown in Figure 2.12, the transformer architecture maintains the classical encoder/decoder model, but it succeeded the RNN by a new design called Multi-head attention. The original encoder and decoder architecture is composed of a stack of 6 identical layers. Encoder layers broke into two sub-layers: a multi-head self-attention mechanism and a position-wise fully connected feed-forward network. The authors of the transformer use residual connections.
around each sub-layers, which improves training performance. In addition to the sub-layers in each encoder layer, the decoder includes a third sub-layer, which performs multi-head attention over the encoder’s output. Multi-head attention allows the model to pay attentions to the information w.r.t different representation sub-spaces and at different positions of the words in the sequences. The decoder’s self-attention mechanism only has access to the words created by the decoder, and words that need to be processed are listed as masked when masking words with high negative values will be discarded from the learning algorithm.

In order to deal with order of the sequence, The authors use a method called positional encoding for each input word which can inject some information about the relative or absolute position of tokens in the sequence. Positional encoding can obtain by combination of the functions sine and cosine with different frequencies. Final experiments conclude that the transformer model can be trained faster than architectures based on recurrent,
convolutional or fully connected layers. Also, transformers outperformed the Google’s Neural Machine translation model in many tasks which inspired a new wave of transformer architectures which have been trained with huge general language datasets, and can be fine-tuned to specific language tasks.

The encoder encodes an input sequence into some hidden representations \( h_1, \ldots, h_p \). These hidden representations from the last encoder layer is fed to the decoder as hidden states before building the output prediction as shown in Figure 2.13. The decoder output is a vector set \( \{(p_{i1}, \ldots, p_{id})\}_{i=1}^{m} \), where \( d \) - target language dictionary power, \( m \) - maximum possible prediction sequence length, \( p_{ij} \) - the model’s conditional probability that the \( i \)th token of the predicted sentence is the \( j \)th token from the target language dictionary.

As a result, the translation model builds a 'probability distribution' in the space of the cartesian product of the target dictionary.

![Figure 2.13: Transformer encoder-decoder model](Ala19)

**Bidirectional Encoder Representations from Transformer : BERT**

The popularity of transformer architectures sparked a lot of new papers based on this idea. BERT, first introduced in [DCLT18], is one of the most popular transformers that outperformed traditional deep learning models for many NLP problems. In this section, we will study BERT as one of the four approaches we use in our proposed design.

BERT is a universal language model, pre-trained on large amounts of text data to fine-tune it on downstream tasks in a supervised manner with relatively little data. It emphasize the importance of bidirectional learning during pre-training using stack of encoders having 12 layers for base bert models with 12 heads, 24 for large BERT model with 16 heads . Semi-supervised sequence learning [DL15] introduced two approaches that use unlabelled data to improve sequence learning with recurrent networks. BERT takes its inspiration from semi-supervised, generative pretraining, ELMO[PNI+18]
and ULMfit[HR18] to build a deeply bidirectional model to learn contextualized embeddings using bidirectional self-attention (left-to-right and right-to-left) as depicted in Figure 2.14.

BERT is jointly conditioning on both the left and right context by training on two unsupervised tasks simultaneously, Masked Language Model (MLM)[Tay53b] and Next Sentence Prediction (NSP) tasks. It is built on only encoder layers of transformer language model [VSP+17a]. In MLM task, it randomly masks 15% of the input tokens and conditions each word bidirectionally to predict the masked word/token as the original vocabulary id of the word. Along with MLM task, bert also trains simultaneously on NSP task in which text-pair representations are jointly pre-trained like a binary classification task which outputs if the second sequences follows the first sequence or not. This helps BERT to understand context within its own sequence and across different sequences. BERT is available by Google in two model sizes as follows11:

- BERT\textsubscript{BASE}. 12 transformer blocks, 768 hidden units, and 12 attention heads.
- BERT\textsubscript{LARGE}. 24 transformer blocks, 1024 hidden units, and 16 attention heads.

BERT\textsubscript{BASE} is comparable in size to the GPT[Ala19] transformer in order to compare performance. BERT\textsubscript{LARGE} was introduced to prove that larger models lead to accuracy improvement. More details of about BERT related to attention mechanism, input representation and pre-training tasks as mentioned as followed:

1. Multi-headed self-attention in BERT: Multi-head attention improves the ability to focus on different words of the input sentences and give the attention layer multiple representation subspaces [Ala18]. As shown in Figure 2.14, self-attention gets query, keys and values vectors learned after passing the input embedding vector through linear fully connected layers of neural network.

   a) Query vector (Q): The query vector of the current word is used to multiply the key vector of other words to get the attention score of other words relative to the current word. We only care about the query vector of the word currently being processed.

   b) Key vector (K): The key vector is like the label of each word in the sequence. It enables us to match the object when searching for related words.

   c) Value vector (V): The value vector is the true representation of the word. When we calculate the attention score, we use the value vector to perform a weighted summation to obtain a vector that can represent the context of the current location.

Keys, queries and values derive analogy from the databases where for each query asked by the user to get the value, a key is generated for the query. This key is the location of the value in the block of memory. But in transformer architecture, same input vector is fed as the query, key and value vector. The key and query vectors are multiplied (using dot product), scaled (dividing by dimensionality of the word embedding vector) to keep the flow gradients stable across all the layers. These values are passed through softmax function and normalized to generate scores. A dot product of the softmaxed scores and value vector is taken to determine the

11https://github.com/google-research/bert
Figure 2.14: BERT-base Encoder blocks (3D representation) [Ala18]

context/importance of the word w.r.t to input sentence. And each of the multiple self-attention layer’s outputs are concatenated linearly to form Multi-headed self-attention. These attention layers do not share any parameters and hence is able to learn unique relationships with other words/tokens in the sentences. For a bert-base model, there are 12 heads in each of the 12 layers, so there are total of 144 distinct attention weights and scores learned for each of the word. This way, the multiple heads enhance the diversity of attention to obtain diversified semantic vector of each word in different semantic spaces by combining multiple enhanced semantics of each word. During this whole process, we learn the weights for each of the query, key and value vector operation in each of the heads and this whole process is called as Multi-headed Self Attention.

2. Input to BERT: BERT’s Input can represent both a single sentence and a pair of sequences in one token sequence. The first token of every sequence is always a
special classification token ([CLS]). A special separation token ([SEP]) separates the words in each sequence. For a given word token, its input representation is built by adding the corresponding tokens, segments, and positional embeddings as shown in Figure 2.15. Token embeddings are pretrained embeddings using Word-Piece tokenization algorithm [WSC+16] with 30000 words in vocabulary which uses "##" to denote split word fragments. Segment embeddings are sentence number encoded into a vector. For example, is the sentence belongs to first sentence or second, if it belongs to first sentence then a vector of 1’s followed by vector 0’s will form segment embedding. Position embedding account for the position of the word within that sentence that is encoded into the vector. Since BERT is permutation invariant i.e. it does not processes the input sequentially like in RNN. Thus, segment and position embeddings are both used to account for temporal ordering of the inputs.

Figure 2.15: BERT input representation introduced in [DCLT18]

3. Masked Language Model: The task of MaskedLM is described as: Given a sentence, randomly erase one or several words in the sentence, and predict what the erased words are based on the remaining words as referred in cloze task [Tay53a]. Unlike the denoising autoencoder [VLBM08], it just let BERT predict the occluded masks instead of asking to reconstruct the entire input. In this task, just like in the standard language model, the final hidden vector corresponding to the masked token is fed to output softmax function w.r.t to the actual word in vocabulary. BERT takes in sequence/sentence with random words/tokens replaced with [MASK] token and learns to predict these masked tokens by learning conditional relationship between all the words in the sentence as shown in Figure 2.16. MLM is unidirectional and the bidirectional The pre-training process of the BERT model is actually imitating our language learning process. It selects 15% of words in a sentence for prediction. For words that are erased in the original sentence, a special symbol [MASK] is used in 80% of the cases, an arbitrary token is used in 10% of the cases, and the original token remains unchanged in the remaining 10%. The main reason for this is that the [MASK] will not appear in the sequence in the subsequent fine-tuning task, and another advantage of this is that when predicting a token, the model does not know whether the token in the corresponding position is correct. So BERT is forced to maintain a distributed contextual representation of each input token.

4. Next Sentence Prediction (NSP): Many tasks, such as question answering (QA) and natural language inference (NLI), are based on understanding the relationship between two or multiple text sequences, which are not captured by language modeling. In order to train a model to understand relationship across sequences, a binary clas-
Figure 2.16: BERT language modeling with masked token prediction simplified by [Ala18].

Classification task of next sentence prediction is employed. NSP is described as: Given two sequences, determine whether the second sequence immediately follows the first sequence or not. For this the input sentences pairs are combined by [SEP] token and passed to the encoder layers. Sequence A and sequence B have word vectors respectively. In pre-training process, the author randomly selects 50% correct sentence pairs and 50% wrong sentence pairs from the text corpus for training. BERT performs joint training on the MLM task and the NSP task, so that the vector representation of each token/token output by the model can describe the semantic information within the sequence and across the sequences as accurately as possible, shown in Figure 2.17.

Generative Pretrained Transformer 2: GPT2 The encoder-decoder architecture is not the only approach for transformers concerning language modeling. Like Bert which uses stacks of transformer encoder for language modelling, GPT2 uses stacks of transformer decoder. GPT-2 [RWC+19b] is a “Generative Pre-trained Transformer 2” where it is called as Generative as it is used to predict the next word. GPT-2 was trained on even more data as compared to BERT i.e. 40GB dataset crawled from the webtext, and this made GPT-2 very powerful in dealing with different NLP tasks.

The basic idea of GPT-2 architecture is stacking up transformer decoders as high as possible and feed more data through the pre-trained model. And since GPT-2 is made up of decoders, it can only yield one token at a time. GPT-2 is also an Auto-regressive Model because when each token is produced, it is also added back to the sequence of inputs and this new sequence becomes the input of the model in next step. Being an auto-regressive model prevents GPT-2 to capture the context bidirectionally for the sentence and hence can only look forward. Auto-regression is an important functionality in GPT2 which also
makes RNN effective. More details follow the input-output representation used in GPT2 and its decoder blocks in detail:

1. **Byte Pair Encoding (BPE)**: To process input, GPT2 uses Subword algorithm [SHB16] of Byte-Pair Encoding [Gag94, WCG19] to tokenize input to prepare the word vectors. Philip-Gage (1994) proposed BPE, a data compression algorithm [Gag94] which is a mix of character-level and word-level representations and can handle large number of words. It relies on subword units which were extracted using statistical analysis over training dataset. It pays more attention to byte pairs instead of unicode characters which gets generated. The idea of this algorithm is to use a new code to represent the most common bigram in the data (can be byte-pair, character-pair, word-pair, etc.), and iterate until the frequency of the remaining bigrams is 1. BPE not only controls the vocabulary scale, but also greatly alleviates the problem of data sparseness, so it can support better distribution of subwords.

2. **Input & output**: GPT-2, like other transformers performs tokenization using subword algorithm and generates BPE tokens in single list. To refine the input word embedding in its embedding matrix which are added with positional encoding to account for the order of the words, the input sequences start with ‘<s>’ start token and the following inputs/sentences are separated by ‘$’ tokens, we can also use a special token ‘</s>’ as stop token to indicate the input is finished. In case of NL2SQL, the natural language question is followed by ‘<s>’ start token and schema information (table names, column names and datatypes of columns) follows the question but are separated by ‘$’ token and ‘</s>’ at the end. This prepared input is then passed through the decoder layers. After processing the token by the highest decoder layer, the model produces its output vector, multiplying it by the

---

**Figure 2.17**: BERT language modeling with Next Sentence prediction simplified by [Ala18].
embedding matrix. The result of this multiplication is interpreted as a score for each word in the model’s vocabulary. The best token with the highest probability would be selected. To improve the performance of the model, GPT-2 can consider the top 40 tokens with the best score. The model continues iterating until the entire context is generated (1024 tokens) or until an end-of-sequence token is produced. Figure 2.18 shows the process of feeding tokenized inputs to GPT-2 transformer model.

Figure 2.18: GPT-2 transformer architecture simplified by [Ala19].

3. Masked Self Attention: GPT-2 uses stacks Transformer-Decoder blocks in its model which is a stack of decoders as shown in Figure 2.20 in which each block includes a particular Masked Self-Attention layer [LSP+18] which is unidirectional in essence.

Figure 2.19: Self-Attention vs. Masked Self-Attention. [Ala19]

The self-Attention layer in BERT allows the model to pay attention to its right tokens, but Masked Self-Attention layer blocks information from tokens that are to the right of the calculated position. Figure 2.19 shows the difference between the self-attention layer in both architectures. GPT-2 starts to process the tokens by generating actual samples. The model only has one input token at the same time, and only its associated path would be active. Given the vector output, the model can score the vector against its vast vocabulary and returns the token with the highest
probability. Following, the model adds the output into the input sequence, and the next prediction step begins. Thus, masked self-attention prevents the leakage of information by covering the future words using MASK tokens. [AHH19].

4. GPT2 Decoder blocks: GPT2 consists of stacks of decoder followed by 2 feed forward layers, where one of them is called called as 'LM head' where decoders work on the conditional probability distribution of last target vector($Y_{0:i-1}$) and tokens generated on the input contextualized embedding($X_{1:n}$) obtained after subwords of input sentence is passed through multiple layers of decoder. LM head maps this encoded sequence to a logit vector $l_{i+1}$ using a softmax function to get the highest probable token from the probability distribution over the whole output vocabulary as shown in equations Equation 2.15

$$p_\theta(y|X_1:n, Y_{0:i-1}) = \text{Softmax}(f_\theta(X_1:n, Y_0:i-1))$$

$$= \text{Softmax}(W_{emb}^T y_{1-1})$$

$$= \text{Softmax}(l_i)$$ (2.15)

Logit vector $l_i+1$ is learned through multiple conditional operations in various decoder layers, it can also be called as the similarity score of learned encoded output vector with the output vocabulary. All these output conditional probabilities are multiplied together as shown in Equation 2.16[Hug21].

$$P_\theta(Y_1:m|X_1:n) = \prod_{i=1}^{m} p_\theta(y_i|Y_0:i-1, X_1:n)$$ (2.16)

In GPT2 Masked self-attention is unidirectional attention which focus its attention to past and current output tokens generated and does not look into any future tokens. So, in uni-directional self-attention, each query vector $q_i$ is compared only to its respective key vector and all previous ones, namely $k_0,...,k_i$ keys to yield the attention weights which are then multiplies with value vector and summed together to obtain attention over all encoded input vectors.

