FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Text to query using large language models on local infrastructure
Student:	Patrik Laurinc
Supervisor:	doc. Ing. Pavel Kordík, Ph.D.
Study program:	Informatics
Branch / specialization:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	until the end of summer semester 2024/2025

Instructions

Explore posibilities to transform text to SQL or other queries by large language models (LLM). Create a working prototype and test its capabilities on particular usecase such as mapping speech to ReQL calls. Find suitable LLM libraries and explore posibilities to run LLM inference on a local infrastructure. Focus on LLMs performance and resource usage when executing text to query tasks. Evaluate the inference system on testing tasks such as https://paperswithcode.com/task/text-to-sql

Electronically approved by Ing. Magda Friedjungová, Ph.D. on 26 April 2023 in Prague.

Bachelor's thesis

TEXT TO QUERY USING LARGE LANGUAGE MODELS ON LOCAL INFRASTRUCTURE

Patrik Laurinc

Faculty of Information Technology
Department of applied mathematics
Supervisor: doc. Ing. Pavel Kordík, Ph.D.
May 11, 2023

Citation of this thesis: Laurinc Patrik. *Text to query using large language models on local infrastructure.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Contents

List of Figures

List of Tables

Most of all, I would like to thank my supervisor, doc. Ing. Pavel Kordík, Ph.D., who guided me through the process of writing this thesis, provided valuable feedback and relevant information from the quickly evolving field of large language models. I am also grateful to all my friends and family who supported me during my studies.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of the university theses.

In Prague on May 11, 2023 ....

Abstract

In a world teeming with digital data, simplifying human interaction with databases is an ever-pressing challenge. This undergraduate thesis delves into the intriguing potential of large language models (LLMs) to transform natural text into query languages, such as SQL.

Keywords large language models, deep learning, query languages, natural language processing, transformers

Abstrakt

Ve světě plném digitálních dat je zjednodušení lidské interakce s databázemi stále naléhavou výzvou. Tato bakalářská práce se zabývá potenciálem velkých jazykových modelů (LLM) transformovat přirozený text na dotazovací jazyky, jako je SQL.

Klíčová slova velké jazykové modely, hluboké učení, dotazovací jazyky, zpracování přirozeného jazyka, transformers

Abbreviations

AI	Artificial Intelligence
IR	Information Retrieval
LLM	Large Language Model
ML	Machine Learning
NLP	Natural Language Processing
SQL	Structured Query Language

Chapter 1

Introduction

1.1 Motivation

The rapid increase in digital data in the last decade has changed how we store, process, and work with information. Databases are a crucial part of this, making storing and finding data easy. SQL is one of the most popular languages for accessing and managing databases, but people need technical skills to access data. In this bachelor's thesis, we will look at how large language models can help turn everyday language into SQL queries, making databases easier for users without much technical knowledge to access.

Natural language processing has improved exponentially in recent years, mainly by introducing powerful large language models like GPT-4 by OpenAI. These models are good at understanding and generating human language. They could help users ask questions in everyday language and get answers from databases using SQL.

While companies like Google and OpenAI are competing to build the most powerful language models, mainly by increasing their size and using new methods for training and fine-tuning, a third faction is rising to the occasion. Even though these companies hold their research in secrecy, only publishing fractions of their work, the open-source community started collaborating and rapidly producing their methods and models. It all started when the open source community got their hands on their first really capable foundation model, the Meta's LLaMA, that was leaked to the public. It did not use state-of-the-art techniques like instruction/conversation fine-tuning or Reinforcement Learning from Human Feedback (RLHF). However, the community understood the capability of this model and started experimenting. Many innovations were being produced, with days between major developments and breakthroughs.

The open-source community quickly released its variants of the model with instruction tuning, quantization, quality improvements, multimodality, RLHF, and many others. The community quickly solved the scaling problem and allowed individuals to run extremely capable models on consumer hardware, even on the phone. With the barrier to entry becoming marginally smaller, much recent research shifted from major research organizations to individual researchers running their experiments on a single laptop. Soon, despite being significantly smaller, the open-source community started producing models that had comparable performance to models like OpenAI GPT-4 or Google Bard. One of the models, Vicuna-13B[1] containing 13 billion parameters, achieved more than 90% quality (rigorous evaluation needed, output was graded by GPT-4 itself) of OpenAI GPT-4, which is rumored to have 1 trillion parameters, 77 times more than Vicuna-13B. Finally, the training cost of GPT-4 was more than $100 million. Meanwhile, Vicuna-13B had a cost of just $500.

The open source community has proven that the performance of enormous models like GPT-4 can be almost achieved by using specialized smaller models by carefully fine-tuning and improving the methods used during training rather than scaling up the size. These facts open up enormous possibilities for applying LLMs in the real world.

1.2 Goals of the thesis

In this thesis, we will explore and evaluate multiple LLMs, comparing their performance on benchmark datasets to determine their efficacy in translating natural language queries to SQL. By analyzing the strengths and weaknesses of each model, we aim to identify the most promising candidates for further development and optimization. This comprehensive comparison will provide valuable insights into the capabilities of LLMs in the context of natural language-to-SQL translation, thereby informing the future trajectory of human-database interactions.

1.3 Outline of the thesis

The thesis is organized as follows. In Chapter 2, we will discuss the background of the topic and the related work. In Chapter 3, we will describe the datasets and the evaluation metrics. In Chapter 4, we will describe the models and the training process. In Chapter 5, we will present the results of the experiments. In Chapter 6, we will discuss the results and the future work.

Chapter 2

Literature review

As we venture into an increasingly digital and data-driven era, the ability of machines to understand and interact with natural human language has become a crucial component of modern technology. Natural Language Processing stands at the forefront of this revolution, enabling more intuitive human-computer interactions and expanding possibilities in data analysis, information retrieval, and automation. Coupled with the power of artificial neural networks, NLP has seen significant advancements, from understanding simple text commands to generating human-like text.

This chapter embarks on a journey through the evolution of NLP and artificial neural networks, delving into the core techniques and architectures that have shaped this field. We explore the world of large language models, particularly key models like GPT, BERT, and T5, that have been exemplary in pushing the boundaries of what machines can understand and generate.

We also examine the intricate task of transforming natural language into Structured Query Language (SQL), discussing various approaches, challenges, and models that have emerged in recent research. This literature review provides a foundation for understanding the current state of NLP and a detailed look into our explorations and finding in the following chapters.

2.1 Natural language processing

Natural language processing (NLP) first appeared in the 1950s as a combination of artificial intelligence (AI) and linguistics. Originally, NLP was distinct from textual information retrieval (IR), which focuses on storing, retrieving, and evaluating information from document repositories, especially textual information using statistics-based techniques.

Early approaches to NLP were based on formal grammar and rules, but these approaches were unsuccessful. The reason is that NLP must be able to extract meaning ("semantics") from the text. Formal grammars address syntax primarily, and while they can be extended to address natural language semantics, the rules would become unmanageable and unpredictable with more frequent ambiguous parses (the same word sequence having multiple possible interpretations).

The 1980s sparked a fundamental reorientation in NLP with the introduction of statistical methods. Machine learning (ML) methods that used probabilities became prominent. ML algorithms were trained using large, annotated documents of text (corpora) - the annotations contained correct answers - and provided gold standards for evaluation.[2]

In the late 2000s and early 2010s, the field of NLP was revolutionized by the introduction of neural networks and deep learning approaches. The recurrent neural network (RNN) was a prominent model in this era. However, it also came with problems - primarily the vanishing and exploding gradient problems - these are distinct problems, although related to the difficulty of training deep RNNs. This problem gets more profound the deeper the network is. RNN models were also hard to train because of the sequential approach, which did not allow parallel computations, and every hidden state was dependent on a hidden state at time step minus one. The vanishing gradient problem has since been addressed by introducing long short-term memory and gated recurrent units (GRU). While relatively significant advancements, these approaches did not solve the problem directly, and other issues with training deep RNNs still exist.

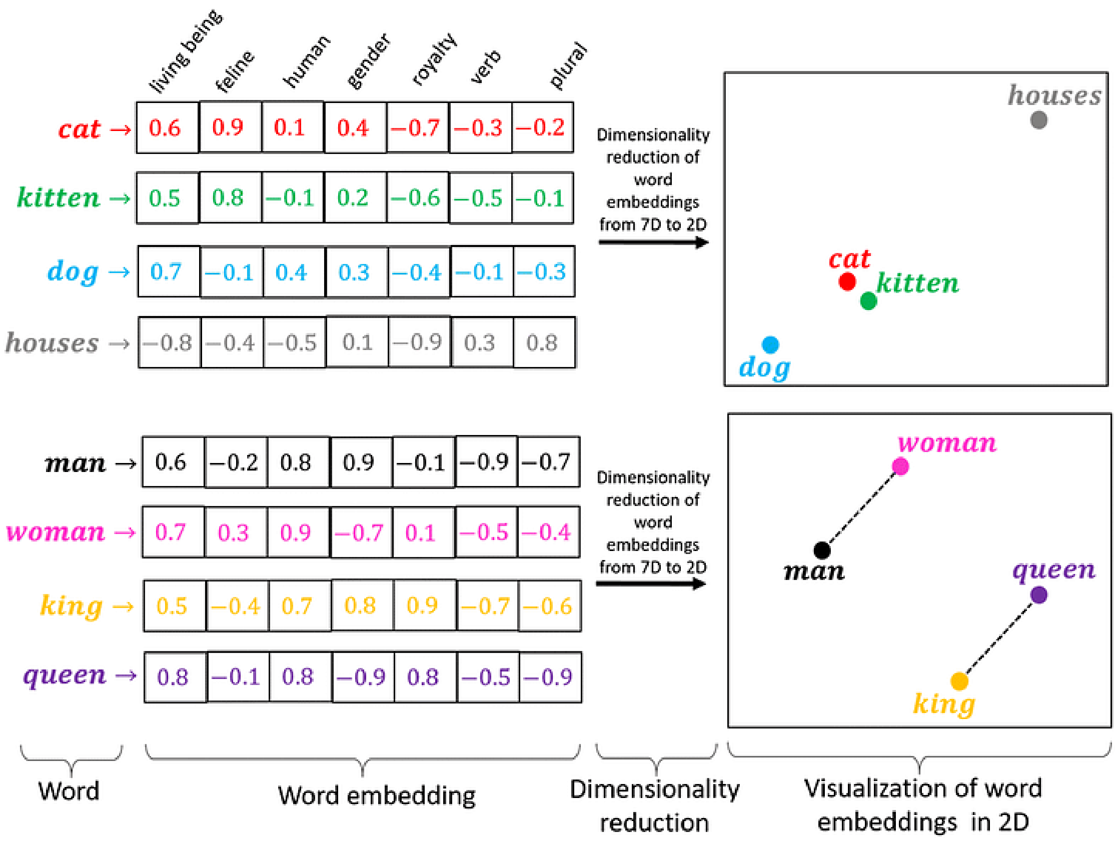
Recent years have seen the most considerable advancement in NLP via the introduction of transformer models (Vaswani et al. in 2017) with a self-attention mechanism, which solved the mentioned RNN model's problems. Transformer models also have their downsides, but the benefits far outweigh them and allow bigger models to be trained faster, cheaper, and achieve greater depth. We will focus on transformer models more in-depth in section 2.2.3.