Figure 2.20: GPT2 Decoder-only transformer blocks: English to German translation [Hug21]
OpenAI released four versions of GPT2\textsuperscript{12} where the larger number of layers were stacked to improve the contextual embedding:

- **GPT2\textsubscript{SMALL}**: 12 decoder blocks, 768 hidden units, and 12 attention heads.
- **GPT2\textsubscript{MEDIUM}**: 24 transformer blocks, 1024 hidden units, and 24 attention heads.
- **GPT2\textsubscript{LARGE}**: 36 transformer blocks, 1280 hidden units, and 36 attention heads.
- **GPT2\textsubscript{EXTRA–LARGE}**: 48 transformer blocks, 1600 hidden units and 48 attention heads.

**A Robustly Optimized BERT Pretraining Approach: RoBERTa**

RoBERTa \cite{LOG+19a} uses GPT’s tokenizer algorithm for tokenization \cite{Kud18, Gag94, AHH19} of input and BERT’s \cite{DCLT19} encoder blocks as its architecture to improve performance of existing BERT. RoBERTa is also trained on even more larger input text along with smarter hyper-parameter optimization to produce a robust and lighter language model.

The optimization in ROBERTA is performed at data level, where 160 GB sized dataset (Book Corpus + Wiki) is used with BPE algorithm \cite{SHB16}. And instead of random masking used in BERT over fixed dataset, RoBERTa copies dataset 10 times during and uses a different mask in each of the copies. Also, the mask generation is dynamic and hence is not predefined during preprocessing. This led to increase of 15-20 Million increased number of parameters as compared to BERT. As part of hyper-parameter optimization, it used following hyper-parameters:

1. Adam, beta1=0.9, beta2=0.999, epsilon=1e-6, L2 weight decay=0.01
2. Learning rate, the first 10,000 steps will increase to 1e-4, and then linearly decrease.
3. dropout=0.1
4. GELU activation function
5. Training steps: 1M
6. batch-size=2k, training 125k steps
7. Input length: 512

RoBERTa provides more refined and robust version of the BERT model. It made improvements to the previously proposed BERT in three aspects. One is the specific details of the model and improved the optimization function. Second is strategy during training i.e. dynamic masking, in NSP (Next Sentence Prediction) task and using a larger batch size; the third is at data level, a larger data set is used with BPE (Byte-Pair Encoding) is used to process text data which increased the vocab size from 30000 to 50000 as compared to BERT’s vocab size.

\textsuperscript{12}https://github.com/openai/gpt-2
Why BERT, GPT2 and RoBERTa?

BERT is the first largest bidirectional transformer model with 345 million parameters with its large model and 110 million parameters with smaller model. Its bidirectional behaviour makes it better to learn alternate language representations. BERT has produced state-of-the-art results as mentioned in Section 2.3.7 in both machine translation, question answering, summarization etc. As our problem of NL2SQL is equivalent to machine translation, we are inspired to use BERT for this task. GPT2 on the other hand uses a subword algorithm to generate Byte-pair encoding output and also only uses decoders which are single directional. It is also reports reasonable results on summarization, question-answering and translation tasks. It is more synchronized with the idea of how most of the languages are read and written. Hence, we select GPT2 for our study to see if it performs better for NL2SQL task. Similarly, RoBERTa is an optimized BERT version which uses GPT2 tokenizer. It also uses dynamic masking instead of random masking and is proven to produce competitive contextualized representation. RoBERTa can be used as both encoder and decoder and hence uses cross-attentions in between the self attention layers. It is thus even larger than GPT2 and BERT as it requires more number of parameters to converge on the similar tasks. The competitive results reported and the difference in attention motivated us to test the performance of these three transformers on the NL2SQL task.

2.4 NL2SQL approaches

Recent deep learning approaches for NL2SQL uses encoder-decoder model where all methods encode the natural language and differ in the ways they use database schema to generate final SQL queries. SQLNet[XLS17] takes in question in natural language $Q_{NL}$ and set of tables and columns. It uses separate RNN for columns and table and also used column-attention over the generated and reinforcement technique to train the decoder to predict aggregate, conditional operator of where conditions and column name over WikiSQL dataset. PT-MAML uses meta learning over $Q_{NL}$ and sequence of column names[HWS18] and TypeSQL[YLZ18] augments it with data-type information for each word in $Q_{NL}$. Both uses single RNN encoder to encode $Q_{NL}$. Similarly, Coarse2Fine[DL18] also uses single RNN encoder but it first generates sketch representation of $Q_{NL}$ by identify variables and arguments and then uses slot filling mechanism to generate the final SQL. SQLNet and PT-MAML both report 68%, Seq2SQL reports 59.4% and TypeSQL reports 77.7 execution accuracy over WikiSQL dataset. IncSQL[STC18] uses Glove embedding and reports 87.1% execution accuracy. SQLova[HYP19] reports state-of-the-art 89.6% execution accuracy using BERT based encoder. Since SQLova is one of the best performing semantic-parsing models using BERT based transformers till July 2020, we select this as the first model for comparative evaluation.

SyntaxSQLNet[YYY18] takes inspiration from SQLNet and uses SQL grammar and builds syntax tree to predict the values for 9 different modules like column modules, operator module, keyword modules etc and predicts the column names and values for each of the modules on spider dataset. Graph Neural Networks[BGB19] uses gated GNN[LBN19] and encodes database schema and builds relationship to form graph and
then passes through encoder and decoder to generate SQL. Following grammar rules of SyntaxSQLNet, IRNet\textsuperscript{[GZG+19]} uses glove-based embedding designed SEMQL grammar which acts as intermediary language between natural language and SQL. It also uses table and column schema information and encodes it using a schema encoder. Next follows a grammar-based decoder which takes in $Q_{NL}$, schema encoded information and checks for grammatical mistakes at each step and generates SEMQL representation instead of prediction SQL keyword containing sequence. This way it is able to fix the grammatical or syntactical errors at each time step. SyntaxSQLNet reports 19.7% execution accuracy, GNN reports 47.4% execution accuracy and IRNet reports 46.7% execution accuracy using Glove embedding and 54.7% execution accuracy using Bert on Spider dataset. Since IRNet produces state-of-the-art results(June 2020) on Spider dataset, hence we use it as the second model for our comparative evaluation.

The goal of this work is to perform comparative evaluation of Semantic parsing model called as SQLova\textsuperscript{[HYPS19]} and grammar-based parsing model called as IRNet\textsuperscript{[GZG+19]} for NL2SQL task especially in context of different types of transformers\textsuperscript{[WDS+20, DCLT19, LOG+19a, AHH19]} which employs different types of attention mechanisms\textsuperscript{(Subsection 2.3.4)} to generate better learned contextualized vectors for input, also called as embeddings (Subsection 2.3.6).

### 2.4.1 SQLova

SQLova\textsuperscript{[HYPS19]} is a BERT based encoder-decoder architecture proposed by Naver Corp in NeurIPS 2019 conference for semantic parsing of Natural language questions to SQL queries which works on WikiSQL dataset for single table queries. It implements 3 different modules in increasing order of complexity for specific tasks which takes the word contextualized embeddings from BERT transformer and sends it to a shallow layer, decoder layer and NL2SQL Decoder layer which takes in the concatenated output of shallow and decoder layer. It produces competing logical form accuracy of 83.6% and execution accuracy of 84.4% on WikiSQL dataset’s test set. Following are the details of SQLova modules :

1. **Word Contextualization using BERT** : NL question and table headers are separated using [SEP] tokens without using data from the table, with [CLS] token at the start. This is fed to the shallower layers of SQLova encoder containing 1 recurrent LSTM layer followed by a linear fully-connected layer.

2. **Shallow Encoder Layer** : It takes encoded representations as input from table aware Bert encoder and predict the probability of “Select Column”, “Select aggregation”, “Where column”, “where operator”, “where number - number of where conditions” given the question and table headers as shown in Figure 2.21. It also predicts where values by predicting the probability of each of the natural language tokens is selected as the start index of where value corresponds to the condition values for “Where column”.

3. **Decoder Layer** : Decoder does not generate the entire table header, but it generates sequences of pointers to augmented SQL vocabulary, natural language question tokens, table header tokens, start and end tokens to be used during the inference stages using LSTM decoders. Pointer to SQL module generates the final SQL query
Figure 2.21: SQLova shallow encoder layer [HYP19]

using attention as a pointer to select start and end tokens for where-value over header tokens as input. Shown in Figure 2.22

Figure 2.22: SQLova Decoder layer [HYP19]

4. NL2SQL Layer (SQLova): It takes in the input and output from shallow layer and decoder layer as concatenated inputs and predicts the probability of SQL tokens for each of the 6 modules as shown in Figure 2.23. The output form table aware BERT encoder is again encoded again by LSTM question encoder and LSTM header encoder on which column attention calculated over table header is applied which is
adapted form SQLNet[XLS17]. The only difference here is that SQLova predicts the start and end tokens for where values, which depend on number of where conditions, where column and where operator’s output.

![Figure 2.23: SQLova NL2SQL layer [HYPs19]](image)

5. Query generation: Execution guided decoding[WTB+18] is performed to generate final SQL query. For example, select column and aggregation operator pair for string column are excluded and the highest joint probability of both are taken into account.

6. Evaluation: SQLova reports logical form accuracy and execution accuracy on dev/-validation and test set of WikiSQL[ZXS17] dataset. Execution accuracy is reported by executing the generated SQL query over the database and logical accuracy is reported by matching the individual column and value prediction over the 6 sub modules in and NL2SQL layers.

SQLova does not share parameters in 6 decoders and predicts where clause values using span detection instead of pointer generators. Next, let’s see the SQLova decoders in detail. As mentioned in Figure 2.23, each of the 6 decoders uses column-attention[XLS17] over the contextualized embeddings obtained from BERT. The notations used to explain the decoders are: $D_c$ is header(c) encoding vector and the LSTM output is denoted by $E_n$ for the nth token. $W$ are the weight vectors learned at every layer. $C_n$ is the context vector for the NL question. The 6 decoders are explained as follows:

1. **select column decoder**: Returns the column id from the contextualized header vector. $C_n$ attention vector obtained first with softmax taken over column attention output which is calculated over the affine transformation of $D_c^T$ and $E_n$ vectors done in LSTM cells. And then applying the softmaxed operation over $E_n$ encoded question token to finally get attention vector $C_n$. As mentioned in the following equations, $p_{sc}(c)$ returns the probability column c using softmaxed operation over the affine transformation of $W$ vector with concatenated tokenized header $D_c$ and column attention vector $C_c$ using fully connected layers:

$$s(n|c) = D_c^T W E_n$$
$$p(n|c) = \text{Softmax}(s(n|c))$$
$$C_c = \sum_n p(n|c) E_n$$

$$s_{sc}(c) = W \cdot \tanh([W D_c; W C_c])$$
$$p_{sc}(c) = \text{Softmax}(s_{sc}(c))$$ (2.17)
2. **select aggregation operator**: it returns the index of aggregation operator over the list of 6 possible choices [NONE, MAX, MIN, COUNT, SUM, AVG] over the context vector $C_c$ obtained in the last step which is then passed through fully connected layers and a softmax function is applied over it. The operation is denoted as:

$$p_{sa}(agg|c) = \text{Softmax}(W(\tanh(WC_c))) \quad (2.18)$$

3. **where number decoder**: $s_{wn}$ returns the number of where conditions to be predicted for the given contextualized NL question denoted as $C_Q$ conditioned on contextualized column vector $C$. This module is derived from SQLNet[XLS17] and is explained with the following equations:

$$p(c) = \text{Softmax}(WD_c)$$
$$C = \sum_c p(c)D_c$$
$$h = WC$$
$$c = WC$$
$$p'(n) = \text{Softmax}(Wbi - \text{LSTM}(E_n, h, c))$$
$$C_Q = \sum_n p'(n)E_n$$
$$P_{s_{wn}}(c) = \text{Softmax}(W\tanh(W \cdot C_Q)) \quad (2.19)$$

Here, $h$ is the hidden input and $c$ is the cell state input to LSTM encoder[HYP19]. Here, first a distribution over column vector is generated followed by the affine transformation into 2 vectors $h$ and $c$. These vectors along with encoder hidden output vector is passed through more affine transformation later followed by softmax function to generate distribution over

4. **where column decoder**: predicts the probability $p_{wc}$ of generating each is calculated over the column attention output calculated over affine transformation of concatenated encoded header vector $D_c$ and column vector $C(c)$. The highest probability columns(top k) are selected over the number of where conditions(k) returned by the where number decoder as mentioned by the following equations:

$$s_{wc}(c) = W \cdot \tanh(WD_c; WC_c)$$
$$p_{wc}(c) = \text{Sigmoid}(s_{wc}(c)) \quad (2.20)$$

5. **where operator decoder**: returns the index of operator from operator list [$=, <, >$] conditioned over column vector $c$. The index is obtained by applying softmax over affine transformation of concatenated encoded header vector $D_c$ and column vector $C(c)$ passed through fully connected layers having Tanh non-linear activation function. The column vector is reused from where column decoder.

$$s_{wo}(op|c) = W \cdot \tanh(WD_c; WC_c)$$
$$P_{wo}(op|c) = \text{Softmax}(s_{wo}(op|c)) \quad (2.21)$$
6. **where value decoder**: This decoder returns the probability of the $n^{th}$ token of question being the start index and similarly being the end index to find the start and end positions from the question containing the where value [DL18].

\[
\begin{align*}
\text{vec} &= [\text{En}; \text{WC}_c; \text{WD}_c; \text{WV}_{\text{op}}] \\
\text{s}_{\text{swv}}(n|c, \text{op}) &= \text{Wtanh}(\text{Wvec}) \\
\text{P}_{\text{swv1}}(n|c, \text{op}) &= \text{Softmax}(\text{Wtanh}(\text{Wvec})) \\
\text{P}_{\text{swv2}}(n|c, \text{op}) &= \text{Softmax}(\text{Wtanh}([\text{Ws}_{\text{swv1}}; \text{Wvec}]))
\end{align*}
\]

Here, $V_{\text{op}}$ is the one-hot vector over operator list. $s_{\text{swv1}}$ is the start index and $s_{\text{swv2}}$ is the end index of the question vector. To get the end index, the first predicted output of start index is passed again. If not end index if predicted or if the predicted end index is less than start index then the length of the string is passed as the value for second index.

### 2.4.2 IRNet

IRNet is a recent model proposed by Microsoft in ACL 2019 which works on cross domain context free grammar based approach to generate intermediate grammar called as SEMQL to bridge natural language and SQL generation step as shown in Figure 2.24[GZG+19]. It also fixes the mismatch problem because of difference in ordering of where conditions. This model is implemented on complex cross-domain dataset Spider[YZY+] which consists of a wide range of simple and complex queries in cross domain settings. IRNet is an encoder-decoder architecture with a NL encoder to encode the question and question token types, a schema encoder to encode schema information followed by a decoder to synthesize SEMQL query as output of the model as shown in Figure 2.25. It reports Exact Set Match Accuracy of 53.2% for dev set and 46.7% for test set using Glove embeddings. And IRNet+Bert(base model) reports Exact Set Match Accuracy of 61.9% for dev set and 54.7% for test set.

**Understanding SEMQL** : Intermediate query representation

SEMQL bridges the gap between Natural language and SQL query by constraining the search space during synthesis and gives better results when coupled with beam search algorithm in the decoder. The inspiration of the tree based structure is taken from SyntaxSQLNet[YZZ+18]. It is designed to be an abstract syntax tree obeying the rules of SQL syntax and once the tree is built, Dijkstra’s algorithm is used to find the shortest path to get all the table names from the tree and connect the logical forms together. The labels to be predicted in order to generate SEMQL query corresponds to Sup(Superlative), Sel(Select column id), Order, Root, Filter, A(Aggregation), C(Column) and T(Table) with Root1(Select node : starting node) as default value fed to the decoder. Idea is that the model first builds logical forms and then build the CFG tree from bottom-up by identifying relationship with the parent nodes.

SEMQL components are explained more in detail here:
1. **Root1**: Since all queries start with SELECT as NL2SQL task focuses mainly on DQL statements, the first node which is fed as the first input to the decoder followed by the context vector generated by encoder is always Root1. It is interpreted as starting node for the query pointing to the SELECT keyword.

2. **R and Select**: R and Select are called as the relation node which is either the parent or starting node of a subtree and it implies nodes for nested or join query. This node is either the parent node of a subtree or child node of the Root1 node. R node will always start with Select value derived from Select node followed by the Order, Superlative, Filter nodes to form conditional clauses. Select node is a combination of single or multiple A nodes. * This Select is not the "SELECT"
3. **A**: stands for the logical forms whose parent node is the node containing relation name over which the aggregation operation in A has to be performed. Types of operations covered are max, min, count, sum, average, and none. A is always a child node pointing to the column name of the table T.

4. **Z**: Z stands for the node capturing single or multiple relations (table names) and next node is predicted from Union/Intersect and Except labels are predicted with label-id (0,1 and 2 respectively).

5. **Order**: order node can have 2 values asc or desc denoted by 0 and 1 respectively. These nodes contain another child node containing value in string/number format, which represents the column names of the table.

6. **Filter**: Filter node is forms the smaller logical forms like >, <, = etc operations followed by operator, column name or value/R containing nested select nodes.

7. **T**: stands for Table node

8. **C**: stands for Column node

**IRNet modules**

IRNet architecture is shown in Figure 2.24. It solves end to end natural language to SQL question conversion/parsing task in 3 phases along with evaluation explained as follows:

1. **Schema linking**: Relationship of the relations and column names is identified over tokenized schema and question to identify columns and table names in the question and identifying cell values in the question assuming unidentified strings within quotes as values to be predicted without looking at data in database. This is done by schema encoder. As can be seen, Input question is passed through BERT attached NL encoder and schema information through BERT attached schema encoder which are then concatenated and passed through decoder units.

2. **Intermediate SemQL grammar generation**: over natural language question is done using glove or BERT embeddings. NL Encoder takes natural language tokens and assigns types to each token identified and Schema Encoder takes database schema as input. Tables and column representations identified during schema linking as outputs. Schema encoder also performs attention over span embeddings generated over schema information to obtain the context vector. It takes over initial embedding, context vector and type embeddings over column

3. **Decoder**: Decoder takes in the first token, schema and NL encoder’s concatenated hidden context vector and generates sequence in SEMQL grammar, as soon as the next token predicted is type T node, the decoder’s following output goes through fully connected layers and softmax function to first output if the relation name to be predicted is from memory or schema (information vector generated by schema encoder). It uses memory augmented pointer network where the first priority is given to identify columns from memory and then from pointer networks to find table from identified columns using attention based fully-connected layers followed by softmax
function. While identifying entities such as columns, tables and values the preference is given in the mentioned decreasing order: Decoder identifies 3 actions apply rule, select column and select table and then predicts operation index or column index labels as values for: Sup(Superlative), Sel(Select column id), Order, Root, Filter, A(Aggregation), C(Column) and T(Table) nodes or label outputs. This way it checks for grammatical errors at each time-step during decoding.

4. **Evaluation in IRNET**: IRNet reports exact matching accuracy over end to end SQL query and component matching accuracy and component matching F1 score over Select, From, Groupby, Orderby and from clause over Glove and BERT embeddings on Spider dataset. We report exact matching accuracy in this thesis.

### 2.5 Summary

In this chapter, we discuss in SQL language history, syntax and relations followed by the evolution of NL2SQL task, its different approaches and challenges. We explain the evolution of NLP using deep learning models which has accelerated the direction of research in this field. The later section include the basic and somewhat more technical outline of recurrent neural networks and sequence to sequence architecture evolved because of the progress in deep learning. Then we also explain the Attention concept which has taken the NLP community by storm, followed by transformer models using attention mechanism in detail. Lastly, we explain the architecture and concepts related to grammar-based model(IRNet) and parser-based model(SQLova) which is the focus of this thesis.
3 Design

In the previous Chapter 2, we presented the necessary background knowledge concerning the topics covered in this thesis. In this chapter, we define precise research questions addressed in this research, with the design of NL2SQL systems and architecture overview. We explain why the proposed design of the approach seeks to serve well the task of neural machine translation using transformer models. We also explain the datasets and the evaluation measures used with our NL2SQL design. This chapter is structured as follows:

- In (Section 3.1), we begin by establishing the research questions to be evaluated in our work.
- In (Section 3.2), we present the model architecture using attention-based transformers with SQLova and IRNet models, as baselines for NL2SQL task
- In (Section 3.4), we explain the evaluation measures used to evaluate NL2SQL models
- Finally, we provide a summary of the main topics discussed in this chapter (Section 3.5)

3.1 Research Questions

This research initiative is an attempt to understand how well the most recent attention-based transformer architectures (BERT, GPT2 and RoBERTa) perform on the task of NL2SQL neural machine translation problem.

Based on the literature review we are able to specify the specified set of evaluation questions we intend to answer in this thesis:

1. What tuning choices play a large role in the performance of a semantic parsing model on complex cross-domain datasets? (Model: SQLova)

2. To what extent can the performance of a grammar based model be improved and generalized on a complex cross-domain datasets? (Model: IRNet)

3. To what extent does word contextualization using various transformer models improve the performance of parsing and grammar-based models? (Model: GPT2, Bert, RoBERTa)
3.2 Pipeline

For our evaluation, we preprocess the Spider\cite{YZY19} and Sparc\cite{YZY19} datasets separately for both models SQLova\cite{HYPS19} and IRNet\cite{GZG19} by adapting the preprocessing techniques provided in WikiSQL\cite{ZXS17} and spider papers respectively. We also adapted to the existing code to use the hugging-face transformer api to get a contextualized vector from the transformer models Bert\cite{DCLT19}, GPT2\cite{AHH19} and RoBERTa\cite{LOG19a} for both the NL2SQL models.

For our work, we require three main components. First component is a pre-processing algorithm for Sparc and Spider to convert to WikiSQL format, pre-processing algorithms to convert Sparc to Spider format. Second component requires the script to load hugging face api’s tokenizer, trained model and adapt the input format to feed NL query and schema information and output formats to feed to the respective encoders of SQLova and IRNet models. It also requires modification to the output generated by decoder to get the predicted BPE based output in the case of using GPT2 with IRNET. Third script to experiment with different hyper-parameters, evaluate and generate relevant visualization for the comparative evaluation of NL2SQL models.

We follow the following pipeline to experiment with SQLova and IRNet models:

![Figure 3.1: NL2SQL pipeline](image)

3.3 NL2SQL Models with Transformers

In this section we present our design for the use of alternative transformers on grammar-based and parsing-based NL2SQL models. We use IRNet and SQLova as two examples of these model classes. For each case we explain in detail the aspects of adopting the tokenizers and input/output of transformers, in addition to connecting this to the input/output

\[\text{https://openai.com/blog/better-language-models/}\]
of the NLP2SQL models. We provide details also in the context of 2 datasets, which we present in detail in a later chapter.

### 3.3.1 Grammar-based model with Transformer models

We use transformer models BERT, GPT2 and RoBERTa to generate better contextual embeddings. Our choice for these models is already explained in Section 2.3.7. We implement IRNet as a baseline model and replace BERT transformer with GPT2 decoders and RoBERTa encoders and train to generate better word contextualized embeddings, which is our main contributions to this thesis.

**Figure 3.2:** Modified IRNET with Transformer Models GPT2 or RoBERTa to encode schema and Natural language question, adapted from IRNET paper [GZG+19]

Bert uses bidirectional attention, GPT2 uses masked self-attention and RoBERTa uses dynamic masking mechanisms which when coupled with IRNET’s memory augmented pointer network, should result in better prediction of columns and actions to generate SEMQL intermediate query. Pointer networks with better contextualized embedding will first look into the memory so as to prevent the prediction of same columns from schema. And schema encoder with transformer networks applies dot product attention over span embedding to get a context vector by summing together column type embedding and initial embedding from transformers. Different context vectors of different sizes are obtained from BERT, GPT2 and RoBERTa and are then passed through IRNet decoder separately. Decoder then generates SEMQL queries using bidirectional LSTM cells followed by Pointer networks to predict which action i.e. SELECT COLUMN or APPLY RULE to take over which column for a given natural language question.

The input SQL queries, table information (columns names) and header information (start and end indices of question in tokens and header) were fed as as single sequence to transformer models on top of schema encoder connected to memory-augmented pointer
network as part of the decoder. And NL based input question is passed separately to NL encoder. For this, the encoder part of the code to generate input sequence has to be modified to be fed to the corresponding transformer models:

- **GPT2 tokenization**: The input question and columns from header information for each batch is converted to: `<s>` Natural Language question `</s>` col1 `</s>` col2 `</s>` ... col-n `</s>`. where `</s>` acts as separator tokens. Since GPT2 tokenized uses BPE, a list containing token id for each of subwords is flattened and kept as reference(index id) to word tokenized list of NL question and column information used later during prediction of tokens(detokenization). Id based list generated out of input sequences are padded with 0’s to fit to the maximum sequence length of 512 allowed in the batch. This input is then further tokenized by the GPT2 tokenizer.

- **GPT2 model input**: In order to get the final output by GPT2’s last decoder, the sub-tokens of tokens and their indexes are built using lists which when passed through inbuilt `tokenize` and `convert_tokens_to_ids` method of tokenizer class to get single and flattened list of sequence of id’s for the input sequence (NL+column). Similarly, mask array (tensor objects in pytorch) for the input sequence is generated for the valid input id’s returned by the tokenizer. And these tensors are then passed as input to the trained GPT2 model.

- **GPT2 model output**: GPT2Model class is passed the input_ids, attention_mask and when trained, it returns last the last hidden state. This final output vector is then passed through Schema and NL encoders separately.

- **RoBERTa tokenization**: Tokenization in RoBERTa is derived from GPT2’s tokenizer² and hence uses BPE encoding algorithm to tokenize input sequences. Hence, we use the same input format as used in IRNet+GPT2, which is then further tokenized by RoBERTa tokenizer. The difference lies in the internal strategy of RoBERTa which treats spaces as part of the tokens, so tokens which were preprocessed for special characters will have extra spaces introduced between then. This created space is then treated as part of the token by the tokenizer.

- **RoBERTa model input**: In order to get the final output by Roberta’s last encoder, the sub-tokens of tokens and their indexes are built using lists which when passed through inbuilt `tokenize` and `convert_tokens_to_ids` method of tokenizer class to get a sequence of id’s for the input sequence (NL+column). Similarly, mask array (tensor objects in pytorch) for the input sequence is generated for the valid input id’s returned by the tokenizer. And these tensors are then passed as input to trained Roberta model.

- **RoBERTa model output**: RobertaModel class is passed the input_ids, attention_mask and when trained, it returns last hidden state of the encoder. This output is then converted to a list object.

- **IRNet input**: Output of corresponding GPT decoder or Roberta encoders is passed separately through the fully connected layer, Bi directional-LSTM layers of NL encoder. Over which the attention vector is calculated over span embeddings to calculate context vector and column representation vector is generated by adding type embedding vector. In Schema encoder of IRNet, RoBERTa encoder’s embedding

---

²https://huggingface.co/transformers/model_doc/roberta.html, last visited: 21 Feb 2021
vector or GPT2 decoder's embedding vector and the new schema context vector is passed to generate the final vector. The IRNet model is then trained with already trained Roberta and GPT2 model respectively.

- **IRNet output**: The final combined vector from schema and NL encoder passes through more fully connected layers first predict if the output should be the action to take (node type) or value to be predicted. On the basis of previous value which is then fed to the decoder, the Rule label of type Root, Select, Filter, Order, Sup and node is selected in case of node/action to take. Else, the value if selected using the same final vector which then passes through fully connected layer of one node to output if the next value should be selected from memory or schema information as part of input. This output along with final vector is then passed through pointer net decoder to output index of column or table id from the schema or memory and then depending on the node action and type of memory/selection binary value selected as the last output.

- **IRNet Decoder**: For Sparc dataset, we extend this decoder by passing the additional previous hidden context vector to take into account the previous question and turn passed. This is one of our our contributions in the thesis.

### 3.3.2 Parsing-based model with Transformers

SQLova[HYPS19] is a neural semantic parsing model that already uses a BERT encoder model to encode table schema information so as to get better word contextualized embedding over WikiSQL dataset. We use GPT2 decoder-based transformer model with SQLova to generate competing contextualized word embeddings as that of BERT as already mentioned in Subsection 3.3.1. BERT and GPT2 both are fine tuned for NL2SQL parsing task and we have the best models for both BERT and SQLova+BERT. Because of fine-tuning, the training model is also very slow. The modified architecture of SQLova is shown in Figure 3.3. This is one of our main contributions.

![Figure 3.3: SQLova with GPT2](image-url)
The following steps are taken to get the encoded output from GPT2 and retrain the model with SQLova:

- **GPT2 tokenization:** The input question and columns from header information for each batch is converted to: `<s>` Natural Language question </s> and Header Columns as `<s>` col1 </s> col2 </s> ...col-n </s>. where </s> acts as separator tokens. Since GPT2 tokenized uses BPE, a separate list of list containing index of subtokens generated for each sub-token is kept which as reference(index id) to word tokenized list of NL question and column information. Sequences are padded with 0’s to fit to the maximum sequence length of 512 allowed in the batch. This input is then further tokenized by GPT2 tokenizer.

- **GPT2 model input:** In order to get the final output by GPT2’s last decoder, the sub-tokens of tokens and their indexes are built using lists which when passed through inbuilt `tokenize` and `convert_tokens_to_ids` method of tokenizer class to get single and flattened list of sequence of id’s for the input sequence (NL, header columns). Similarly, mask array (tensor objects in pytorch) for the input sequence is generated for the valid input id’s returned by the tokenizer. And these tensors are then passed as input to the trained GPT2 model.

- **GPT2 model output:** GPT2Model class is passed the input_ids, attention_mask and when trained, we take the last hidden state. The final vector outputted from the last hidden state is then passed through 6 SQLova module’s header encoder and question encoder separately to generate column attention output respectively.