2.1.1 NLP techniques

Word embeddings

In NLP, word embedding is a term used for the representation of words and used for text analysis. Typically, word embeddings represent a real-valued vector that encodes the word's meaning so that words with similar meanings are expected to be close to each other and vice versa.

Methods for producing word embeddings typically include probabilistic models, dimensionality reduction on the word co-occurrence matrix, and mainly neural networks. In practice, ML practitioners do not usually train their word embeddings but utilize pre-trained embeddings available on the internet. Different word embeddings are distinct mainly in two aspects - the number of dimensions and the vocabulary size.[3]



	living being	feline	human	gender	royalty	verb	plural
cat →	0.6	0.9	0.1	0.4	−0.7	−0.3	−0.2
kitten →	0.5	0.8	−0.1	0.2	−0.6	−0.5	−0.1
dog →	0.7	−0.1	0.4	0.3	−0.4	−0.1	−0.3
houses →	−0.8	−0.4	−0.5	0.1	−0.9	0.3	0.8

man →	0.6	−0.2	0.8	0.9	−0.1	−0.9	−0.7
woman →	0.7	0.3	0.9	−0.7	0.1	−0.5	−0.4
king →	0.5	−0.4	0.7	0.8	0.9	−0.7	−0.6
queen →	0.8	−0.1	0.8	−0.9	0.8	−0.5	−0.9

Dimensionality reduction of word embeddings from 7D to 2D
Dimensionality reduction of word embeddings from 7D to 2D
houses
cat
kitten
dog
woman
man
queen
king
Word
Word embedding
Dimensionality reduction
Visualization of word embeddings in 2D

Figure 2.1 Example representation of word embeddings visualized in 2D through dimensionality reduction. Image was taken from [4]

2.2 Artificial neural networks

Artificial neural networks are one of the most popular topics in machine learning in the academic community and industry. Neural networks, and subsequently deep learning, gather inspiration from how the human brain is designed.

Perceptron

Perceptron, also called an artificial neuron, is the most straightforward neural network and also one of the building blocks of deep learning. Similarly to biological neurons, the perceptron algorithm acts as an artificial neuron. It is connected to multiple inputs, where each input has an associated weight that yields an output together. The basic form of the perceptron algorithm for binary classification tasks is the following:

$$y(x1, x2, ..., xn) = f(w1x1 + w2x2 + ... + wnxn) \tag{2.1}$$

Where x_i is the input, w_i is the weight, f is the activation function (usually sigmoid, tanh or ReLU), and y is the output.

The perceptron algorithm learns a hyperplane that separates the data into two classes. The problem is that the hyperplane cannot shift away from the origin. To overcome this problem, we can introduce a new variable called bias. Bias will have a constant input of 1, therefore, the perceptron will be able to move away from the origin and separate data correctly.[5]

The updated form of perceptron alrorithm with bias can be expressed as:

$$y(x1, x2, ..., xn) = f(w1x1 + w2x2 + ... + wnxn + b) \tag{2.2}$$

2.2.1 Feed-forward neural networks

Feed-forward neural networks are the most common type of neural network. They are called feed-forward because the information flows in one direction - from the input layer to the output layer. Perceptron is the simplest type of feed-forward neural network with only one layer. In practice, however, feed-forward networks are comprised of multiple layers of stacked neurons, where the neurons between each layer are all connected.[5]

The computation (also called the forward pass) can be expressed using the matrix form of the perceptron formula:

$$y = \boldsymbol{f}(\boldsymbol{W}x + \boldsymbol{b}) \tag{2.3}$$

2.2.2 Recurrent neural networks

Many situations in NLP require capturing long-term dependencies and contextual order between words. This is challenging with ordinary feed-forward neural networks, as they do not have any memory. Traditionally, NLP practitioners used Hidden Markov Models (HMMs) to compute contextual information. The main problem with this approach is that HMMs are limited to a fixed number of previous steps when generating each prediction. Recurrent neural networks (RNNs) relax this constraint by accumulating information from each time step into a ”hidden state.” This ”condenses” sequential information, and the model can generate predictions using the entire sequence history.

Another advantage of RNNs over HMMs is their ability to process variable-length sequences. This allows samples of different lengths to be mapped into the same feature space and allows comparison between such samples.

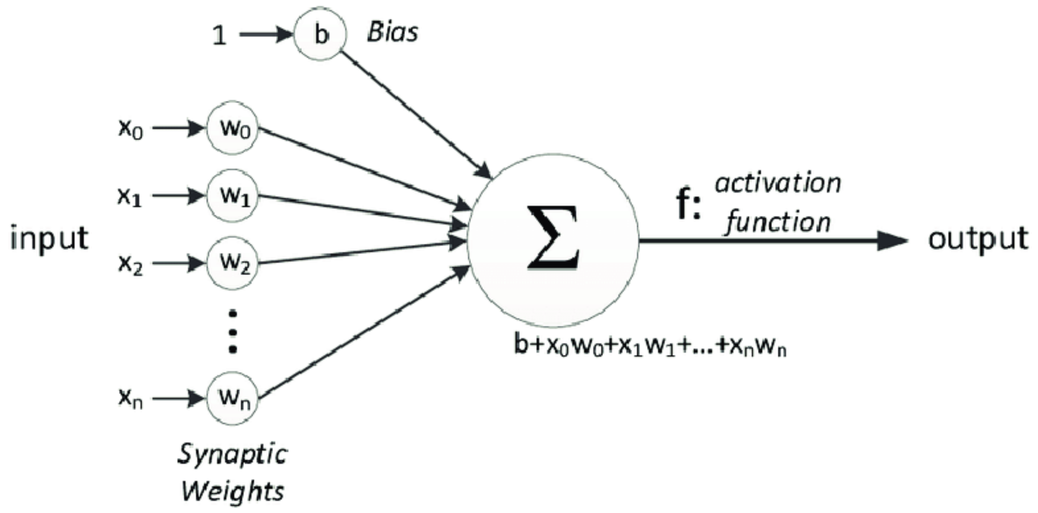


Figure 2.2 Illustration of a perceptron. Image was taken from [6]

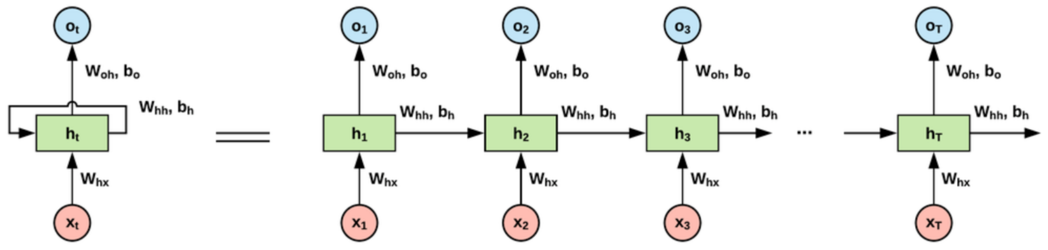


Figure 2.3 Illustration of a simple RNN. Image was taken from [7]

Recurrence and memory

Let us have a T length input sequence as X, where $X = \{x_1, x_2, ..., x_T\}$, such that $x_t \in \mathbb{R}^N$ is the vector input at time t. We then can define our memory up to time t as $\boldsymbol{h_t}$. The output can then be expressed as:

$$\boldsymbol{o}_t = f(\boldsymbol{x}_t, \boldsymbol{h}_{t-1})) \tag{2.4}$$

The memory from the previous time step is used to compute the output at the current time step. For the initial time step $\boldsymbol{x}_1, \boldsymbol{h}_0$ is equal to the vector $\mathbf{0}$.

Output $\boldsymbol{o}_t$ is considered to have condensed the information from the entire input sequence up to time step t. This provides us with the equation:

$$\boldsymbol{h}_t = \boldsymbol{o}_t = f(\boldsymbol{x}_t, \boldsymbol{h}_{t-1}) \tag{2.5}$$

Now we can easily see where the recurrence comes from. Output is directly dependent on the previous result and applying the same function. We can extend this concept to neural networks by using the matrix form of the previous formula:

$$\boldsymbol{h}_t = f(\boldsymbol{W}_x \boldsymbol{x}_t + \boldsymbol{W}_h \boldsymbol{h}_{t-1} + \boldsymbol{b})) \tag{2.6}$$

2.2.3 Transformers

Before delving into transformer architecture, it is essential to understand the limitations of previous deep-learning architectures and the key innovations that enabled the development of transformers. Before transformers, RNNs, and their advanced variants LSTMs, were the dominant model architectures for sequential data. One major challenge in these models, particularly in sequence-to-sequence tasks, is handling long-range dependencies in the input. This issue arises because all of the information about the input sequence is stored inside a fixed-length vector called the context vector. This problem gets more significant the longer the sequence is - encoding 100 tokens versus 1000 tokens into a vector of the same length will encode a smaller portion of the information. Therefore the decoder knows about the input sequence only what is sent inside the first step of the decoder. The attention mechanism was introduced as a solution to this problem.

Attention mechanism

The attention mechanism allows the decoder to focus selectively on different parts of the input sequence when decoding each token. Here is a high-level overview of how the attention mechanism works:

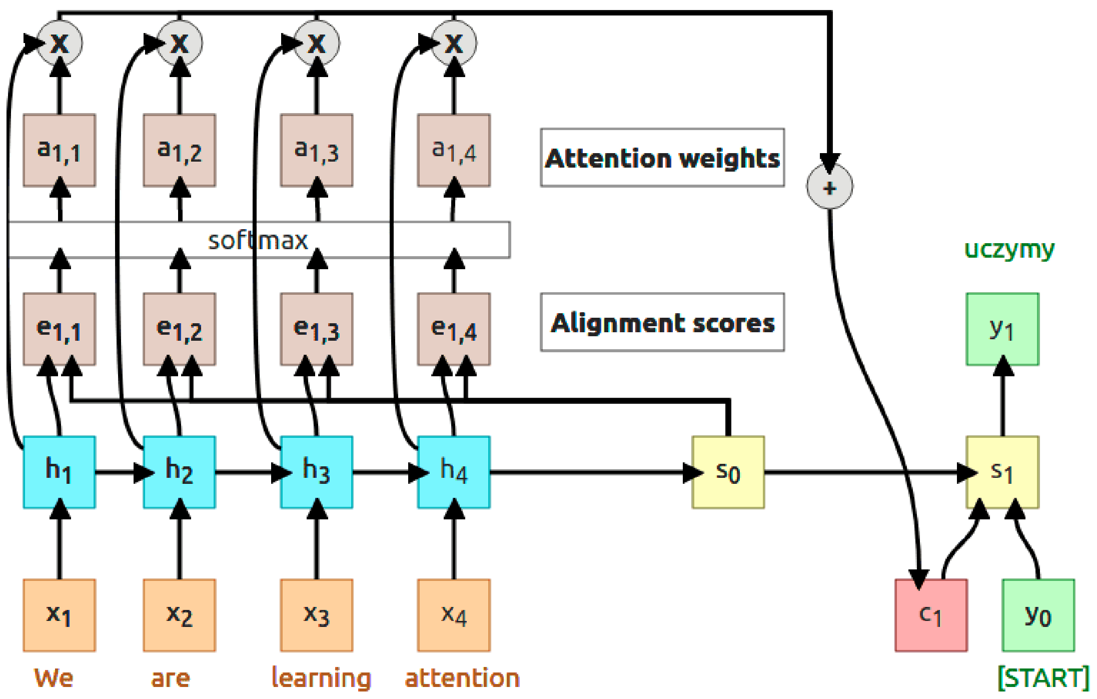


Figure 2.4 Illustration of the attention mechanism. Image was taken from [8]

1. **Encoding** The input sequence is first processed by an encoder (usually an RNN or LSTM). The encoder outputs a sequence of hidden states, one for each token at a specific position in the input sequence.
2. **Calculating attention weights** The decoder then calculates an alignment score for each hidden state of the encoder. This alignment score indicated the relevance of the given hidden state at the input position to the current output position. These scores are typically calculated using some similarity measure (e.g., dot product) between the decoder's hidden state at position I and each of the encoder's hidden states. Intuitively, we would want to take the hidden state of the encoder with the highest alignment score and pass that into the decoder-an argmax operation. However, argmax is not differentiable. Therefore, we use a softmax function which produces attention weights.
3. **Calculating context vector** The context vector is computed by taking the weighted sum of the encoder's hidden states, where the weights correspond to the attention weights calculated in the previous step. This context vector captures the most relevant information from the input sequence for generating the current output.
4. **Decoding** The last step combines the context vector with the decoder's hidden state to generate the following output token. This process is repeated until the decoder generates an end-of-sequence token.

Transformer architecture

The Transformer architecture was introduced in the paper ”Attention is all you need” [9] and revolutionized the field of natural language processing. It is designed to address the limitations of RNNs and LSTMs, namely the inability to handle long-range dependencies and parallelization during training. The Transformer architecture relies heavily on the attention mechanism, does

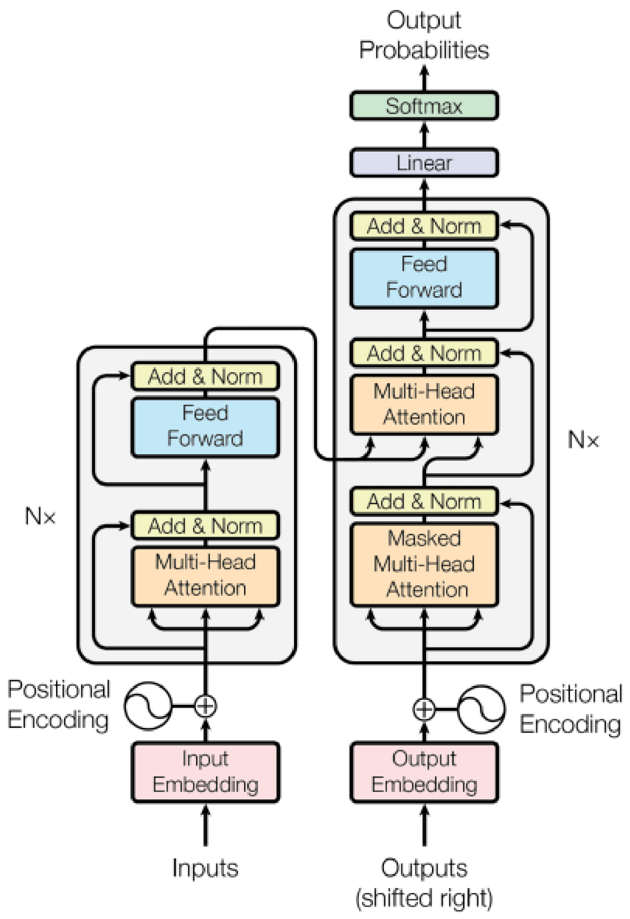


Figure 2.5 Illustration of transformer model architecture. Image was taken from [9]

not use recurrent or convolutional layers, and does not need to process sequences sequentially. Here is an overview of the architecture:

1. **Input embedding** The input sequence is first tokenized and embedded into a sequence of vectors by multiplying the input sequence with an embedding matrix learned during training. The embedding matrix contains a vector for each token in the vocabulary.

2. **Positional encoding** Because Transformer architecture does not rely on recurrent layers, it has no knowledge about the order of the tokens in the input sequence. Therefore, the input sequence is augmented with positional encoding, a vector that encodes the token's position in the sequence. Positional encodings can either be fixed or learned during training.

3. **Encoder and decoder stacks** Transformer architecture uses a well-known encoder-decoder architecture. Each stack is composed of several identical layers (6 in the original paper). The encoder stack encodes the input sequence into a sequence of hidden states, while the decoder stack decodes the hidden states into the output sequence.

4. **Multi-head attention** One of the main components of the Transformer is the multi-head attention mechanism. It is an improved version of the attention mechanism described in the

previous section. It is called multi-head attention because it uses several attention mechanisms in parallel - the number of heads is a hyperparameter of the model. The benefit is that each head can learn to attend to different parts of the input sequence. Every head uses so-called ”scaled dot-product attention,” which is essentially just an attention mechanism scaled by the square root of the dimensionality of the critical vectors. The weighted sums from all heads are concatenated into a single vector that captures a wide range of relationships in the input.

5. **Position-wise feed-forward network** The position-wise feed-forward network is a fully-connected network applied to each position separately and identically. It consists of two linear transformations with a ReLU activation in between.[9]

6. **Residual connections and layer normalization** Residual connections around every sub-layer (multi-head attention and FFNN) allow gradients to flow directly through the network. Layer normalization is applied to the output of every sublayer before it is passed to the next sublayer.

7. **Encoder** The encoder stack consists of multi-head attention and position-wise feed-forward network sublayers. The final output of the encoder is then passed to the decoder stack.

8. **Decoder** The decoder stack also has a multi-head attention sublayer, but it is masked. This is to prevent the decoder from attending to future tokens. Additionally to the standard position-wise feed-forward network, the decoder stack contains an encoder-decoder attention sublayer, which allows the decoder to attend to the encoder’s output. The decoder then generates one token at a time using a linear layer followed by a softmax layer that produces token probabilities.

Beam Search

Beam search is a heuristic search algorithm used in many NLP tasks, especially in sequence-to-sequence problems and Transformer models used in this work. The algorithm’s goal is to find the most likely sequence of tokens given a starting point. This is done by keeping a set of the most probable sequences found so far and iteratively expanding them. Beam search balances a greedy approach (always choosing the most likely next token) and exhaustiveness (considering all possible sequences). Changing the beam width can make the search more greedy (small beam width) or more exhaustive (large beam width). Beam search does not guarantee to find the most probable sequence, which would require having a beam width equal to all possible sequences, which is infeasible for long sequences and extensive vocabularies.

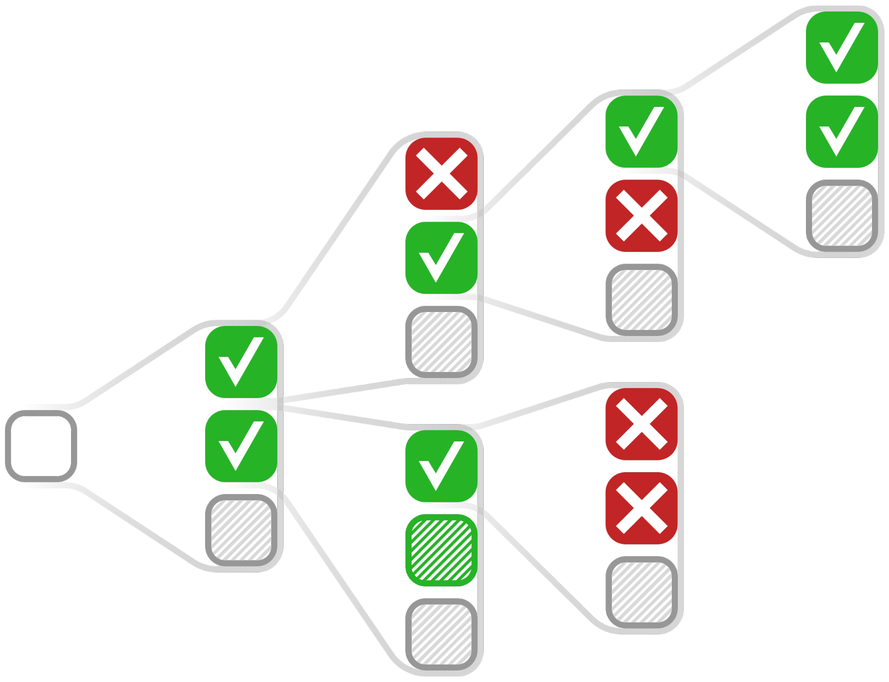


Figure 2.6 Illustration of beam search. Image was taken from [10]

2.2.4 Transfer learning

Transfer learning is a technique used in machine learning where a model trained on one task is used as a starting point for training on a different task. The idea is that the knowledge gained from the source task can be transferred to the target task. Transfer learning is beneficial when the target domain has limited labeled data or it is computationally costly to train the model from scratch. The source and target task should share similarities or related features to allow the model to reuse some of its knowledge. Transfer learning has been especially successful in deep learning, mainly in computer vision and NLP. For example, models like T5, GPT, and other LLMs have been pre-trained on massive datasets and can be fine-tuned for various specific tasks, often achieving great state-of-the-art results at a fraction of the cost and time consumption.[11]

2.3 Large language models

Large language models (LLMs) are a type of deep learning models that are trained massive amounts of unlabeled text data, which allows them to generate human-like responses to various questions and prompts. While LLMs are trained on unlabeled data, the massive size of the data allow them to build a statistical model of language that can then be used to generate new text, answer questions and many other language-related tasks. LLMs have gained a lot of popularity in the recent years, because they provide a strong baseline for a lot of tasks and even achieve SOTA performance in some of them out of the box.