- **SQLova input and output** Output of corresponding GPT3 decoder or Roberta encoders is passed separately through the header encoder and column encoder of the 6 modules of SQLova. The detailed operation is explained in Chapter 2 chapter. The SQLova model is then trained with already trained Roberta and GPT2 model respectively. The predicted output from each of the 6 blocks is passed through post-processing script to fill the slots of each of the 4 logical forms to generate final predicted SQL.

### 3.4 Evaluation of NL2SQL models

Evaluation measures should evaluate systems on how well it generalizes on unseen data. Hence, the test set on which the system is going to evaluated should have separate distribution, such that even though the entities such as tables, column names referred by the query might be same but the intent of query i.e. the results returned by queries should be different. Thus, splits should be based on SQL query and not on the natural language question. As suggested by Finegan Dollak et al.(2018), the Yale lily lab team designed Spider[YZY+] and later Sparc[YZY+19] datasets and used database splits to ensure that the SQL query generated is entirely different for each of the train, dev and test sets. We evaluate over dev set and use the following evaluation metrics to report the performance of the NL2SQL models:
3.4.1 Exact Match Accuracy

For each of the logical forms (LF) or SQL clauses like select column, select aggregation, where number, where column, where operator, where value exact "string matching" is performed on generated $Q'_{SQL}$ and ground truth $Q_{SQL}$ SQL query clauses.

Predicted SQL statements denoted by $SQL'$ is matched with $SQL$ that are completely correct. Among them, the order of the columns does not affect the calculation of accuracy. The calculation formula is as follows:

$$
Score_{match} = \begin{cases} 
1, & SQL' = SQL \\
0, & SQL' \neq SQL 
\end{cases}
$$

$$
Acc_{match} = \frac{1}{N} \sum_{n=1}^{N} Score_{match}^{n}
$$

(3.1)

The predicted query is matched with the ground truth and is true only if each of the components such as SELECT, WHERE, ORDER BY, AGGREGATION, GROUP BY, KEYWORDS (INTERSECT, IN, NOT, JOIN) are correct. This exact matching metric can handle the "ordering issue" between multiple components of SQL as mentioned in work done for the SQLNet model [ZXS17].

3.4.2 Sketch Accuracy

A sketch is an outline of an incomplete SQL query, with gaps (e.g. missing the content of the FROM clause) which has to be completed. Sketches can have hints for each gap or missing entry that give some direction to the model as to how it needs to be filled. Sketch accuracy is good when the models predicts the right next node or right next value according to the given question and schema information.

$$
Score_{sketch} = \begin{cases} 
1, & SEMQL' = SEMQL \\
0, & SEMQL' \neq SEMQL 
\end{cases}
$$

$$
Acc_{sketch} = \frac{1}{N} \sum_{n=1}^{N} Score_{sketch}^{n}
$$

(3.2)

Sketch accuracy relies on the SEMQL synthesis of such partial forms or SQL clauses generated during SQL generation using local SQL clauses generation [BTGC16]. Sketch accuracy is calculated with the predicted label for each of the partial forms i.e. Sup, Sel, Order, Root, Filter, A, N, C, T labels. The IRNet paper reports sketch accuracy of the decoder to evaluate its performance of generating next node and its values (SQL functions) of the SEMQL grammar after seeing the previous nodes and previously predicted values. We use the existing code to evaluate sketch accuracy of the decoder provided by the IRNet team to for our experimentation with hyper-parameters and use of transformer models to improve its performance. Spider [YZY+] and Sparc [YZY+19] have different SQL queries in each of the training, validation and test sets, we use sketch accuracy [DL16b] to report the performance of IRNET [GZG+19] w.r.t to the SEMQL tree based output.
3.4.3 Execution Accuracy

It measures if predicted SQL execution result in terms of number of tuples/rows is consistent with the execution result of ground truth SQL. This is done by executing the query generated by NL2SQL decoders in the database and return the average number of rows returned denoted by $Y'$. Finally, average matching score between the predicted number of rows $Y'$ and the true number of rows denoted by $Y$ for ground truth SQL query is calculated to return the execution accuracy as denoted by the following equations:

$$Score_{ex} = \begin{cases} 1, & Y' = Y \\ 0, & Y' \neq Y \end{cases}$$

(3.3)

$$Acc_{ex} = \frac{1}{N} \sum_{n=1}^{N} Score_{ex}^n$$

We use this to report Execution accuracy of SQLova and SQLova+GPT2 model.

3.4.4 Loss

**Negative Log Likelihood Loss or Negative Log Loss (NLL)**

$$L(y) = -\log(y)$$

(3.4)

It maximizes the high confidence or high probability over the correct predictions and discourages the model with low confidence values or low probability for correct labels and thus encourages high predicted probability for correct labels. Negative log likelihood loss works in conjunction with softmax function which is used at the classification layers of IRNet model. As mentioned above, the input of NLL is a probability and we learn conditional probability for NL and question pairs. IRNet minimizes negative log likelihood loss.

**Cross Entropy Loss (CE)**

Cross-entropy as a loss function used to learn the difference in probability distributions of the data. CE loss measures the cross-entropy between the predicted probability and the actual value.

$$Loss(x, y) = -\sum x \log(y)$$

(3.5)

where $x$ is probability of true label and $y$ is probability of predicted label.

Unlike NLL loss which only takes into account the correct predictions, CE penalizes more if incorrect predictions are predicted with high confidence. SQLova minimizes the cross-entropy loss for each of the 6 modules. CE loss is calculated for classifying over column labels, operator labels for where column, select column, where operator and aggregation operator modules of SQLova.
3.5 Summary

In this chapter, we finalize our research questions, formulated to be answered as part of this thesis. We present final architecture designs for using pre-trained transformer models on NL2SQL task. We also explain the datasets preprocessing steps required to be used with the respective models. Finally, we conclude this chapter by providing information about the evaluation metrics to be used as part of this thesis. We report execution accuracy for both the models, IRNet and SQLova.
4 Experimental Setup

In this chapter we present and discuss the key configurable components of the experiments we propose to address the research questions raised in Chapter 3. This chapter is structured as follows:

- In (Section 4.1), we explain the modifications adopted to work with cross-domain datasets as part of the thesis.
- In (Section 4.2), we explain the hyper parameters used to improve the performance of baseline models.
- In (Section 4.3), we provide the configurations of machine and software libraries used.
- Finally, we provide summary of this chapter (Section 4.4)

4.1 Datasets Preprocessing

There are already some datasets publicly available to study NL2SQL models. Among them we can name: WikiSQL, Spider, Spark and CoSQL.

WikiSQL is a NL2SQL dataset sponsored by Salesforce. It consists of 80k queries over 26k different tables, with each query addressing a single table. The data itself is crawled from Wikipedia tables, and the SQL statements and questions were generated via crowdsourcing.

Spider is a dataset produced by researchers from Yale. It consists of 20k queries, addressing 200 different databases covering diverse domains. This dataset was manually curated by the researchers, and it has queries that span many tables, with an average of 5.1 tables per query. In contrast to WikiSQL, Spider also represents queries having Order By, Group By and other more complex clauses.

Sparc is a NL2SQL dataset created by both Salesforce and researchers from Yale. Unlike previous datasets, it considers sequences of queries. Sparc was developed with the intention of helping researchers to build and test models able to use context and query refinement behavior from users, to improve the NL2SQL task. Sparc consists of 12k questions, over 200 databases from different domains, grouped into 4k sequences. In addition, it has complex clauses, like Spider.

Finally CoSQL is an extension to Sparc that adds further aspects likely to appear in a conversational system, such as unanswerable questions, or questions where user intent recognition is important.
For our research we have used WikiSQL, Spider and Sparc, according to the possibilities identified in the models. In this section we talk about the pre-processing required to use these datasets.

Spider[YZY+19] and Sparc[YZY+19] datasets have longer input NL sequences and more table header information. When processing Sparc dataset is converted to Spider dataset format to experiment it with grammar-based IRNet[GZG+19] model. And Wikisql[ZXS17] to experiment with parsing-based SQLova[HYP19] model by finding the values in schema information. The detailed preprocessing is mentioned as follows:

### 4.1.1 Preprocessing for IRNet

A table vocabulary is generated separately in IRNET for memory augmented pointer network to find the relevant column name in decoding stage using vocab.py file provided by the IRNET-Microsoft team at the github page\(^1\).

**Spider dataset preprocessing**

1. Filter special characters and repeated quotes to clean the data for the NL question
2. Use nltk’s WordNet Lemmatizer to tokenize the question and then convert all tokens to lowercase, put into ‘question_toks’ key
3. For each of the table schema information from tables.json file. And for each table obtained for the database_id from train_spider.json file, get the table names and append the type="table" for TYPE list which is passed with the input formatted query. Since some of the column names are either incomplete and hence only partially match, a separate partial match function iterates over all columns and match each sub-string’s existence is checked in the header information. This is done using partial_header function. With this we get the type of the information to be obtained for question. The question_arg_type for SQL query 'how many singer do we have ?' looks like : 
   ```json
   ["NONE"], ["NONE"], ["table"], ["NONE"], ["NONE"], ["NONE"], ["NONE"], ["NONE"]
   ```
4. With all the columns appended to one list, we need to identify which column belongs to which table id of that database, for this a col_table output is generated where for each entry of tables names the index of the table name is repeated for the column names of that table. The database info containing singer table is parsed as :
   ```json
   "table_names": ["stadium", "singer", "concert", "singer in concert"]
   "col_table": [-1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3]
   ```
      to extract singer table which is table of index 1(singer) from table_names key. Here, index -1 points to no table found.
5. Part-of-speech(POS) tagging is generated for each token of question using nltk.pos_tag : output of this steps looks like :
   ```json
   ["how", "WRB"], ["many", "JJ"], ["singer", "NN"], ["do", "VBP"], ["we", "PRP"], ["have", "VB"], ["?", "."]
   ```

---

\(^1\)https://github.com/microsoft/IRNet/tree/master : last accessed: 20 Feb 2021
6. We use SQL2SEMQL function provided by IRNet which generates the rule_label which is the CFG-grammar-based tree for corresponding SQL. The output rule_label looks like: "Root1(3) Root(5) Sel(0) N(0) A(3) C(0) T(1)"

We use the preprocessed dataset containing SEMQL representation as labels and pass it to the transformer models like GPT2 and RoBERTA instead of passing SQL query itself, to report the performance over these competing transformer language models. For this we develop custom methods to point to the GPT2 and RoBERTa vocabulary of tokens in order to get byte-pair representations of the tokenized input sequences and send it to the Model. After that, get the predictions from decoder to generate the final utterance, which is a SEMQL query in case of IRNet. This is one of our main contributions for this thesis.

Sparc dataset preprocessing

Sparc\[YZY^+19\] is modified to fetch single queries independent of multiple turns in which they were asked by user to be experiment with IRNET which processes one query at a time. The similar preprocessing steps was adapted with just one modification for each of the interaction, question-query pair and its schema information was fetched as separate inputs. This is done by iterating over each interaction and fetching each of the utterance and query combination. And then we apply the same preprocessing steps as mentioned for spider dataset in above section are applied to generate nltk_pos and question_arg_type information and SEMQL label is generated. For each of the turn under interaction, we add a key of "turn" and "count of interaction" as value. This is added to the schema information. This is one of our main contributions to generalize IRNet model over Sparc dataset.

4.1.2 Preprocessing for SQLova

For SQLova, WikiSQL input data is preprocessed to tokenize SQL query and natural language question using Stanford nltk library as mentioned in the Section 4.3. Preprocessing steps are follows:

1. Convert to lowercase using nltk library
2. **Tokenize**: use nltk library sentence split(sstring) method to tokenize the input to get individual tokens in a list. Tokenize, NL question, schema header information such as column names of the table and SQL query.
3. **Query**: Generate query object pass the tokenized output to it. This will identify iteratively, if each of the token belongs wither of these: [aggregation operation, matches partially to a column name, conditional operator, matches partially to table name]
4. **wvi_corenlp**: For each of the value found while processing Query object, if none of the tokenized word matches to the Query expected label, then the input is labels as value to be predicted. Index of this value is partially matched in the question_toks and founded position for the value if added as a list of [start index, end index]. If
the partial matching is failed then the length of question tokens is returned as end index.

5. Get the rest of the schema information key-value pair as is

6. Generate turn number according to order of the interaction and append it to the schema information.

The output of example preprocessed query of wikisql query looks like: table_id"1-10015132-11,"question": "List all the salary values players received in 2010 and 2001.", "question_tok": ["List", "all", "the", "salary", "values", "players", "received", "in", "2010", "and", "2001", "."], "query": /"agg": [0], "sel": [276], "conds": [[272, 2, 2010.0]], "query_tok": ["SELECT", "salary", "FROM", "salary", "WHERE", "YEAR", ",="", "2010", "UNION", "SELECT", "salary", "FROM", "salary", "WHERE", "YEAR", ",="", "2001"], "table_id": "baseball_1", "wvi_corenlp": [[8, 8], [10, 10]], "sql": "agg": [0], "sel": [276], "conds": [[272, 2, 2010.0]]. This is one of the main contributions of SQLova model over SQLNet[XLS17] to identify values for where clause along with the identification of generation of where columns, number of where conditions, where operators, where value start index and where value end index. Our contribution is to use it use SQLova with GPT2 to improve its performance, as GPT2 has proven useful for machine translation tasks because of its emulation to the idea of Left to Right Language Models.

4.2 Model Hyper-parameters

To improve the performance of existing and modifies baselines of IRNet and SQLova, the following hyper-parameters were considered:

- **Vector Size & Network width**: Vector Size is set by encoder_dim and hidden_size parameters which controls the size of context vector to be allowed to capture the relationships learned in these vectors and the size of the network respectively. **Embedding vector length** in IRNet is set by action_embed_size, field_embed_size and type_embed_size for SEMQL action(APPLY RULE, SELECT COLUMN, SELECT TABLE), field(column id), and type(entity type i.e. table, column or value) label’s vector size to be varied in the range of [64,128,768] where the maximum value of 768 is selected as it is equal to the smallest size of transformer model’s embedding size. For SQLova, hS is one contextual vector learned to predict the output for each of the 6 modules as mentioned in Subsection 3.3.2. In order to pass the inputs to the fully connected LSTM layers of both IRNet and SQLova models, we also increase the width of the network by increasing the number of neurons in hidden layers at encoder level, which is also an important parameter.

- **Activation Functions**: In the last decoder layers of IRNet and SQLova model, we experiment with different activation functions like: GELU, RELU, SELU, SiLu(Swish function in Pytorch) and Leaky RELU to see the impact in model performance.
• **Beam Size for decoding**: Beam width/beam size is another important parameter when decoding as mentioned in Chapter 2. 5 and 10 for IRNet and SQLova to improve the accuracy of corresponding decoders.