2.3.1 History and development of large language models

Large language models (LLMs) are a type of deep learning model trained on massive amounts of unlabeled text data, allowing them to generate human-like responses to various questions and prompts. While LLMs are trained on unlabeled data, the massive size of the data allows them to build a statistical model of language that can then be used to generate new text, answer questions, and do many other language-related tasks. LLMs have gained much popularity in recent years because they provide a strong baseline for many tasks and even achieve SOTA performance in some of them out of the box.

2.3.2 GPT

GPT (Generative Pre-trained Transformer) architecture is largely based on the original Transformer model, which consists of an encoder and a decoder. However, GPT adopted a ”decoder-only” architecture, which means that it uses only the decoder part of the architecture and completely disregards the encoder. Originally in the Transformer model, the encoder transformed input sequences into continuous representations, and the decoder generated sequences based on these representations. This approach simplifies the architecture and is generally better for unsupervised pre-training and many NLP tasks, mainly language modeling and text generation.[12]

In our experiments, we are using the most recent (publically available) version of the GPT model, the GPT-2. GPT-2, the successor to GPT-1, also comes in multiple sizes and configurations, with the largest one, GPT-2 XL, containing 1.5 billion parameters. The models were trained using a transformer-based architecture and unsupervised learning on a vast corpus of internet text.[13]

We will list the variations of the GPT-2 model and their parameters in the table below.

Model	**Layers**	**Parameters**	**Attention Heads**	**Feed-Forward Dimension**
GPT-2 Small	12	117M	12	3072
GPT-2 Medium	24	345M	16	4096
GPT-2 Large	36	774M	20	5120
GPT-2 XL	48	1.5B	25	6400

Table 2.1 The specifications of different versions of the GPT-2 model.

2.3.3 BERT

BERT (Bidirectional Encoder Representations from Transformers) is a large Transformer-based model developed by Google. Unlike GPT, which uses decoder-only architecture, BERT uses encoder-only architecture. This makes the model capable of outstanding performance in tasks that do not require generating text, such as text classification, named-entity recognition, and more. BERT, also unlike GPT, can understand bidirectional context thanks to its masked language modeling (MLM), a pre-training objective used during training. The MLM task involves randomly predicting masked tokens in the input sequence based on the whole sequence. This process allows BERT to learn bidirectional representations.[14]

2.3.4 T5

T5 (Text-To-Text Transfer Transformer) is another large Transformer-based model developed by Google. Unlike BERT, which uses only the encoder, T5 follows the original Transformer architecture more closely, having both the encoder and the decoder, which can generate text. In the T5 paper (Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer), researchers conclude that for all tasks tested in the paper, encoder-decoder architecture with the denoising objective performed best. Denoising objective refers to a learning goal where the model reconstructs a text’s original version from a ”noisy” version. This can be achieved via masked language modeling or reconstructing sentences from a shuffled set of tokens. Their encoder-decoder variant had the highest parameter count (2P) but the same computation cost as the P-parameter decoder-only models. The researchers also concluded that sharing parameters across the encoder and decoder performed nearly as well while having lower computational costs.[15]

T5 comes in multiple sizes and configurations, with the largest one, T5-11B, containing 11 billion parameters. The models were pre-trained using an unsupervised denoising objective on a 750 GB dataset of text from the Common Crawl, a corpus of web crawl data, and a mixture of datasets for supervised text-to-text language modeling objectives.[15]

We will list the variations of the T5 model and their parameters in the table below. The number of layers is per encoder/decoder. Therefore the total number of layers is double the number in the table.

Model	Layers	Parameters	Attention Heads	Feed-Forward/Projection Dimension
T5-Small	6	60M	8	2048
T5-Base	12	220M	12	3072
T5-Large	24	770M	16	4096
T5-3B	24	3B	32	16384
T5-11B	24	11B	64	65536

Table 2.2 The specifications of different versions of the T5 model.

2.4 Previous research on transforming natural language to SQL

Research on natural language to SQL (NL2SQL) can be traced back to the 1970s. Since deep learning approaches were not present yet, such systems relied on transforming text to intermediate logical representation, which was then transformer into SQL. Other possibilities were using rule-based systems. However, these systems heavily relied on hand-crafted mapping rules for translation. Therefore complex queries or generalization across more DB schemas were complicated. Natural language processing advancements like natural language parsers allowed more advanced rule-based systems to be made. However, it was not until the integration of deep learning approaches that NL2SQL started achieving greater results and generalization across unseen schemas. We will look at some approaches for NL2SQL task and some breakthrough models.[16]

2.4.1 Output decoding approaches

Text-to-SQL systems following the encoder-decoder architecture can have different approaches to decoding the output of SQL queries. The most common approaches are sequence-based decoding, sketch-based slot-filling decoding, and grammar-based decoding. We will look at each of these approaches in more detail.

Sequence-based decoding

Sequence-based decoding includes systems that generate the predicted SQL as a sequence of words. This is the simplest of the decoding techniques, and it was adopted by Seq2SQL[17], which is one of the first deep-learning text-to-SQL systems. However, it is also prone to errors. Therefore later systems steered away from this approach. The reason for that is that sequence decoding treats the SQL query as a sequence that needs to be learned, and during inference, there are no measures to ensure the produced SQL query is syntactically correct. It does not consider the strict SQL grammatical rules, nor does it have any mechanism to prevent generating incorrect columns and table names. Despite these drawbacks, sequence-based approaches are starting to be used again. They are achieving great results thanks to two advances: the introduction of large pre-trained Transformer seq2seq models and decoding techniques that constrain the output of the decoder and prevent it from generating invalid queries (e.g., PICARD[10]).

Sketch-based slot-filling decoding

Systems in this category try to simplify the difficult task of generating a SQL query into a more manageable task of predicting certain parts of the query, such as the columns in the WHERE clause. This approach transforms the generation task into a classification task. In particular, we use a query sketch with a certain number of empty slots to be filled and develop neural networks to predict the most probable elements for each slot. While this approach divides the text-to-SQL into smaller sub-tasks, it has two drawbacks. The first one is that the neural net architecture can be quite complex, having a dedicated network for every part of the query sketch. The second one is difficulty in extending to complex SQL queries because generating sketches is not a trivial problem in itself.

Grammar-based decoding

Grammar-based decoding approaches can be seen as an extension of sequence-decoding approaches. Instead of producing a sequence of tokens composing a SQL query, they produce a sequence of grammar rules. These grammar rules, when applied, can create a structured query. Most often used grammar-based decoders have been previously proposed for code generation as an Abstract Syntax Tree (AST). These models consider the grammar of the target language (SQL in our case), and the target program is considered an AST. All nodes of the AST are expanded at every tree level using the grammar rules until all branched reach a terminal rule. Because the predictions are based on the grammar and the state of the AST, it makes generating grammatically incorrect queries less probable. Although they were once considered the most advantageous option, their status is currently challenged by sequence-based decoding approaches mainly due to the rise of Transformer models. It is possible that more advanced decoding techniques may replace grammar-based decoding in the future, as the field has been rapidly progressing in recent years.[18]

2.4.2 Seq2SQL

The Seq2SQL model, introduced in the paper "Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning"[17], was a very influential model for the text-to-SQL problem. This model is based on a seq2seq architecture, specifically the encoder-decoder architecture. The Seq2SQL model uses reinforcement learning (RL) to address the SELECT, WHERE, and AGGREGATION clauses in SQL queries. Here's a high-level overview of this architecture:

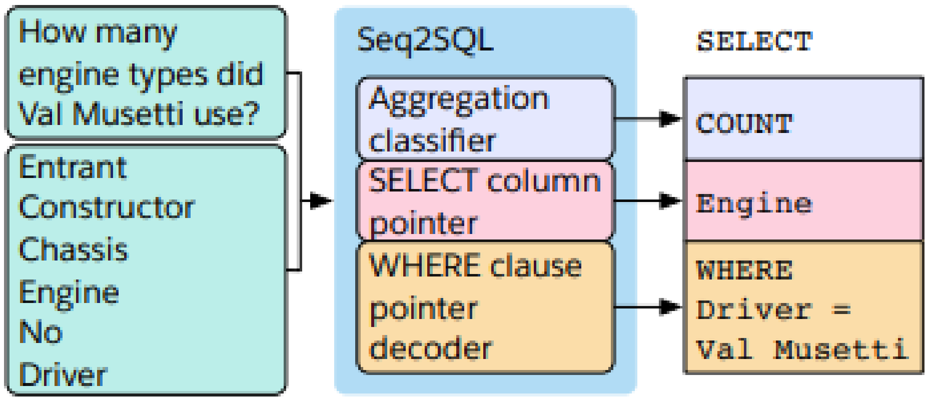


Figure 2.7 Illustration of the architecture used for composing the SQL query. The input consists of Natural Language question and table columns. Image was taken from [17]

1. **Encoder** The input is first processed by an encoder, which is an LSTM network. The encoder is used to encode the input sequence into a fixed-length context vector that captures the semantics of the input sequence.
2. **Decoder** The decoder, also an LSTM network, takes the context vector from the encoder and generates the output SQL query token by token. The decoder is composed of three main components to handle different parts of the SQL query:
   - **Column Pointing network** This network is used to predict the columns that are involved in the SELECT and WHERE clauses.
   - **Aggregation Prediction network** This network predicts the aggregation function (e.g., COUNT, MAX, MIN, etc.) applied to the column in the SELECT clause.
   - **WHERE Prediction network** This network predicts the WHERE clause's conditions, which are composed of a column, an operator (e.g., =, LIKE), and a value.
3. **Reinforcement Learning** The Seq2SQL model uses a policy-gradient-based RL algorithm called REINFORCE to train the model. The idea behind using RL is that the model is trained to maximize the reward, which is the accuracy of the generated SQL queries, instead of relying just on the cross-entropy loss used in typical seq2seq models.
4. **Training** Training of the model consists of a two-step process. First, the model is trained using supervised learning (SL) with cross-entropy loss. This helps the model learn the general structure of SQL queries. Then, the model is fine-tuned using RL with the execution-guided reward signal to generate more accurate SQL queries.

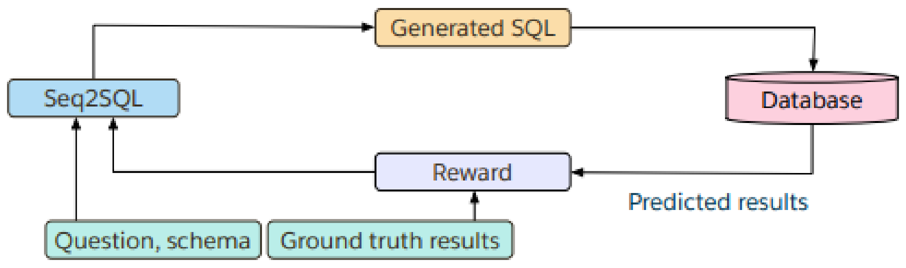


Figure 2.8 Illustration of the training process using RL. Image was taken from [17]

The Seq2SQL model demonstrated significant improvements over the previous model for the text-to-SQL task, mainly in the ability to handle complex queries with multiple WHERE clauses conditions.

2.4.3 RAT-SQL

RAT-SQL is a model that was introduced in the paper "RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers"[19]. This model is based on a Transformer architecture and designed to address the challenges of encoding and linking complex database schema information. This is achieved by using a particular type of attention mechanism called "relation-aware self-attention," which captures the relationships between different database schema elements, such as tables and columns. Here is an overview of the model architecture:

1. **Input Representation** The input to the model consists of natural language questions, database schema, and the set of relations between the tables in the schema. The question is tokenized together with the schema, and the relations are encoded as edges in a graph.
2. **Relation-aware Schema Encoding** The schema is encoded using a Transformer-based encoder. In the paper, the researchers tried two approaches to the encoding part. One used a bi-directional LSTM network to produce the initial representation. The second used a pre-trained BERT model. Separate encoders are run over the question, and the schema and the relation-aware self-attention mechanism are used to produce joint representations from the combined representations of the question and schema.
3. **Decoder** The decoder is a bi-directional LSTM that follows grammar-based decoding. It generates the SQL program as an AST in depth-first traversal order. The AST is then expanded at every node until it terminates in a leaf node, producing a complete SQL query.

RAT-SQL's relation-aware schema encoding and linking mechanisms demonstrated significant improvements over previous models in the text-to-SQL problem, such as Seq2SQL, especially on datasets with complex schemas and relations (e.g., Spider).

2.4.4 Graphix-T5

Graphix-T5 is a model that was introduced in the paper "Graphix-T5: Mixing Pre-Trained Transformers with Graph-Aware Layers for Text-to-SQL Parsing"[20]. This model is based on the T5 model used and described in this work. It is a current state-of-the-art model on numerous evaluation datasets, including Spider.

This model uses a Transformer-based encoder-decoder architecture. The decoder is the original T5 pre-trained decoder block, while the encoder is a modified version of the T5 encoder block. The original Transformer layers are still pre-trained. However, the layers are modified to add a Graph Neural Network (GNN). The researchers call this a "Graphix layer," and it works similarly to the original T5 encoder layers. First, the input sequence is taken in the same way as in T5. It gets encoded into a hidden state through multi-head attention and a feed-forward network. They call this "Semantic Representation". However, the researchers have shown that the joint input question and database schema can be displayed as a heterogeneous graph. Therefore, in each Graphix layer, a second type of representation is produced called "Structural Representation" by using the relational graph attention network (RGAT)[21]. The hidden states of both representations are then added together to form a final hidden state containing both semantic and structural information. The researchers have shown that this method is more effective than previous works, which simply added a GNN between the original encoder and decoder blocks.

Chapter 3
Methodology

In this chapter, we will describe the methodology used in this research. We will start by describing the datasets chosen for training and evaluating our models. Then, we will present the methods and metrics used for the evaluation of the model's performance. Finally, we will focus on describing the tools and libraries used for implementation, followed by a detailed explanation of our implementation.

3.0.1 Description of the datasets

In the past years, many publications have been concerned with the text-to-SQL task and relevant benchmarks. Creating a dataset for this task is a challenging task on its own, as it requires a lot of manual labor to create a dataset that is both extensive and consists of complex queries. Two of the most popular datasets include the WikiSQL[17] and Spider[22] datasets. While WikiSQL is a large dataset containing 80654 hand-annotated questions with related tables and queries, almost all of the queries are simple and use only a simple table without any foreign key. Therefore, in this thesis, we will focus on the Spider dataset and its variants, which are much more complex, focusing on cross-domain semantic parsing and difficult queries with multiple aggregations and nested queries.

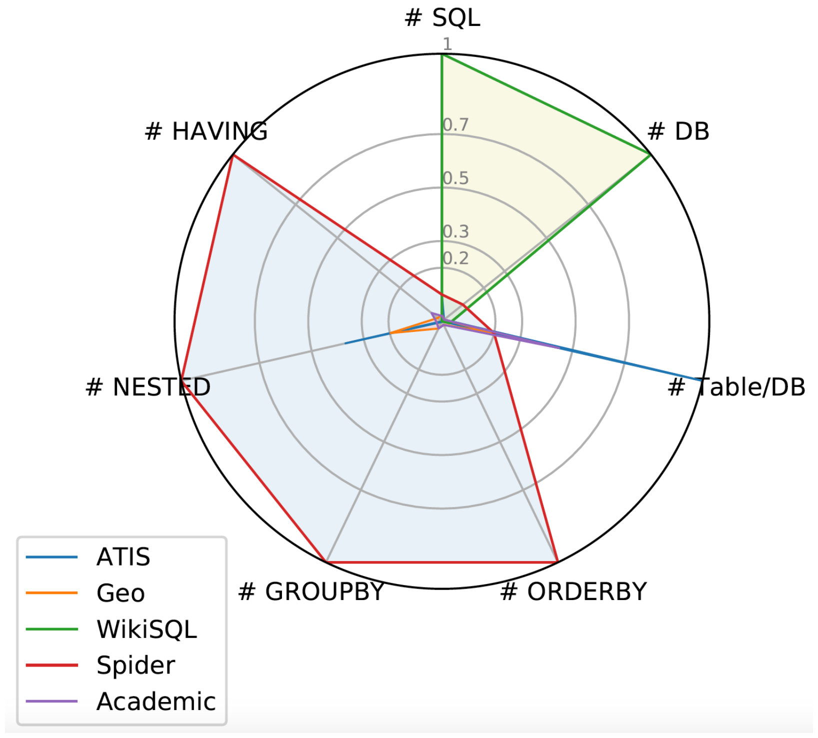
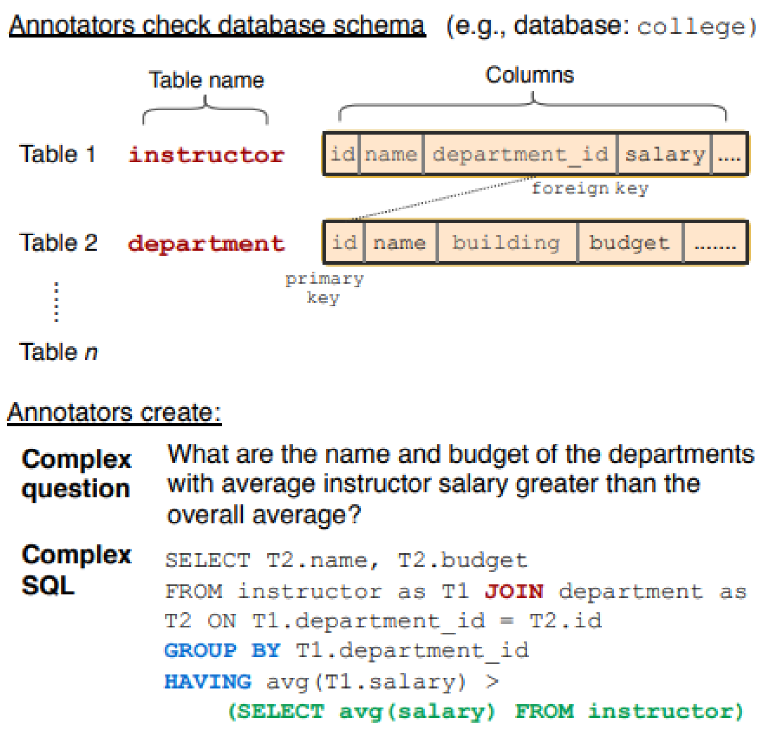


Figure 3.1 Comparison of different datasets designed for the text-to-SQL problem. Image was taken from [22]

3.0.1.1 Spider

Spider is a large-scale dataset designed for complex and cross-domain semantic parsing and text-to-SQL tasks. The dataset is annotated by 11 Yale students who are proficient in the SQL query language. It consists of 10,181 questions, 5,693 unique complex SQL queries, and 200 databases from 138 domains. In the train and test sets, different complex SQL queries and databases appear, which require the model to not only generalize well to new SQL queries but also unseen database schemas.



What are the name and budget of the departments with average instructor salary greater than the overall average?