• **Learning Rate, weight decay and gradient clipping**: We used $10^{-2}$ and $10^{-3}$ with the weight decay of 0.5, 0.05 and 1 for training our models. With IRNet we used gradient clipping value of 2.

• **Dropout**: During test time, the model usually report better performance over the unseen data. We use 0.2 and 0.5 values for dropout.

• **Layer Normalization (LN)**: We employ layer normalization (LN) by computing the normalization statistics separately at each time step in the decoder layers of IRNet and SQLova to report the impact in performance of the model.

• **Batch Size and accumulate gradients**: When training neural networks, batch size is often an important parameter when RAM is limited and model generalization is needed to drive the learning of complex functions in right direction while minimizing loss. But since, transformer model when loaded on GPU runs into 'Out of memory' error if the batch size is more than 8 in case of GPT2, RoBERTa and 16 in case of BERT given the experimental setup in Section 4.3, we do not modify this parameter.

### 4.3 Experimental Environment

In this section, we mention the details about the experiment setup to run the experiments for this thesis. The following configurations is as follows:

**Machine Configuration**

- **Operating System**: Ubuntu 18.04.3 LTS
- **Processor**: 2.7 GHz Dual-Core Intel Core i5
- **Memory**: 8 GB 1867 MHz DDR3
- **Graphics**: NVIDIA Tesla V100-SXM2 32GB

**Programming Framework**

- **Programming Languages**: Python (Version 3.7.9)
- **Programming Tools**: PyCharm community edition (Version 2016.2.3), Jupyter Notebook (Version 4.3.1), Google Colab

**Libraries**:

1. Pytorch 1.6.0, torchvision 0.7.0
2. Cuda 10.2
3. Transformers(3.4.0)\textsuperscript{2},
4. pytorch-pretrained-bert
5. Scikit-learn (Version 0.22.2)\textsuperscript{3}
6. sqlite 3.33.0, SQLAlchemy 1.1.14
7. ujson 4.0.1
8. stanza 1.1.1
9. tensorboard-pytorch 0.7.1
10. stanford nltk library 3.4.0 or above
11. matplotlib (Version 3.2.1), numpy, pandas
12. pattern, babel 2.5.1
13. tqdm 4.31.1, records 0.5.2, defusedxml

\subsection{Summary}

In Section 4.1 we explain the detailed preprocessing steps for datasets Spider, Sparc and WikiSQL. We also explain the hyper-parameters to report their impact on model’s performance in Section 4.2. Finally, we explain experimental environment and libraries in Section 4.3 used to work during the thesis.

\textsuperscript{2}https://huggingface.co/transformers/, last accessed : 23 Feb 2021
\textsuperscript{3}https://scikit-learn.org/ : last accessed : 21 Feb 2021
5 Evaluation and Results

In this chapter, we present and discuss the evaluation results for the experiments we conducted to address the research questions. This chapter is organized as follows.

- We reinstate the research questions in Section 5.1
- In Section 5.2, we present out results and discuss the effect of tuning hyper-parameters of semantic parsing model combined with transformers to get contextualized embeddings for input to answer our first research question
- Section 5.3 is dedicated to comparing the performance of combining pre-trained transformer models with of grammar based models over cross-domain datasets to answer our second research question
- In Section 5.4, we compare the evaluation results for the experiments conducted by different transformer models and answer our third research question
- Section 5.5 provides an about the fundamental observations of this chapter.

5.1 Research Questions

In particular, we address the following research questions for comparative evaluation of grammar based and parsing based model over cross-domain datasets. We use the term dev set for dev set of datasets.

1. What tuning choices play a large role in the performance of a semantic parsing model on complex cross-domain datasets? (Model: SQLova)
2. To what extent can the performance of a grammar based model be improved and generalized on complex cross-domain datasets? (Model: IRNet)
3. To what extent does word contextualization using various transformer models improve the performance of parsing and grammar-based models? (Model: GPT2, BERT, RoBERTa)

5.2 RQ1: Hyper-parameter Tuning on Parsing-based model (SQLova)

In this section, we present the evaluation results by tuning hyper-parameters of the parsing-based NL2SQL model i.e. SQLova to answer our first research question. Unless otherwise mentioned, we use batch size of 16 and train over 30 epochs with a learning rate
of 0.0001 using Adam optimizer and weight decay of 0.05. All 6 modules of SQLova minimize cross entropy loss function, where the target refers to Execution accuracy evaluated over the number of tuples returned after executing the generated query. The performance results mentioned are over the test set of WikiSQL dataset. Our tests are however in no means exhaustive due to the time and scope of this research work as well as the vast possibilities of configurations for these experiments.

Note: For all the graphs in the following, x-axis represents number of epochs and y-axis represents loss/accuracy as mentioned in the graph. And all the best values are highlighted in the tables except in Section 5.5.

### 5.2.1 Vector Size & Network width

Our hypothesis is that vectors with larger sizes at least equal to the size of encoder vectors can capture more information especially for longer sequences. Increasing the width of the network refers to the number of units/nodes/neurons on the hidden layers of the network. And model with high enough capacity of neurons, layers when tuned correctly should be good enough to capture the input dependencies. So, along with changing the input vector sizes we also increase the dimensions of model’s encoder layers to be of the same size which leads to increase in the size of network.

For SQLova, hS i.e. hidden size is a contextual vector for encoder learned to predict the output for each of the 6 modules as mentioned in Section 3.5. Whereas all other vectors sizes for header information such as column name length and NL token length are kept consistent with the hidden size i.e. hS parameter. Unless otherwise mentioned, all the layers use same number of hidden dimension as the input vector, to form fully connected layers passed through TanH activation function, together forming a recurrent neural network.

### Results and Discussion

We conduct our first experiment by increasing the input vector and hidden layer size, hence increasing the number of hidden layers to give a bigger contextual vector. As can be seen from the graphs in Figure 5.2 and Figure 5.3, the training loss smoothly decreases for both 300 and 768 vector sized input to the models, and the dev loss abruptly increases. But for 300 dimension Figure 5.2, the model’s loss on dev set is less as compared to 768 dimension Figure 5.4.

<table>
<thead>
<tr>
<th>Input and Encoder Dimension</th>
<th>s-col</th>
<th>s-agg</th>
<th>w-num</th>
<th>w-col</th>
<th>w-op</th>
<th>w-val</th>
<th>Execution Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>97.3%</td>
<td>90.5%</td>
<td>98.7%</td>
<td>94.7%</td>
<td>97.5%</td>
<td>95.9%</td>
<td>87.2%</td>
</tr>
<tr>
<td>300</td>
<td>95.1%</td>
<td>89.8%</td>
<td>97.4%</td>
<td>94.0%</td>
<td>97.7%</td>
<td>95.9%</td>
<td><strong>87.6%</strong></td>
</tr>
<tr>
<td>768</td>
<td>94.8%</td>
<td>90.8%</td>
<td>98.5%</td>
<td>93.1%</td>
<td>97.8%</td>
<td>95.1%</td>
<td>85.2%</td>
</tr>
</tbody>
</table>

Table 5.1: Effect of hidden context vector-size tuning on different modules of SQLova Model.
We report the result for the corresponding vector size as mentioned in Table 5.1. Here, the execution accuracy of SQLova for increased vector size of 300 increases by +0.4%. But the accuracy for 768 vector size similar to the size of BERT encoder, leads to decrease in the performance of the model by -3%. SQLova+BERT is able to perform better with 300 dimension of vector sizes than 768 dimension. The decrease in performance for 768 dimensional vectors can happen due to the increase in the sparsity at the output produced by encoder layers. Since WikiSQL queries are shorter and the maximum sequence length is only 222, we can say that the context vector size of 300 is better suited to learn the conditional dependencies and capture the relationship between NL question and column names to make the relevant predictions for the modules of SQLova. Whereas 768 size is too large and contains more sparse entries in the weight vector of encoder and hence decoder as the largest sequence length in WikiSQL is only 222.

5.2.2 Activation Functions

Activation functions play an important role in squashing the bounds of outputs and hence are responsible to capture the complex relationships(linear/non-linear) between large-dimensional inputs. Rectified Linear Units also called as ReLu adds positive linearity to neural network and prevents the network from vanishing gradients problem, but it suffers from dying neurons problems for negative gradients. As large negative gradients are also helpful in learning the occurrence of errors, we use GELU activation function.
function which allows more negative gradients to pass through, as compared to leaky ReLu function. GELU is called as Gaussian Error Linear[HG20]. It uses a standard Gaussian cumulative distribution activation function introduced to penalize the error and keep the gradients within the fixed negative range and infinite positive range of a gradients. It aims to combine the dropout and activation function to regulate the neural network.

Along with right selection of activation function and dropouts, right kind of regularization is required to correctly tune the neural network. More such regularization techniques are weight initialization, output normalization(batch or layer norm), weight normalization etc. We do not change this parameter and instead introduce a new self-normalizing function called as SELU. SELU i.e. Scaled Exponential Linear Unit[KUMH17] is introduced to normalize neural networks by itself. SELU activation function is a self normalizing activation function which makes sure that neurons activation are pushed to zero mean and unit variance. It also prevents the network to suffer from exploding and vanishing gradients problem by keeping the bounds of output within 0 mean and unit variance. Thus we use these 3 activation function i.e. RELU, GELU and SELU.

All modules first have 2 LSTM fully connected recurrent layers followed by 2 linear fully connected layer and softmax function on the last linear output layer. All fully connected layer uses TanH non-linear activation function. We use other competing activation functions like RELU, GELU and SELU which can lead to improvement in SQLova’s existing performance. We also use early stopping over dev loss with patience value of 5 so that the model stops training if loss does not improves after 5 epochs. Since SQLova is a very slow model with the training time of approximately 62 minutes for 1 epoch, hence we used early stopping and trained the model for 30 epochs.

### Results and Discussion

<table>
<thead>
<tr>
<th>Activation Function</th>
<th>s-col</th>
<th>s-agg</th>
<th>w-num</th>
<th>w-col</th>
<th>w-op</th>
<th>w-val</th>
<th>Execution Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>TanH</td>
<td>95.1%</td>
<td>89.8%</td>
<td>98.4%</td>
<td>93.0%</td>
<td>97.7%</td>
<td>94.9%</td>
<td>83.6%</td>
</tr>
<tr>
<td>RELU</td>
<td>87.5%</td>
<td>86.9%</td>
<td>95.2%</td>
<td>68.4%</td>
<td>93.2%</td>
<td>81.4%</td>
<td>54.5%</td>
</tr>
<tr>
<td>GELU</td>
<td>86.6%</td>
<td>90.7%</td>
<td>94.5%</td>
<td>66.0%</td>
<td>93.7%</td>
<td>80.1%</td>
<td>53.8%</td>
</tr>
<tr>
<td>SELU</td>
<td>86.3%</td>
<td>90.9%</td>
<td>95.4%</td>
<td>63.8%</td>
<td>93.5%</td>
<td>78.4%</td>
<td>51.1%</td>
</tr>
</tbody>
</table>

Table 5.2: Effect of vector-size tuning on SQLova Model with transformers over WikiSQL dataset based on execution accuracy.

As can be seen in the graphs Figure 5.8, Figure 5.6 and Figure 5.12 since we used early stopping, GELU stopped training after 15 epochs, SELU stopped after 28 epochs and RELU stopped after 25 epochs as the dev loss did not improve and kept on increasing.

In Table 5.2, we see the result of 3 activation functions replaced by the default TanH function of SQLova model in both encoder and decoder layers computer. Here, we see that for column prediction for select and where module RELU, GELU and SELU function does not perform better than the actual TanH function. Also, only for the aggregation module, GELU and SELU functions perform better than TanH and RELU function.
RELU although still does perform as good as TanH but it manages to get good accuracy on operator and number of condition predictions for the where modules. Looking at the training loss convergence graph in Figure 5.5, the loss decreases abruptly. For GELU, the convergence for training loss can be seen in Figure 5.7, where the training loss increases initially but decreases drastically after the 6th epoch. The dev loss however kept on increasing until the training finally stops because of early stopping. GELU performs good for 3 modules aggregation prediction, number of conditions and operator prediction of where module. Hence, more parameter tuning for GELU in other modules might see improvement.

We think that for SELU to perform better adding more layers in encoder and all of the layers with SELU activation function will give better results in order to have self normalizing properties across all the layers of the model when the SQLova+BERT trained
for more iterations. As can be seen in Figure 5.11, loss reduction is very smooth during training implying that the gradients remained within a mean of 0 and variance of 1. Hence, as evident from loss on dev set of we do not face any vanishing gradients problem in the model using SELU activation function. But for SQLova, using existing TanH function is indeed a better choice.

5.2.3 Dropout

Dropout helps to prevent overfitting by randomly setting p percent of the node’s output to zero. Overfitting occurs when the model almost ‘memorizes’ the training data, thus yielding very less errors during the training and hence, failing to generalize over unseen dataset. The dropout probability ranges from 0.1 to 0.9 meaning randomly 10% or 90% of the node’s output are set to zero in every iteration of the layers of neural network, probability of 1 means no dropout is used. By using dropouts, we allow the model to randomly select some nodes and remove them (i.e. set output to 0) with all of its corresponding incoming and outgoing connections in the network that is linked to those particular nodes. This prevents the model to memorize the data in the model when the model is huge, which prevents it from assigning higher weights to some nodes. Thus, it takes away the reliance on some components and hence prevents overfitting. We set dropout probability value to 0.2 and 0.5 for our experiments. We use the same reasoning for our hypothesis, to prevent overfitting in our model by using 2 dropouts, small value of 0.2 and large value of 0.5. This causes an initial increase in loss for the higher dropout, which also takes more time for the model to converge but over time with more number of iterations the loss stabilizes. The model was only trained over 30 epochs for this case.

Results and Discussion

![Figure 5.11: Dropout of 0.5 : Loss on training set for SQLova model over wikisql dataset](image1)

![Figure 5.12: Dropout of 0.5 : Loss on dev set for SQLova model over wikisql dataset](image2)

We gather the results from this experimentation with introducing dropouts in the network and present them in Table 5.3. The possibilities are vast so we experiment with only a few dropout rates for this research work but this could definitely be expanded in further studies in future works. From the results in the table, we see that a dropout of 50% on all
layers of the network perform better on the dev set in comparison to 20% selected dropout rate and default 0.3 dropout rate over test set of WikiSQL dataset. The improvement in accuracy is by +0.2% for 0.5 dropout. Hence, our suggested hypothesis of improving the model performance with higher dropout is proven true.