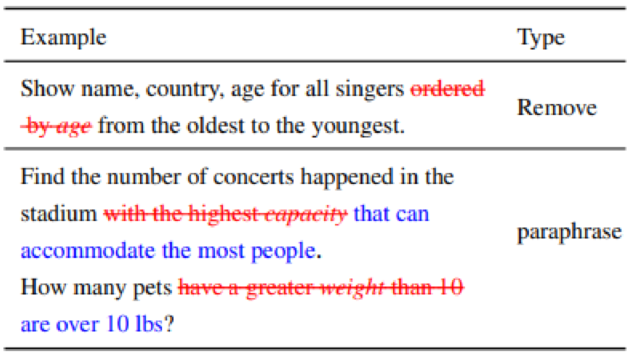
```
SELECT T2.name, T2.budget
FROM instructor as T1 JOIN department as
T2 ON T1.department_id = T2.id
GROUP BY T1.department_id
HAVING avg(T1.salary) >
    (SELECT avg(salary) FROM instructor)
```

Figure 3.2 Example of the Spider dataset annotation process and query. Image was taken from [22]

The Spider dataset became really popular in the field of text-to-SQL research, as it is much more complex than the WikiSQL dataset, which was the most popular dataset before Spider. This gave other researches the idea to extend Spider dataset with more advanced use-cases and difficulties. We will look into some of the variations of Spider dataset.

3.0.1.2 Spider Realistic

While the Spider dataset is a great and widely used benchmark for text-to-SQL problem, it eases the task by using names and keywords that closely match their paired SQL queries. An example is mentioning the column names in the question, meanwhile in practice natural language references to columns usually differ from the column names inside the database. To alleviate this problem, researchers proposed to train the models with cross-domain dataset like Spider, but add another eight single-domain datasets like ATIS[23] and Geo-Query[24] for evaluation. The problem is that some of the datasets differ a lot from Spider in query structures and dataset conventions, resulting in very poor performance of the model in some datasets. One proposed and accepted solution to this problem is a new realistic and challenging evaluation set based on Spider. The researchers selected a complex subset of the Spider dev set where there are columns compared against values or used in clauses like ORDER BY and HAVING. The researchers then manually modified the natural language questions in the subset to remove or paraphrase the explicit mentions of column names, except for columns inside the SELECT clause, while keeping the SQL queries unchanged.[25]

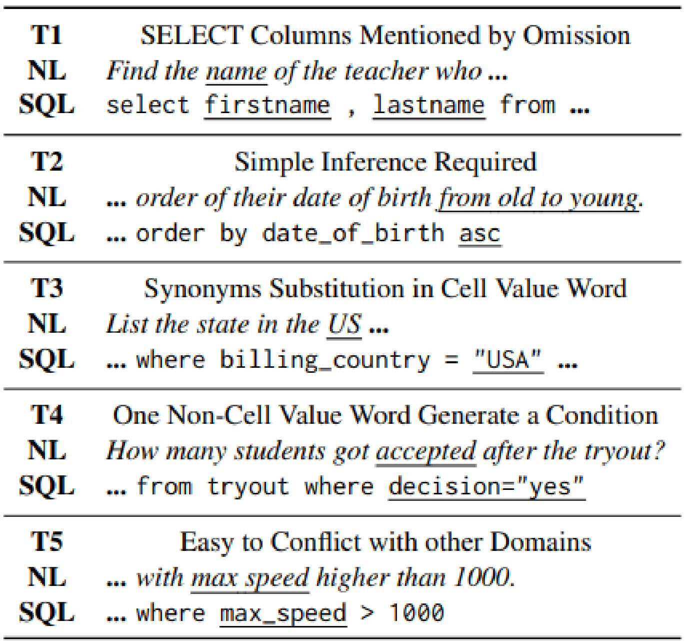


Example	Type
Show name, country, age for all singers ~~ordered by *age*~~ from the oldest to the youngest.	Remove
Find the number of concerts happened in the stadium ~~with the highest *capacity*~~ that can accommodate the most people. How many pets ~~have a greater *weight* than 10~~ are over 10 lbs?	paraphrase

Figure 3.3 Example of the Spider Realistic dataset changes. Image was taken from [25]

3.0.1.3 Spider-DK

In recent years, there has been significant progress in translating natural language into SQL queries under zero-shot cross-domain settings. Despite achieving excellent performance, recent works achieved over 70% accuracy on the Spider benchmark and over 90% on the WikiSQL benchmark, it doesn't imply that most problems in this field are solved. Numerous studies[26][25] showed that the generalization performance is much worse in more challenging scenarios. For example, the Spider Realistic dataset mentioned above investigated cases where explicit mentions of database columns are in the natural language question. Similarly, [27] observes that model accuracy drops when replacing schema-related words with synonyms. These papers showcase the important challenges of generalizing to new unseen databases. However, their performance degradations are somewhat expected because removing the explicit column mentions breaks the assumptions that make schema linking effective. Also, the SQL queries can be from different distributions than the training set, which also degrades performance. Spider-DK tries to demonstrate that the generalization performance can be poor even when both the natural language questions and SQL queries come from similar distribution to the training set. Spider-DK is a variant of the Spider dev set (a similar approach to Spider Realistic) that focuses on the evaluation of the model's understanding of domain knowledge. In this paper, the researchers introduce five types of domain knowledge and named them T1 to T5.



T1	SELECT Columns Mentioned by Omission
NL	*Find the name of the teacher who ...*
SQL	`select firstname , lastname from` ...
T2	Simple Inference Required
NL	... *order of their date of birth from old to young.*
SQL	... `order by date_of_birth asc`
T3	Synonyms Substitution in Cell Value Word
NL	*List the state in the US ...*
SQL	... `where billing_country = "USA"` ...
T4	One Non-Cell Value Word Generate a Condition
NL	*How many students got accepted after the tryout?*
SQL	... `from tryout where decision="yes"`
T5	Easy to Conflict with other Domains
NL	... *with max speed higher than 1000.*
SQL	... `where max_speed > 1000`

Figure 3.4 Example of the Spider-DK domain knowledge types introduced to dev set. Image was taken from [28]

3.0.2 Methods and metrics for evaluating and comparing the models

Meanwhile the models are trained to minimize the loss function, which is cross-entropy in our case, it is not a good metric for the text-to-SQL task. It does not take into account the structure, correctness, or even validity of the SQL query. Therefore, our evaluation metrics include Component Matching, Exact Matching, and Execution Accuracy.

3.0.2.1 Component Matching

Previous works directly compared the generated SQL query with the gold SQL query. However, this approach is not robust to the differences in the SQL query structure. For example, the model can generate a query with a different order of clauses, but the query is still correct. The Spider paper introduces a new approach that treats each SQL component as a set. The SQL query is split into the following components:

- SELECT
- WHERE
- GROUP BY
- ORDER BY
- KEYWORDS (includes all keywords without column names and operators)

We then decompose each component in the prediction and the ground truth as bags of several sub-components and check whether the two bags match exactly. For example, the SELECT max(col1), min(col1), max(col2) are decomposed into the following sets (max, min, col1) and max(col2). After that, we check if the gold and predicted sets are the same. The model's overall performance on each component is computed as an F1 score on exact set matching.

3.0.2.2 Exact-Match Accuracy

Exact-Match Accuracy is a metric used to evaluate the model's performance in the text-to-SQL task. It is a straightforward metric that calculates the percentage of the predicted SQL queries that are exactly the same as the ground truth SQL queries after being decomposed as described in the Component Matching subsection.

3.0.2.3 Execution Accuracy

Since Exact Matching can provide false negative evaluations when the model generates novel syntax structures, the Spider paper also uses Execution Accuracy. It is a metric that evaluates the model by executing the predicted SQL query on the database and comparing the values against the gold SQL query. Execution Accuracy can, however, also produce false negatives. For example, both gold and predicted SQL queries can return NULL while having different semantical meaning.

3.0.3 Description of the tools and libraries used for implementation

The whole codebase is written in the Python programming language, which is an industry-wide typical choice among machine learning practitioners. The main reason is the simplicity of the language and the vast amount of libraries and frameworks available, particularly for ML. All of the models were fine-tuned using the PyTorch framework either through a collection of Python scripts (T5 models) or a single Jupyter Notebook (GPT models).

PyTorch is a machine learning framework that is developed by Meta AI (previously known as Facebook AI Research). It is an optimized Python library for tensor operations using GPUs and CPUs. The Tensor is a PyTorch class that is responsible for storing and operating on homogenous multidimensional rectangular arrays of numbers. PyTorch is used by engineers all around the world for prototyping to deploying actual production-grade systems. It has a widespread community with numerous tutorials, examples, and study materials.

Jupyter Notebook is an open-source web application that allows to create and share documents that contain live code, visualizations, markdown text, and numerous other options. It is a very popular tool among data scientists and machine learning practitioners for sharing workflows and collaborating.

While the GPT models were trained on a single Jupyter notebook using standard PyTorch and several standard libraries (e.g., Pandas, Matplotlib, etc.), the T5 models are trained through pre-made Python scripts and tools included in the official PICARD[10] implementation[29]. The reason for that is that the particular PICARD tool, an 'incremental parsing algorithm that integrates with ordinary beam search'[29], is very hard to implement into the model generating program.

The PICARD tool is written in C++ and is not available as a Python library. Therefore, the authors of the PICARD paper[10] provided a Python wrapper for the tool. However, the wrapper is not compatible with the latest version of the T5 library. Therefore, the PICARD tool was used as a standalone program, and the T5 model was fine-tuned using the scripts provided by the authors of the PICARD paper[10].

At last, the Hugging Face library was used as an interface for large pre-trained Transformer models (GPT-2, T5 in our case). It allows simple downloading/uploading, fine-tuning, and inference of the models.

3.0.4 Detailed explanation of implementation

In this section, we will describe the implementation of the models in detail. First, we will start with introducing the structure of the dataset used for training and evaluation. Then, we will explore the data preprocessing steps used for training the model. Finally, we will describe the process of fine-tuning the models and running evaluation.

3.0.4.1 Data preprocessing

Table 3.1 describes the structure of the Spider dataset questions. Every question has a corresponding database ID, SQL query and a natural language question. The database ID references a database that is contained inside the Spider dataset tables file described in table 3.2.

Spider dataset questions		
Attribute name	Attribute definition	Attribute example
db_id	ID of database for query	department_management
query	SQL query	SELECT count(*) FROM singer
query_toks	List of query tokens	[SELECT, count, (, *,), FROM, singer]
question	Natural language question	How many singers are there?
question_toks	List of NL question tokens	[How, many, singers, are, there, ?]