<table>
<thead>
<tr>
<th>Dropout</th>
<th>Select Col</th>
<th>Agg Op</th>
<th>Where Cond</th>
<th>Where Col</th>
<th>Where Op</th>
<th>Where Val</th>
<th>Execution Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.3</td>
<td>97.3%</td>
<td>90.5%</td>
<td>98.7%</td>
<td>94.7%</td>
<td>97.5%</td>
<td>95.9%</td>
<td>87.2%</td>
</tr>
<tr>
<td>0.2</td>
<td>94.8%</td>
<td>89.9%</td>
<td>98.5%</td>
<td>94.1%</td>
<td>96.9%</td>
<td>95.2%</td>
<td>83.6%</td>
</tr>
<tr>
<td>0.5</td>
<td>95.9%</td>
<td>92.8%</td>
<td>98.5%</td>
<td>97.4%</td>
<td>98.0%</td>
<td>95.4%</td>
<td>87.4%</td>
</tr>
</tbody>
</table>

Table 5.3: Effect of tuning dropout on 6 modules of SQLova model over wikisql dataset

5.3 RQ2: Generalization on cross-domain dataset (Sparc) and Hyper-parameter Tuning on Grammar-based Model (IRNet)

Unless otherwise mentioned, we train over 100 epochs using Adam optimizer, early stopping and minimize Negative log-likelihood (NLL) loss. We report Exact match accuracy and sketch accuracy over SEMQL query label. The performance results mentioned are over the dev set of Spider and Sparc dataset, as test set is kept private by owners of the dataset. The models have to submitted along with our evaluation code and the inference over test set is performed by the Yale’s Lily team providing that the inference time of the model is less. Like other models reported in recent survey by POSTECH team and Google[KSHL20] that report performance on Spider’s validation/dev set, we also report the performance on the same. Also, since Sparc is meant for conversational agent but has lot of better revised queries than spider, we modify its objectives from multiturn conversation NL2SQL task to single turn NL2SQL task and report its performance on Exact Match accuracy which is our novel contribution to the thesis. We work with IRNet+BERT, IRNet+GPT2 and IRNet+RoBERTa models over Spider and Sparc dataset to report hyper-parameter optimization results. We also provide training graph over Sparc dataset in Appendices.

5.3.1 Vector Size & Network width

As discussed in Subsection 5.2.1, we follow the same hypothesis of capturing more information with bigger vector size. Although the larger size of the vector could lead to sparse values but we expect that we can also benefit by capturing more dependencies and avoid vanishing gradient problem as our input consists of larger sequences of Spider and Sparc. These datasets have larger query length as compared to other NL2SQL datasets such as WikiSQL.

In IRNet, encoder_dim and hidden_size controls the size of context vector to be allowed to capture the relationships learned in these vectors. In our experiments we refer to this as hidden layer dimension as IRNet’s encoder comes after the transformer.
model, hence is analogous to hidden layers of the model. And action_embed_size, field_embed_size and type_embed_size for SEMQL action (APPLY Rule, Select column, Select table), field(column id), and type(entity type i.e. table, column or value) label’s vector size is varied in the range of [64,128,768] where the maximum value of 768 is selected as it is equal to the dimensions of transformer’s smaller models used. We refer to this as input vector size in our experiments wit the exception that type_embed_size is kept at 32 for GPT2 and RoBERTa. We expect that increasing the units in hidden layers of encoder and increasing the size of input vector will positively impact the model output and vice-versa. We also show the corresponding training graphs for Sparc dataset in appendix.

Results and Discussion

![IRNet: Training Loss on input vector size and hidden layer dimensions](image)

**Figure 5.13:** IRNet with different transformers: Training loss on input vector size and hidden layer dimensions

The training loss for this experiment is shown in Figure 5.13, where we can see that IRNet+GPT2 with 64 dimension of input vector and 768 hidden vector size gives higher loss as compared to the other experiments, but with the increase in size of input vector 128, it gives the lowest training loss. Whereas, for both IRNet+BERT and IRNet+RoBERTa have similar decrease in training loss.

We show the results on dev set of Spider in Table 5.4, by increasing vector sizes for action type and column embedding and increasing the context vector of encoder of IRNet+GPT2, IRNet+BERT, IRNet+RoBERTa to be equal to the dimension of existing transformer models GPT2, BERT and RoBERTa and see that the performance of model does not improve over dev set. But with the graph for training loss, IRNet+GPT2 gives lowest loss during training, hence we think that with more tuning of IRNet+GPT2 w.r.t to other parameters and using larger GPT2 model, it might be able to perform well on dev set.
### Table 5.4: Effect of vector-size tuning on IRNet Model with transformers over Spider dataset based on Sketch Accuracy and Exact Match accuracy

<table>
<thead>
<tr>
<th>Input Vector</th>
<th>Hidden Layer</th>
<th>Model</th>
<th>Sketch Accuracy</th>
<th>Exact Match Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>300</td>
<td>IRNet+BERT</td>
<td>89.12%</td>
<td>61.9%</td>
</tr>
<tr>
<td>64</td>
<td>768</td>
<td>IRNet+BERT</td>
<td>82.9%</td>
<td>55.45%</td>
</tr>
<tr>
<td>128</td>
<td>768</td>
<td>IRNet+BERT</td>
<td>82.39%</td>
<td>56.23%</td>
</tr>
<tr>
<td>64</td>
<td>768</td>
<td>IRNet+GPT2</td>
<td>78.79%</td>
<td>53.31%</td>
</tr>
<tr>
<td>128</td>
<td>768</td>
<td>IRNet+GPT2</td>
<td>78.41%</td>
<td>52.92%</td>
</tr>
<tr>
<td>64</td>
<td>768</td>
<td>IRNet+RoBERTa</td>
<td>68.58%</td>
<td>29.48%</td>
</tr>
<tr>
<td>128</td>
<td>768</td>
<td>IRNet+RoBERTa</td>
<td>68.87%</td>
<td>34.44%</td>
</tr>
</tbody>
</table>

### Table 5.5: Generalization over Sparc dataset and effect of vector-size tuning on IRNet Model with different transformers based on Sketch Accuracy and Exact Match accuracy

<table>
<thead>
<tr>
<th>Input Vector</th>
<th>Hidden Layer</th>
<th>Model</th>
<th>Sketch Accuracy</th>
<th>Exact Match Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>768</td>
<td>IRNet+BERT</td>
<td>51.89%</td>
<td>35.88%</td>
</tr>
<tr>
<td>128</td>
<td>768</td>
<td>IRNet+BERT</td>
<td>51.80%</td>
<td>35.79%</td>
</tr>
<tr>
<td>64</td>
<td>768</td>
<td>IRNet+GPT2</td>
<td>61.45%</td>
<td>43.01%</td>
</tr>
<tr>
<td>128</td>
<td>768</td>
<td>IRNet+GPT2</td>
<td>60.94%</td>
<td>40.32%</td>
</tr>
<tr>
<td>64</td>
<td>768</td>
<td>IRNet+RoBERTa</td>
<td>54.57%</td>
<td>28.75%</td>
</tr>
<tr>
<td>128</td>
<td>768</td>
<td>IRNet+RoBERTa</td>
<td>56.41%</td>
<td>32.27%</td>
</tr>
</tbody>
</table>

We also perform similar experiments over Sparc dataset which has even more complex queries than Spider dataset, it gives better sketch accuracy and exact match accuracy for input vector of 128 dimensions and 768 hidden layer dimensions with IRNet+GPT2 transformer as shown in Table 5.5. In case of IRNet+BERT, vector size of 64 and hidden layer size of 768 gives better results. And with IRNet+RoBERTa and input vector size of 128 gives better results as compared to input vector size of 64 with better sketch accuracy for both the experiments as compared to that of BERT.

#### 5.3.2 Activation Functions

Activation functions are important parameters to add non-linearity to neural networks in order to learn more complex functions. Our hypothesis is that, for longer sequences in Spider and Sparc, the model might be suffering lower gradients leading to dying neurons while training the model with RELU as its activation function. Thus, we use Leaky ReLu activation function, which adds linearity for smaller negative gradients up to some limit by replacing all the negative values with pre-selected smaller negative value and zeroing out the negative gradients after it. This helps to fix the dying neurons problem up to some extent by allowing some smaller negative gradients. We try better activation functions than RELU like Leaky ReLu, GELU[HG20] and SELU[KUMH17] hoping that they might lead to better results.
These functions are added at the encoding layer of IRNet model after getting the contextualized embedding output from transformer model which takes in the attention vector generated over columns, column types and NL input question. These functions are also applied at final layer of encoder generating table encoding on the basis of which the output is generated by the IRNet decoder layer. We think that these functions will lead to better prediction and hence generation of next node and node values at the decoder layer.

Results and Discussion

As shown in Figure 5.14, a training loss decreases smoothly for 3 functions used in IRNet+BERT model, except for GELU function there is a slight increase between from 70 to 78th epoch, followed by a sudden decrease until the 80th epoch and then decreases smoothly until 95th epoch like SELU and Leaky RELU. After the 85th epoch, GELU returns “nan” or null value training loss hence there is no following plot in the graph for it. Training loss for SELU is the lowest for IRNet+BERT model.

In case of IRNet+GPT2 as shown in Figure 5.15, training loss decreases smoothly for both SELU and GELU activation functions, with SELU giving the lowest training loss. Except Leaky ReLu, for which the loss converges around the value of 1.0 and stops decreasing after 40th epoch.

IRNet+RoBERTA gives similar decrease as that of IRNet+BERT in training losses and converges for the three activation function as shown in Figure 5.16. The training loss convergence value is still higher than IRNet+BERT.

We show the results over dev set of Spider dataset in Table 5.6 on which IRNet+BERT gives the best result for GELU activation with an improvement of sketch accuracy of
Figure 5.15: IRNet+GPT2: Training loss on activation functions over Spider dataset

Figure 5.16: IRNet+RoBERTa: Training loss on activation functions over Spider dataset

+3.2% and Exact match accuracy of +1.55%. IRNet+BERT also performs better with Leaky ReLu by an improvement of +1.33%.

In case of IRNet+GPT2 model, Sparc dataset gives better exact match accuracy of 37.89% and 38.39% with Leaky ReLu and SELU respectively as compared to all the other hyper-parameter optimization done in this thesis. IRNet+RoBERTa model does not give better results for any of the activation function as compared to that with IRNet+BERT or
Table 5.6: Effect of activation function tuning on IRNet Model with transformers over Spider dataset based on Sketch Accuracy and Exact match accuracy.

<table>
<thead>
<tr>
<th>Activation function</th>
<th>Model</th>
<th>Sketch Accuracy</th>
<th>Exact Match Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leaky ReLu</td>
<td>IRNet+BERT</td>
<td>83.46%</td>
<td>63.23%</td>
</tr>
<tr>
<td>Leaky ReLu</td>
<td>IRNet+GPT2</td>
<td>80.05%</td>
<td>55.56%</td>
</tr>
<tr>
<td>Leaky ReLu</td>
<td>IRNet+RoBERTa</td>
<td>68.87%</td>
<td>34.44%</td>
</tr>
<tr>
<td>GELU</td>
<td>IRNet+BERT</td>
<td>84.24%</td>
<td>63.45%</td>
</tr>
<tr>
<td>GELU</td>
<td>IRNet+GPT2</td>
<td>78.99%</td>
<td>59.27%</td>
</tr>
<tr>
<td>GELU</td>
<td>IRNet+RoBERTa</td>
<td>61.95%</td>
<td>31.52%</td>
</tr>
<tr>
<td>SELU</td>
<td>IRNet+BERT</td>
<td>75.19%</td>
<td>50.00%</td>
</tr>
<tr>
<td>SELU</td>
<td>IRNet+GPT2</td>
<td>80.06%</td>
<td>53.89%</td>
</tr>
<tr>
<td>SELU</td>
<td>IRNet+RoBERTa</td>
<td>73.06%</td>
<td>35.70%</td>
</tr>
</tbody>
</table>

IRNet+GPT2 except with SELU it performs slightly better.

Table 5.7: Generalization and performance tuning using different activation function on IRNet Model with transformers over Sparc dataset based on Sketch Accuracy and Exact match accuracy.

<table>
<thead>
<tr>
<th>Activation function</th>
<th>Model</th>
<th>Sketch Accuracy</th>
<th>Exact Match Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leaky ReLu</td>
<td>IRNet+BERT</td>
<td>50.55%</td>
<td>31.94%</td>
</tr>
<tr>
<td>Leaky ReLu</td>
<td>IRNet+GPT2</td>
<td>62.20%</td>
<td>37.89%</td>
</tr>
<tr>
<td>Leaky ReLu</td>
<td>IRNet+RoBERTa</td>
<td>58.92%</td>
<td>32.27%</td>
</tr>
<tr>
<td>GELU</td>
<td>IRNet+BERT</td>
<td>51.49%</td>
<td>30.43%</td>
</tr>
<tr>
<td>GELU</td>
<td>IRNet+GPT2</td>
<td>55.83%</td>
<td>35.74%</td>
</tr>
<tr>
<td>GELU</td>
<td>IRNet+RoBERTa</td>
<td>55.75%</td>
<td>25.73%</td>
</tr>
<tr>
<td>SELU</td>
<td>IRNet+BERT</td>
<td>58.26%</td>
<td>38.30%</td>
</tr>
<tr>
<td>SELU</td>
<td>IRNet+GPT2</td>
<td>58.68%</td>
<td>38.39%</td>
</tr>
<tr>
<td>SELU</td>
<td>IRNet+RoBERTa</td>
<td>55.82%</td>
<td>29.93%</td>
</tr>
</tbody>
</table>

We also report the performance results over dev set of Sparc dataset in Table 5.7, and with SELU activation function, IRNet+BERT and IRNet+GPT2 both gives highest exact match accuracy of around 38.3%, with IRNet+GPT2 performing better than IRNet+BERT. GELU does not give better results over Sparc dev set, but Leaky ReLu is able to perform relatively better with IRNet+GPT2. Hence our hypothesis is true that GELU, Leaky ReLu and SELU are proven to be better than default RELU function used by IRNet originally. And they given even better performance when used with IRNet+GPT2 model.

We conclude that, using GELU and Leaky ReLu for IRNet+BERT and IRNet+GPT2 works better, as both the transformers use GELU activation function and the bounds of gradients remained consistent both in the positive and negative direction, leading to better performance. SELU activation function performs better with IRNet+GPT2 model for Sparc dataset and one reason for this can be that the gradients remain bounded for longer and more sequences of Sparc and help the model to converge better. For this, we give more
Right values of learning rate can steer the convergence of model in right direction and and decaying it in multiple steps can speed up the training of the model. Since, neural networks are very complicated function of weights it is very difficult to say when we are near the global minima or local minima. IRNet uses the famous Adam\[KB17\] optimizer algorithm. Instead of experiment with different optimization algorithms, We explore few combinations of learning rate with the decay rate for all 3 models IRNet+BERT, IRNet+GPT2 and IRNet+RoBERTa in our experiments, since the scope of selecting the right algorithms and right values is very vast. We continue with Adam as the base optimizer and select 2 learning rates 0.001 and 0.0001 and 3 decay in learning rate of 0.05 and 0.5.