Table 3.1 Structure of the Spider dataset questions

Spider dataset tables		
Attribute name	Attribute definition	Attribute example
column_names	List of normalized column names	[perpetrator id, country, year]
column_names_original	List of column names	[Perpetrator_ID, Country, Year]
column_types	List of column types	[number, text, number]
db_id	ID of database	perpetrator
foreign_keys	List of foreign keys (index of column)	[1]
primary_keys	List of primary keys (index of column)	[1]
table_names	List of normalized table names	[perpetrator]
table_names_original	List of table names	[perpetrator]

Table 3.2 Structure of the Spider dataset tables

The GPT and T5 models used in this work each have slightly different format requirements for the input data. Therefore, we will split the data preprocessing into three categories: common, GPT, and T5.

Common preprocessing

The common preprocessing steps are applied to both GPT and T5 models. Because both models are LLMs, the input data has to be purely textual. This makes it difficult for encoding the database schema and feeding the model information about the structure and relations of tables and columns. After exploring relevant works, we decided to use the same database encoding schema as Shaw et al.(2021)[30]. In this work, the database schema is serialized and encoded together with the question. This allows the model to generalize to unseen databases by providing the new database schema inside the prompt itself during inference. Here is an example for the `climbing` database with schema defined as `Create Table` statements and corresponding serialized string:

```
CREATE TABLE "mountain" (
"Mountain_ID" int,
"Name" text,
"Height" real,
"Prominence" real,
"Range" text,
"Country" text,
PRIMARY KEY ("Mountain_ID")
);

CREATE TABLE "climber" (
"Climber_ID" int,
"Name" text,
"Country" text,
"Time" text,
"Points" real,
"Mountain_ID" int,
PRIMARY KEY ("Climber_ID"),
FOREIGN KEY ("Mountain_ID") REFERENCES "mountain"("Mountain_ID")
);
```

```
| climbing | mountain: Mountain_ID, Name, Height, Prominence, Range, Country
| climber: Climber_ID, Name, Country, Time, Points, Mountain_ID
```

After the schema is serialized, it gets appended to the natural language question and tokenized. The tokenization is handled by the Hugging Face tokenizer and transforms the string representation of the question into a vector of integers. The tokenized input is then fed to the model.

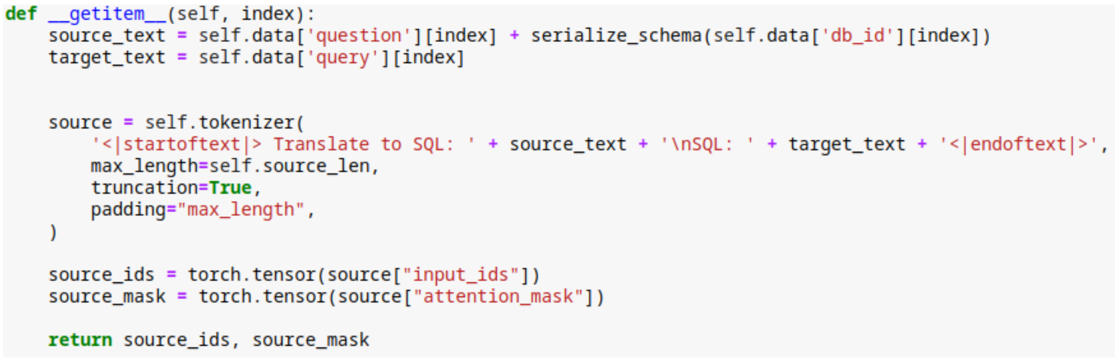
T5

The T5 model is trained using teacher forcing. That means that for training, the model always needs and input sequence and a corresponding target sequence. The input sequence usually begins with a prefix that specifies a given task (e.g., Translate to sql:). Since we fine-tune the model only for text-to-SQL task, this is not needed. Therefore the input sequence is the tokenized natural language question together with the serialized schema. The target sequence is just the corresponding SQL query.

GPT-2

GPT-2 is a language model that is trained to predict the next token in a sequence. Therefore, the data format is a bit different than T5. There is no target sequence, the model requires a single sequence that is fed into the model, and then it continues to generate and append to the output until an end-of-sequence token is generated. Because of this, we need to logically separate the prompt in order for the model to understand what the question is and where to fill the given SQL query. We are using the special separator token `<|sep|>` as shown in the GPT-2 paper[13]. Here is an example of the input data (before tokenization):

```
<|startoftext|>How many climbers are from the United States? | climbing |
mountain: Mountain_ID, Name, Height, Prominence, Range, Country |
climber: Climber_ID, Name, Country, Time, Points, Mountain_ID
SQL:<|sep|> SELECT count(*) FROM climber WHERE Country = 'United States'<|endoftext|>
```



```python
def __getitem__(self, index):
    source_text = self.data['question'][index] + serialize_schema(self.data['db_id'][index])
    target_text = self.data['query'][index]


    source = self.tokenizer(
        '<|startoftext|> Translate to SQL: ' + source_text + '\nSQL: ' + target_text + '<|endoftext|>',
        max_length=self.source_len,
        truncation=True,
        padding="max_length",
    )

    source_ids = torch.tensor(source["input_ids"])
    source_mask = torch.tensor(source["attention_mask"])

    return source_ids, source_mask
```

Figure 3.5 The __getitem__ method preprocessing input sequence. Image is original work.

3.0.4.2 Model fine-tuning

The fine-tuning process is very different in both models, therefore we will describe them separately.

GPT-2

The GPT-2 model is fine-tuned using the Hugging Face Transformers library[31]. Here is the program control flow for the fine-tuning process:

1. **Load the libraries and Spider dataset** First, we import the libraries needed for our fine-tuning process. We are using the following libraries:

   - `transformers` - Hugging Face Transformers library
   - `torch` - PyTorch library
   - `pandas` - Data manipulation library
   - `numpy` - Numerical computing library
   - Other standard Python libraries (e.g. `os`, `random`, etc.)

   Then we load the Spider dataset using the `pandas` library. The dataset is loaded from a JSON file and then converted to a `pandas` DataFrame.

2. **Preprocess and tokenize the data** We create helper functions that load our Spider tables file and serialize the database schema for given `db_id`. After that, we create a `SpiderDataset` class extending the PyTorch `Dataset` class. This class is used for loading and tokenizing the items of the dataset through a DataLoader straight into the model. Here is the most important method `__getitem__`.

3. **Fine-tune the model** After the data is loaded and tokenized, we can start the fine-tuning process. We are using the `GPT2LMHeadModel` class from the `transformers` library. This class is a PyTorch module that is a GPT-2 model with a language modeling head on top. The training process is a simple loop:

   a. **Forward pass** - The input sequence is fed into the model and the output is generated.
   b. **Calculate loss** - The loss is calculated using the `CrossEntropyLoss` function from the `torch` library.
   c. **Backward pass** - The gradients are calculated using the `backward` method of the `loss` variable.

   d. **Update parameters -** The parameters are updated using the `step` method of the `optimizer` variable.

4. **Evaluate the model** After the fine-tuning process is completed, we evaluate the model on the Spider dev set. The evaluation is done using the Spider evaluation script[22].

T5

The T5 model is fine-tuned using the official PICARD repository[29]. The repository contains the necessary scripts for fine-tuning, evaluating, and serving the model. It uses the Hugging Face Transformers library[31], PyTorch, and other standard Python libraries. Everything is executed inside Docker containers for better encapsulation and replication. The repository is also using git submodules because of external tools that are needed mainly for the PICARD algorithm. The repository can be set up in this way:

```
git clone git@github.com:ElementAI/picard.git
cd picard
git submodule update --init --recursive
```

Most of the scripts required for training are located inside the `seq2seq` directory. Because we are using docker, running the training script is as simple as running the following command: `make train`.

However, what we need to do before running the training is to prepare the configuration file. Configuration files are located inside the `configs` directory and are written in JSON format. The configuration file contains all the hyperparameters of the model, the number of epochs to train for, and other arguments (e.g., cache directory, model path, seed, etc.).

Chapter 4

Experiments and results

In this chapter, we will describe the experiments that we conducted and the results that we obtained. First, we will introduce the questions that we want to answer with our experiments. Then, we will describe the experimental setup and the results of our experiments. Finally, we will discuss the results and compare them with previous research.

4.1 Research questions

- Question 1: Is the LLM-based approach better than previous approaches using RNNs?
- Question 2: Is having an encoder and decoder architecture better than having only a decoder architecture?
- Question 3: Are smaller models running locally on a single GPU capable of achieving good results?

4.2 Experimental setup

This section describes the experimental setup of our experiments. We will describe the models that we used, including their hyperparameters, the number of epochs the models were trained for, batch size, and other important information.

All models were trained on the Spider dataset train split and evaluated on the Spider dataset dev split. Additionally, the T5 models were also evaluated on the Spider variants Spider-DK and Spider Realistic, thanks to the performance displayed by the models. Batch size varies model-to-model because of the different number of hyperparameters. Some models are too large to have a batch size larger than one. For that case, we used a technique called ”Gradient Accumulation”, a method to overcome cases where there is not enough space on the GPU to accommodate more items into the batch. Gradient accumulation means running a configured number of steps in the training loop without updating the model variables while accumulating gradients of the mini-batches. After the number of steps is reached, the `optimizer.step()` method is called that propagates the errors through the network and updates the variables of the model. This allows us to have an effective batch size of `gradient_accumulation_steps * batch_size`.