Results and Discussion

Figure 5.17: IRNet+GPT2: Training loss graph for learning rate, weight decay and gradient clipping

As we can see the comparison of training loss convergenceFigure 5.17 on different learning rate on IRNet+GPT2 that learning rate of 0.0001 converges to 2.0 for both weight decay values of 0.5 and 0.05 and gradient clipping value of 2.0. For gradient clip value of 5.0, the model returned "nan" loss after 40 epochs and hence is not shown here. Similarly in case of IRNet+RoBERTa and experiments with learning rate during training is shown in Figure 5.19, the loss decreases smoothly and manages to converge below 0.5 and
giving the lowest loss for learning rate of 0.0001, decay of 0.5 and gradient clipping of 5.0.

And as seen in the Table 5.8, for learning rate of 0.001 model performs better in terms of sketch accuracy dev set if the gradients higher than 2.0 are clipped but the exact match accuracy remains lower. This means that classification of the next node over operation, column or table must be improved but the value prediction of still lower than the original performance with learning rate of 0.001, decay of 0.5 and gradient clipping of 5.0 (shown in the first row of the table).

Finally as shown in Table 5.8, because of overfitting only IRNet+BERT with default learning rate of 0.001 and our suggested decay rate of 0.05 gives 61.84% accuracy and the model is able to reach near the original result of 61.9% with a slight decrease of -0.06% on spider dataset’s dev set. For learning rate of 0.001 the training loss convergence value is higher for IRNet+GPT2 and IRNet+RoBERTa for similar hyper-parameters and hence both report low accuracy on dev set.

![Figure 5.18: IRNet+BERT: Training Loss on learning vector, weight decay and clip gradients](image)

In case of IRNet+BERT, the training loss converges goes as low as 0.1 and stays still lower than 0.25.

We conclude that for Spider dataset, learning rate of 0.001 and decay of 0.5 with the clip gradients value of 5.0 originally works better. Whereas for other values, models does not report improvement because with higher learning rate and higher value for clipping gradients, the model is able to reach near to the global minima.

Performance over the new Sparc dataset’s dev set is outlined in Table 5.9. IRNet+GPT2 reported highest accuracy of 51.56% for learning rate of 0.0001, decay of 0.5 and decay of 2.0. With a higher learning rate of 0.001, IRNet+GPT2 reports 43.00% of dev accuracy. IRNet+BERT model manages to get 41.74% of accuracy whereas IRNet+RoBERTa
Figure 5.19: IRNet+RoBERTa: Training loss graph for learning rate, weight decay and gradient clipping

Table 5.8: Performance tuning using Learning Rate, Decay rate and Clip gradient on IRNet Model with different transformers over Spider’s dev dataset based on Sketch Accuracy and Exact match accuracy.

gives only 32.28% of accuracy. To conclude, IRNet+BERT and IRNet+GPT2 models report relative performance improvement as compared to with other hyper-parameters discussed in this thesis. We recommend using lower learning rate of 0.0001 and smaller clip gradient value of 2.0(or even lower gradient values) to train the models because
<table>
<thead>
<tr>
<th>Learning Rate</th>
<th>Decay</th>
<th>Clip Gradients</th>
<th>Model</th>
<th>Sketch Accuracy</th>
<th>Exact Match Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.001</td>
<td>0.5</td>
<td>5.0</td>
<td>IRNet+BERT</td>
<td>51.89%</td>
<td>35.88%</td>
</tr>
<tr>
<td>0.001</td>
<td>0.05</td>
<td>5.0</td>
<td>IRNet+BERT</td>
<td>60.94%</td>
<td>33.86%</td>
</tr>
<tr>
<td>0.0001</td>
<td>0.05</td>
<td>2.0</td>
<td>IRNet+BERT</td>
<td>60.35%</td>
<td>41.74%</td>
</tr>
<tr>
<td>0.0001</td>
<td>0.5</td>
<td>2.0</td>
<td>IRNet+BERT</td>
<td>50.55%</td>
<td>31.94%</td>
</tr>
<tr>
<td>0.001</td>
<td>0.05</td>
<td>2.0</td>
<td>IRNet+GPT2</td>
<td>58.93%</td>
<td>36.38%</td>
</tr>
<tr>
<td>0.001</td>
<td>0.5</td>
<td>2.0</td>
<td>IRNet+GPT2</td>
<td>60.34%</td>
<td>43.00%</td>
</tr>
<tr>
<td>0.0001</td>
<td>0.05</td>
<td>2.0</td>
<td>IRNet+GPT2</td>
<td>41.79%</td>
<td>38.98%</td>
</tr>
<tr>
<td>0.0001</td>
<td>0.5</td>
<td>2.0</td>
<td>IRNet+GPT2</td>
<td>78.06%</td>
<td>51.56%</td>
</tr>
<tr>
<td>0.001</td>
<td>0.05</td>
<td>5.0</td>
<td>IRNet+RoBERTa</td>
<td>68.29%</td>
<td>28.31%</td>
</tr>
<tr>
<td>0.001</td>
<td>0.5</td>
<td>5.0</td>
<td>IRNet+RoBERTa</td>
<td>68.58%</td>
<td>29.45%</td>
</tr>
<tr>
<td>0.0001</td>
<td>0.05</td>
<td>5.0</td>
<td>IRNet+RoBERTa</td>
<td>58.93%</td>
<td>32.28%</td>
</tr>
<tr>
<td>0.0001</td>
<td>0.5</td>
<td>5.0</td>
<td>IRNet+RoBERTa</td>
<td>55.83%</td>
<td>29.95%</td>
</tr>
</tbody>
</table>

Table 5.9: Performance tuning using Learning Rate, Decay rate and Clip gradient on IRNet model with different transformers generalized over Sparc’s dev dataset based on Sketch Accuracy and Exact match accuracy.

the models needs to take smaller steps to converge while it learns over the current and previous context vector for Sparc dataset which consists of longer and more related queries.

5.3.4 Dropout

We follow the hypothesis mentioned in Subsection 5.2.3 and select the same 2 dropout values of 0.2 and 0.5 to evaluate the performance on the three version of IRNet models. And we expect an improvement in performance over the dev set of Spider dataset. We do not test this parameter over Sparc dataset because of runtime constraint. As for Sparc dataset, IRNet with all the three transformers takes more time because of more number of queries and additional complexity of nested(longer) queries.

Results and Discussion

We first show the training loss convergence graphs in Figure 5.20, the training loss decreases for all dropouts except for 0.5 when IRNet+GPT2.

For 0.5 dropout, the exact match accuracy of IRNet+BERT drops by -14.25%. This is the lowest dev accuracy recorded for spider dataset w.r.t to the other parameters. For dropout of 0.5, IRNet+GPT2 and IRNet+RoBERTa gives good sketch accuracy and Exact Match accuracy as compared to other hyper-parameters. And the final performance over dropouts can be seen in Table 5.10, in IRNET+BERT gives higher accuracy with 0.2 dropout gives an improvement of +1.39% in exact match accuracy of as compared to the default dropout(0.3) performance.
Figure 5.20: IRNet with different transformers: Training loss graph for dropouts

Table 5.10: Effect of dropout tuning on IRNet with transformers over Spider dataset based on Sketch Accuracy and Exact match accuracy for 100 epochs.

We conclude that, since we are not fine tuning the transformer models, lower dropout of 0.2 works better for IRNet+BERT model because only a smaller % of neurons of encoders and decoders are dropped randomly. This forces IRNet+BERT model to rely more on embeddings generated by BERT. Thus it needs to adapt the model by only using a smaller dropout. We also see relative improvement in performance of IRNet+GPT2 and IRNet+RoBERTa for higher dropout of 0.5, we can infer from this that models trained on other hyper-parameters were suffering from overfitting and hence a higher dropout gives better performance when training on Sparc dataset.

5.3.5 Layer Normalization

In deep learning, batch normalization(BN) is performed to reduce the training time and stabilize training by controlling the mean and variance of layers across different batches. However, for RNNs and transformers which deals with sequences in time, for
each mini-batch batch normalization has been discouraged. Layer-normalization is proposed to improve convergence of RNNs[BKH16]. Our hypothesis is derived from this and we propose that, stabilization of decrease in loss during learning can lead to better generalization of the model. We use layer normalization so that mean and standard-deviation are calculated separately over encoder’s output at the last layer which contains information learned over NL question, column name, column type and table embedding.

Results and Discussion

Layer normalization still does not improve on training loss and hence gives lower dev accuracy of 51.55% for IRNet+GPT2, even though GPT2 itself uses GELU activation function. And similarly for IRNet+Roberta it gives 38.94% accuracy which is the highest dev accuracy recorded with IRNet+Roberta over all the other hyper-parameters. As shown in Figure 5.21 with layer normalization, we get smooth decrease in training loss and end up getting smallest loss value for IRNet+BERT and the highest exact match accuracy of 63.55% as to that of the IRNet default accuracy of 61.9% i.e and improvement of +1.65%.

As shown in Table 5.11, applying Layer normalization improves the performance of IRNet+BERT and IRNet+RoBERTa transformer model. For IRNet+GPT2, layer normalization gives lowest sketch accuracy and exact match accuracy as compared to the performance of all the other hyper-parameters discussed till now in the above sections. We conclude that layer normalization, leads to smooth convergence of loss because of which within the same number of iterations we can get even lower loss as compared to other hyper-parameters. We conclude by saying that layer normalization when combined with other hyper-parameters will lead to better convergence.
<table>
<thead>
<tr>
<th>Model</th>
<th>Sketch Accuracy</th>
<th>Exact Match Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRNet+ BERT</td>
<td>84.24%</td>
<td>63.55%</td>
</tr>
<tr>
<td>IRNet+GPT2</td>
<td>78.31%</td>
<td>51.55%</td>
</tr>
<tr>
<td>IRNet+RoBERTa</td>
<td>68.77%</td>
<td>38.94%</td>
</tr>
</tbody>
</table>

Table 5.11: Effect of adding Layer-Normalization on IRNet model for different transformers over Spider dataset.

5.4 RQ3: Word Contextualization using Transformer Models

5.4.1 SQLova with transformer model: SQLova+GPT2

We replaced BERT encoder transformer in SQLova+GPT2 decoder transformer to provide justification with embeddings tokenized using BPE algorithms and derived from left to right transformers like GPT2 works better for parser based models. For this we provide 4 images with training loss and accuracy graphs for SQLova+GPT2 model over training and dev set of WikiSQL dataset. As can be seen in Figure 5.22, the overall training loss for all the 6 modules of SQLova+GPT2 decreases smoothly after 25 epochs and converges to loss of approximately 0.78 as shown in Figure 5.22. The dev loss in Figure 5.23 however does not converges till 50 epochs and kept on increasing which shows that the model IRNet+GPT2 suffers from overfitting. It gives execution accuracy of 66.2% and logical form accuracy of at most 60% averaged over the performance of 6 modules as shown in Figure 5.24 and Figure 5.25, in the figures, x axis represents epochs, y axis represents accuracy.

The overall performance of SQLova+GPT2 over test set of wikisql is shown in Table 5.12. It shows that SQLova performs better for modules other than prediction of column and value for where column. The execution accuracy reported by SQLova+GPT2 is 70.72% whereas with SQLova+BERT the execution accuracy is 86.2%. This makes sense as the where module of SQLova+GPT2 does not perform better, and one of the reasons can be that since the prediction of column itself is wrong, the module fails to find the
correct start index and end index in the schema information dropping the overall logical accuracy of SQLova+GPT2 to 66.17 which for SQLova+BERT is 80.7%. We can infer that SQLova+GPT2 does not perform good enough as that of BERT. And since training for SQLova+GPT2 is very high, it is better to optimize the existing model with BERT and our suggested hyper-parameters of selecting higher encoder dimensions and input vector size in Subsection 5.2.1 and higher dropout value of 0.5 mentioned in Subsection 5.2.3 will give better performance.

### 5.4.2 IRNet with transformer model

We trained IRNet+GPT2 and IRNet+RoBERTa transformer models on both Spider and the new Sparc dataset. We also performed hyper-parameter optimization over these both of these model and to answer the third research question for grammar-based model, we take into consider the best performing model.

For Spider dataset, we got the highest sketch accuracy of 79% and exact match accuracy of 59.27% as shown in Table 5.6. We got 43.01% highest exact match accuracy and 61.45% of sketch accuracy when trained with IRNet+GPT2 on dev set of Sparc dataset as shown in Table 5.5. Even though the accuracy of Sparc is lower than 50% but the model still performs better than the original Question match accuracy(equivalent of Exact match accuracy without values) of 18.5% over dev set recorded by the Yale Lily team which published the dataset[YZY+19]. To conclude, for Sparc IRNet+GPT2 gives better results as compared to IRNet+BERT and IRNet+RoBERTa models. In case of IRNet+GPT2, the overall performance is slightly lower as compared to IRNet+BERT in case of Spider,
but better in case of Sparc dataset. IRNet+RoBERTa however is unable to reach above 50% for both Spider and Sparc datasets.

**Figure 5.26:** SQL Query Translation: Comparison of IRNet with different transformers on Spider’s dev set (1034 NL questions)

We can also see this comparison in terms of number of correct and incorrect SQL predictions in Figure 5.26 and Figure 5.27. IRNet with BERT predicts good 654, GPT2 gives 575 and RoBERTa only gives 358 correct SQL queries for spider dataset. In case of Sparc dataset, IRNet+GPT2 predicts 571, IRNet+BERT predicts 491 and IRNet+RoBERTa predicts only 354 correct SQL queries. One reason for IRNet+GPT2’s better performance can be the set of coherent and refined set of questions and SQL queries that the Sparc provides. As Sparc’s questions are more coherent referring to tables of similar databases across multiple queries, it becomes easy for the model to perform better over already seen tables of database as tables hardly change over different turns and have to focus more on columns array and relationship obtained from the primary key and foreign key pairs from the schema information.

### 5.5 Summary

In this chapter, we first reiterate our research questions. We first explain the hyper-parameter applied to SQLova+Bert model and the present the results in terms of test accuracy. We also replace Bert based encoder with GPT2 and provide our results in Table 5.12. We find that SQlova+GPT2 is even more slow and suffers with a lot of overfitting.
We summarize our results in Table 5.13. We show that for bigger input vector size and encoder dimension and higher dropout SQLova+Bert is able to perform even better with an overall improvement in test accuracy of 1.65%. We explain that the improvement is because encoder is having enough neurons in its first hidden layers which can take in the maximum sequence length of dataset and can accommodate the output of BERT model. This helps SQLova+BERT model to get embeddings from BERT and learn complex conditional probabilities over these embeddings to learn more meaningful representation to pass onto the decoder. And with larger dropout the model is able to prevent overfitting.

In the second research question, we cover even more hyper-parameters for modified grammar based models i.e. IRNet+BERT, IRNet+GPT2 and IRNet+RoBERTa. We also report our new generalization results in Table 5.14 over Sparc dataset over the models and explain that IRNet+GPT2 model gives better results as compared to IRNet+BERT and IRNet+RoBERTa. We conclude that since Sparc consists of more related sequences
across batches it led to better performance. And in case of spider dataset, layer normalization, GELU activation function, lower dropout and smaller decay in learning rate gave better results.

<table>
<thead>
<tr>
<th>Hyper-parameter</th>
<th>Value</th>
<th>Model</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Size</td>
<td>64,768</td>
<td>IRNet+GPT2</td>
<td>43.01%</td>
</tr>
<tr>
<td>Vector Size</td>
<td>128,768</td>
<td>IRNet+GPT2</td>
<td>40.32%</td>
</tr>
<tr>
<td>Activation Function</td>
<td>SELU</td>
<td>IRNet+GPT2</td>
<td>38.39%</td>
</tr>
<tr>
<td>Learning-rate, decay, clip-gradient</td>
<td>0.001,0.5,2.0</td>
<td>IRNet+GPT2</td>
<td>43.00%</td>
</tr>
</tbody>
</table>

Table 5.14: Summary of generalized Results on Sparc dataset

To answer to our third research question, we present the results of SQLova+GPT2 model and show that the original SQLova+BERT model by the authors works better. And the existing SQLova+BERT model will give better results when trained with the suggested hyper-parameters such as vector size for input and encoder and increasing dropout value. We also mention that encoder utilizing embeddings of transformers should have number of neurons between both the length of the sequence allowed and the transformer vector size. We discourage the use of smaller hidden size in the encoder layer attached to large transformer models which was the case with original SQLova model.

We also compare the best results obtained over the IRNet+BERT, IRNet+GPT2 and IRNet+RoBERTa models and report that IRNet+GPT2 model is comparatively better for Sparc dataset with hyper-parameter like activation functions, layer normalization, dropout and encoder vector size. And IRNet+BERT with our mentioned set of hyper-parameters such as activation functions, dropout and layer normalization gives better results for Spider dataset. Finally, we conclude after studying the comparative performance of the models that layer normalization and activation functions like GELU should be used for models utilizing transfer learning with transformers, as it leads to stabilized decreases in loss as the gradients do not vary across different layers. Also, SELU is a good activation function for training the model when used without layer normalization and across all the layers of the model. Hence we discourage its usage especially in the context of trained transformers such as BERT, GPT2 and RoBERTa which already uses GELU activation function.
6 Related Work

In this chapter, we briefly compare our results against those of related work in two categories: transformers over question-answering task, which bears similarity to the studied NL2SQL task; and transformer models for NL2SQL.

6.1 Transformers comparison

“Attention is all you need” [VSP+17a] paper which proposed the transformer architecture, immediately became a popular technique for natural language processing (NLP) tasks. It led BERT [DCLT18] by Google, which used the transformer architecture with unsupervised pre-training on huge amount of data. It achieved state-of-the-art results in multiple NLP tasks. Since then, many new BERT-based approaches have been proposed to assess BERT power on different tasks. Succeeding papers attempted to fine-tune BERT on other NLP tasks such as text classification [SQXH19], extractive summarization [Liu19], or ranking [QXLL19] and the other branches of BERT family, in particular, DistillBERT [SDCW19], and RoBERTa (which we employed in this thesis) [LOG+19b] which proved to be beneficial for general NLP tasks. BERT’s lack of justification for processing masked tokens independent of each other led to release of more state-of-art transformer approaches such as XLNet [YDY+19], GPT2 (which we employed in this thesis) models [RWC+19b], T5 [RSR+20] etc.

The Stanford Question Answering Dataset (SQuAD) provides a paragraph of context and a question. The task is to answer the question by extracting the relevant span from the context [RJL18]. Comparing the performance of the transformer models on question answering task on SQuAD 2.0 dataset in Table 6.1. As shown in Table 6.1, BERT and RoBERTa report better results than GPT2. RoBERTa which is an revised version of BERT trained for longer time trained on more general dataset performs better than BERT. As a point of comparison in our work we have found BERT and GPT2 to consistently perform better than RoBERTa for the task and hyper-parameters under study. However we should note that our results are not using the larger versions of models, and hence this might a reason, apart from the task change, behind our different observations.

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Benchmark</th>
<th>Model</th>
<th>Exact Match Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Question Answering</td>
<td>SQuAD 2.0</td>
<td>BERT</td>
<td>79.0%</td>
</tr>
<tr>
<td>Question Answering</td>
<td>SQuAD 2.0</td>
<td>GPT2</td>
<td>63.1%</td>
</tr>
<tr>
<td>Question Answering</td>
<td>SQuAD 2.0</td>
<td>RoBERTa</td>
<td>86.82%</td>
</tr>
</tbody>
</table>

Table 6.1: Transformer comparison
6.2 Related Work: NL2SQL Approaches

NL2SQL/Text2SQL (NLP term) has been an active area of research just after the evolution of databases since 1970 [OWE00, BJK+12, WKW72, SFS+16, LYJ07, BJK+12, BKK05, DTB08, ZZY+09, KBF07, BH15, ZCZ+17, DTB08, KBZ06, LJ14, WP82, ART93, PEK03, Hal06]. But most of the work focused on single domain i.e. working on one database at a time. To implement Natural Language Interfaces (NLI) for the database systems, most of the works was related to keyword-based methods, parsing based, grammar based methods, rule based methods either individually or combined with each other [LYJ07, LJ14] targeting single database at a time. The emergence of deep learning field (i.e. using deeper neural network models) and especially its impact in the NLP community after the release of RNNs and using them in and Seq2Seq architectures (encoder-decoder), has propitiated its uses in NL2SQL. Using deep neural networks to get better convergence required larger dataset containing pairs of question and answer pairs, as compared to that of the older datasets [DAD94, NYD17, CFDR18].

NL2SQL task is almost similar to question answering task like SQuAD [RZLL16] as the context from which answer should be generated is provided by schema information, as discussed in some work Section 6.1. The larger datasets intended specifically to benefit from deep neural networks are WikiSQL [ZXS17], Spider [YZY+19] and Sparc [YZY+19] datasets which we discussed in this thesis. The open source release of these datasets is one of the contributing factors to the progress in research. We use all the three datasets designed to solve NL2SQL problem using Seq2Seq architecture.

WikiSQL dataset is the largest single-domain dataset with around 80,560 queries, but has oversimplified queries and the dataset itself is too noisy because it is built from the information scraped over Wikipedia pages. It consists of SELECT statements querying information over single tables, with conditions in where clause and aggregation operation. Models working on WikiSQL includes using the execution result as the feedback in reinforcement learning loop [ZXS17], using a sketch based encoder [XLS17] i.e. first generating a sketch then fill missing information using the decoder [DL18], adding column type information with the schema information [YLZ+18], and execution-guided decoding [WHP+18]. Another method of using generative model to improve replicate content sql keywords, column names or cell information and improving where clause to understand the column-cell relation [STD+18]. We adapted the work by SQLova model which is using BERT to represent queries and column names [HYPS19]. This state-of-the-art model has achieved 86% accuracy on WikiSQL.

Spider dataset, however has lesser queries of around 10000 queries which is less than WikiSQL but the queries are distributed over 873 tables and 200 databases. It is more complex and has join queries, nested queries and queries tables from multiple databases. The state-of-the-art model is IRNet plus BERT [GZG+19] achieving 64% accuracy, other works include syntax tree [YYY+18], EditSQL [ZYE+19], gated graph neural network [BGB19]. The most recent model RATSQL [WSL+20] which uses relation aware self-attention mechanism similar to IRNet, performs schema-linking, encoding and feature representation uses BERT to enhance table semantic parsing [WSL+20] over tabular data and textual data (NL question). It gave better performance which was adapted to build Grappa. Grappa is built over RATSQl and performs data augmentation using synchronous CFG (SCFG) rules by combining WikiTable dataset with Spider to build more related examples. It
then uses RoBERTa-large model over the SCFG grammar enhanced semantic pairs and without using any manual labels provided as part of Spider dataset, it outperforms other models as of today with the highest dev accuracy of 75.5% and test accuracy of 70.5%. Clearly, the Seq2Seq architecture in deep learning when adapted with attention based transformer models[DCLT19, RWC+19a, LOG+19a] gives better results than any other traditional models used by NLIDB community in the past.

After WikiSQL and Spider came out and also because of transfer learning inspired from transformer models, the topic of NL2SQL drew even more attention. This led to the release of Sparc multi-turn dataset built specifically for interactive systems like chat-based systems. As Sparc data is meant for conversational interfaces, it has more complex queries collected over multiple turns. NL2SQL models like SyntaxSQL[YYY+18] uses syntax tree and IGSQL uses schema interaction graph encoder and weighs the schema information[CW20]. To make prediction on SQL tokens GAZP[ZLWZ21] performs data augmentation and enhance the columns and tables with their types[ZYE+19], gets the related table columns across different tables and then performs forward semantic parsing using BERT on top of BiLSTM encoder-decoder model. The highest recently recorded exact match accuracy aka Question accuracy for Sparc dataset is 55.8% over dev set[HGR+21] published on 5 Jan 2021. The recently published R2SQL[HGR+21] builds dynamic graph framework coupled with memory decay mechanism outperforms and reports 55.8% accuracy giving state-of-the-art results as of date.
7 Conclusion and Future Work

In this chapter, we summarize our work, and present the conclusions obtained from this thesis work. We also propose some future practices which may extend this study. This chapter is structured as follows:

- In Section 7.1, we present the important conclusions derived from our thesis work.
- In Section 7.2, we discuss some suggestions for future work.

7.1 Conclusion

Addressing our Research Question 1, we experimented with various hyper-parameters of parsing-based (SQLova) and grammar-based (IRNet) NL2SQL models, to study their impact on the model’s overall performance. As SQLova fine-tunes BERT, it is very slow. So, we present our results for 3 hyper-parameters and we manage to get good results by increasing vector dimension for input vector and encoder hidden layers and by using higher dropout of 0.5.

With IRNet, we first replace BERT transformer with smaller version of GPT2 and RoBERTa transformers and then we perform hyper-parameter optimization on three models IRNet with BERT (default), GPT2 and RoBERTa. We also generalize these models over sparc dataset by first preprocessing to modify complex sparc dataset to spider dataset format to answer our 2nd Research Question. We infer that SELU activation function with IRNet+RoBERTa and GELU activation function with GPT2 gives better results than IRNet+BERT. We also report better accuracy for Sparc dataset with lower learning rate and smaller decay rate with both IRNet+GPT2 and IRNet+RoBERTa. By increasing the dropout, we get good sketch accuracy and Exact Match accuracy for IRNet+GPT2 and IRNet+RoBERTa. Layer normalization also improves the performance of IRNet+BERT and IRNet+RoBERTa transformer model.

Finally, we also implement SQLova+GPT2 by replacing BERT encoders. We find that SQLova+GPT2 suffers a lot from overfitting. And hence needs more research. We conclude that grammar based model, IRNet is better than parsing-based SQLova as it could be generalized better and gave better results.

Overall we work with 4 versions of modified NL2SQL models i.e. IRNet+BERT, IRNet+GPT2, IRNet+RoBERTa and SQLova+BERT and highlight the importance of using competing transformer models and performing hyper-parameter optimization to explore the full potential of models in question. And we find that GPT2 is equally competitive as BERT, and better for Sparc dataset when used with larger sized contextual vector, large enough to capture the output from the these transformers. We also find that, using better activation functions like SELU without layer normalization and GELU and
Leaky ReLu with layer normalization can lead to equally better and competitive results.

We also mention that encoder utilizing embeddings of transformers should have a number of neurons ranging between both the length of the sequence allowed and the transformer vector size. We discourage the use of smaller hidden size in the encoder layer attached to large transformer models which was the case with the original SQLova model.

7.2 Future Work

Due to the vast options and combinations of hyper-parameters, a separate study dedicated entirely on this subject matter would yield more conclusive results compared to our limited choice of selections with the parsing-based SQLova model. More recent activation function like Swish which has reported good results could be used with both of the researched NL2SQL models. A good parameter search, especially for learning-rate and decay can help for SQLova. In case of IRNet, we used a bigger beam size of 10 consistently to get better results. We think that with larger transformer models a bigger beam size of 20 or 25 can work better.

Using larger version of GPT2 and RoBERTa models could lead to even better results, as already reported by a very recent $R^2SQL$ model which does not work with schema information provided with Spider dataset but trains on larger dataset by augmenting Sparc with Wikitable dataset. We believe that training IRNet with larger RoBERTa or a GPT2 transformer can lead to improved results. Experiments using different layers of transformer models and performing interpretability analysis will also provide more understanding. Also, in case of Sparc, one more classification layer could be added to output the turn number in case of both SQLova and IRNet. And the model can be trained using a Multi-task loss combining classification and regression loss to predict SQL query and turn respectively (or cardinality, for example), might lead to better results.

While working with IRNet, its better adaptibility, results over other cross-domain dataset and runtime we see immense potential in this model especially when coupled with other transformer models. We also think that SEMQL grammar should be enhanced in order to explore other kinds of SQL queries like stored procedures, triggers etc taking in even larger context. But this also raises the need for such datasets containing DML and DCL queries. Hence, since current NL2SQL models only target DQL based SQL queries, in order to progress we also need better and larger datasets with DML and DCL SQL queries.

A recent pretrained language model called as TaBERT specifically for NL2SQL is released should also be used to perform joint training over the database and NL2SQL dataset. In future, coupling a chat-based system or a voice-based system with NL2SQL model and conducting user-study will provide insights to the usability and pain-points of such system.
Bibliography


[CCBL18] Yun Chen, Kyunghyun Cho, Samuel R. Bowman, and Victor O.K. Li. Stable and effective trainable greedy decoding for sequence to sequence learning, 2018. URL: https://openreview.net/forum?id=rJZlKFkvM.


[YZY+] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task.


Appendices
A Appendices

A.1 SQLova+GPT2 : Training and validation graph for sub modules

For the following figures, x-axis is epochs and y-axis is accuracy.

**Figure A.1:** Select column : training set

**Figure A.2:** Select column : validation set

**Figure A.3:** Aggregation operator : training set

**Figure A.4:** Aggregation operator : validation set
Figure A.5: Where Column: training set

Figure A.6: Where column: validation set

Figure A.7: Number of conditions in where clause: training set

Figure A.8: Number of conditions in where clause: validation set

Figure A.9: Where Operator: training set

Figure A.10: Where Operator: validation set
Figure A.11: Start Index of where value
: training set

Figure A.12: Number of conditions in
where clause: validation set

Figure A.13: Start Index of where value
: training set

Figure A.14: Number of conditions in
where clause: validation set

Figure A.15: IRNet with different transformers: Training graph for input vector size and embedding size over Sparc dataset
A.2 IRNet with different transformers: Training graph for input vector size and embedding size over Sparc dataset

A.3 IRNet with different transformers: Training graphs for learning rate and decay over Sparc dataset

Figure A.16: IRNet with different transformers: Training graphs for learning rate and decay over Sparc dataset

A.4 IRNet with different transformers: Training graphs for dropout over Sparc dataset
Figure A.17: IRNet with different transformers: Training graphs for dropout over Sparc dataset