We have two different learning rates, one for the GPT-2 models and one for the T5 models. The learning rates are recommended by the authors of the models and hyperparameter tuning was not performed because of the computational complexity of the task. We used a linear scheduler with a warmup for the learning rate, which means that the learning rate gradually increased

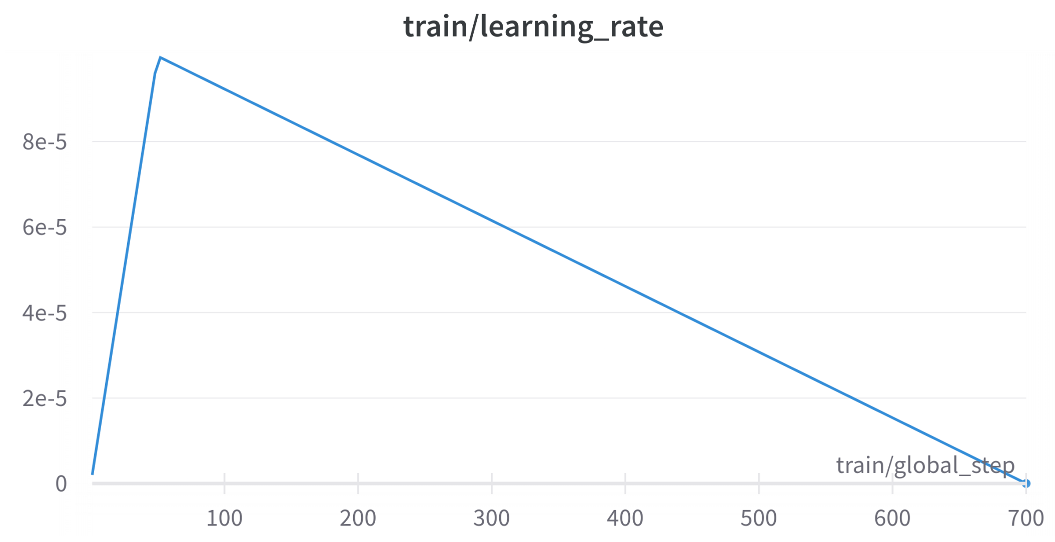


Figure 4.1 Illustration of the linear scheduler for learning rate. Image is original.

from 0 to the defined learning rate in the first `n_warmup_steps` and then linearly decayed to 0 over the whole training process.

The optimizers used for training are the AdamW[32] optimizer and Adafactor[33] optimizer. The AdamW optimizer is a variant of the Adam optimizer that uses weight decay regularization. The Adafactor optimizer is a variant of the Adam optimizer that uses a different method for calculating the learning rate. Both are based on the original Adam[34] optimizer, which is a method of stochastic optimization for training neural networks. Adam works by adaptively adjusting the learning rate for each parameter during training by incorporating the first and second moments of the gradients. It is widely adopted among deep learning practitioners and is often used as the default choice for training neural networks.

The T5 models were trained using Adafactor optimizer as recommended by the authors of the model. The GPT-2 models, except for the largest variant, GPT-2 XL, were trained using the AdamW optimizer. The GPT-2 XL model was trained using the Adafactor optimizer because of using less memory, and the model was too large to be trained on a single GPU otherwise.

All models except the two largest (GPT-2 XL and T5 3B) were trained on a local machine with a single NVIDIA GeForce RTX 3090 GPU. The two largest models were trained on a single NVIDIA RTX A6000 GPU running in the Vast.AI Cloud Computing platform. The reason for using the cloud computing platform is that the models could not fit into 24 GB of VRAM unless doing optimizations that hurt the performance of the models.

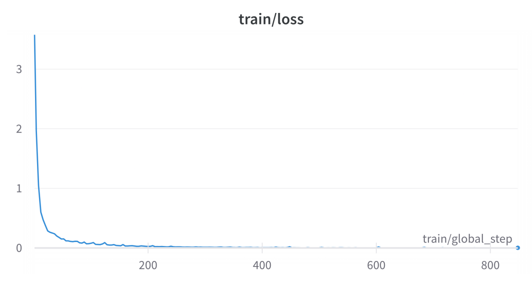
Model	# Parameters	Effective Batch Size	Learning Rate	# Epochs
T5-Base	220M	32	1e-4	50
T5-Large	770M	64	1e-4	50
T5-3B	3B	128	1e-4	20
GPT-2 Small	124M	32	5e-4	50
GPT-2 Large	774M	64	2e-4	50
GPT-2 XL	1.5B	128	1e-4	20

Table 4.1 Summary of model parameters and training settings

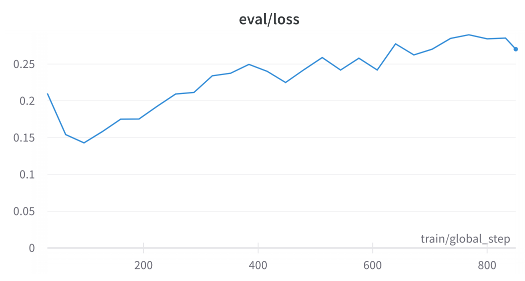
4.3 Fine-tuning insights

The following section provides insights into the process of fine-tuning the models. We will the describe the evolution of key model metrics throughout the training process. We will also delve into the attention mechanism and analyze the attention weights learned throughout training to allow a better interpretation of our model. We chose our T5-3B model since it is the best model of this work, as will shown in the next section.

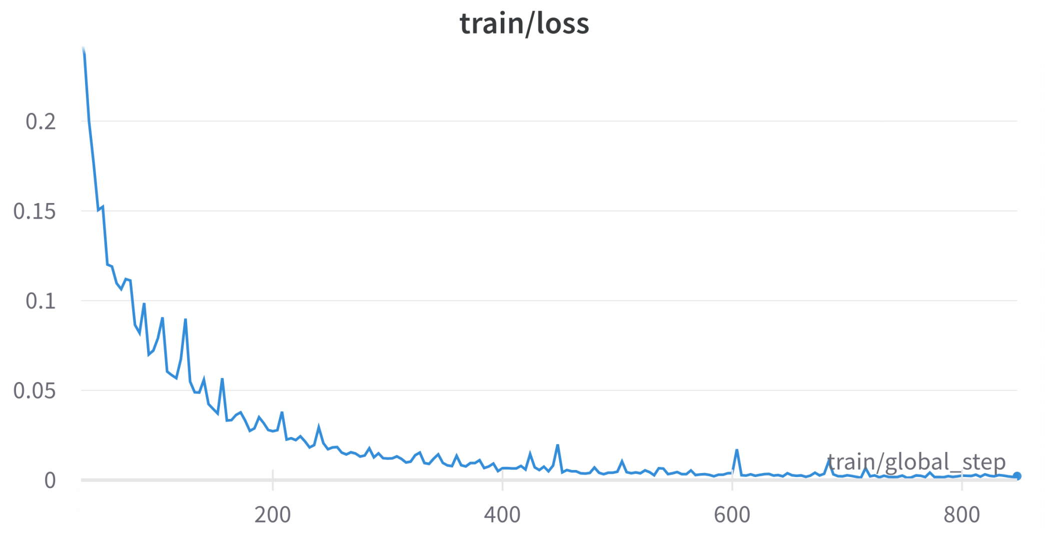
4.3.1 Evolution of key model metrics



(a) Training Cross Entropy Loss



(b) Evaluation Cross Entropy Loss

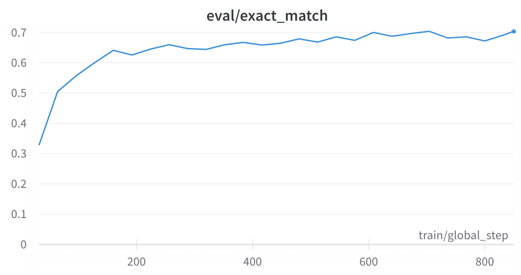


(c) Training Cross Entropy Loss (After 1st epoch)

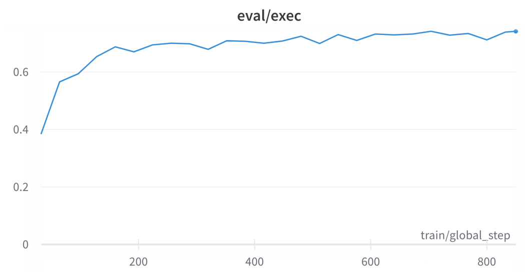
Figure 4.2 Cross Entropy Loss during Training

Figure 4.2 shows the evolution of the cross entropy loss during training and evaluation. The training loss is evaluated per batch, and the evaluation loss is evaluated per epoch. Despite looking like the training loss starts converging at the 200th step, if we ignore the first few steps with extremely high loss, we can see that the training loss is decreasing nicely across the whole training process. However, the evaluation loss decreases at the start, then begins to increase as the training continues. Intuitively, this would suggest that the model is overfitting on the training data, and the generalization to the validation data is getting worse. The opposite is the truth, as we will see in figure 4.3, which shows the official Spider metrics throughout the training.

The reason for that is that the cross entropy loss does not take into account the structure and semantics of the query.



(a) Exact Match Accuracy

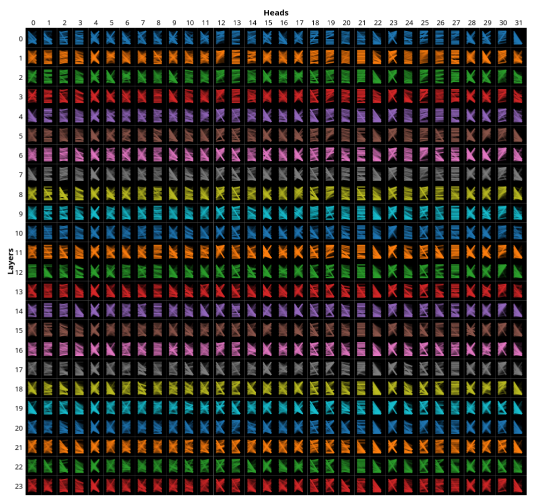


(b) Execution Accuracy

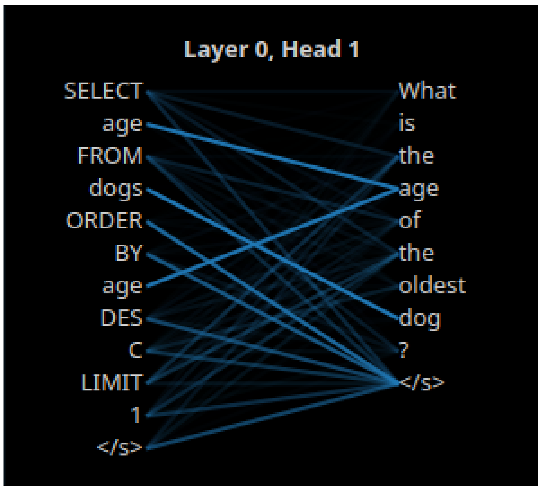
Figure 4.3 Spider Metrics during Training

Figure 4.3 shows us the evolution of Spider metrics on validation data during training. We can see that both the exact match accuracy and execution accuracy are increasing throughout the training process. This is a good sign that the model is learning to generalize well on the validation data. The evaluation during training is done without PICARD for the sake of the training speed because the PICARD algorithm increases the evaluation time by a large margin.

4.3.2 Attention analysis



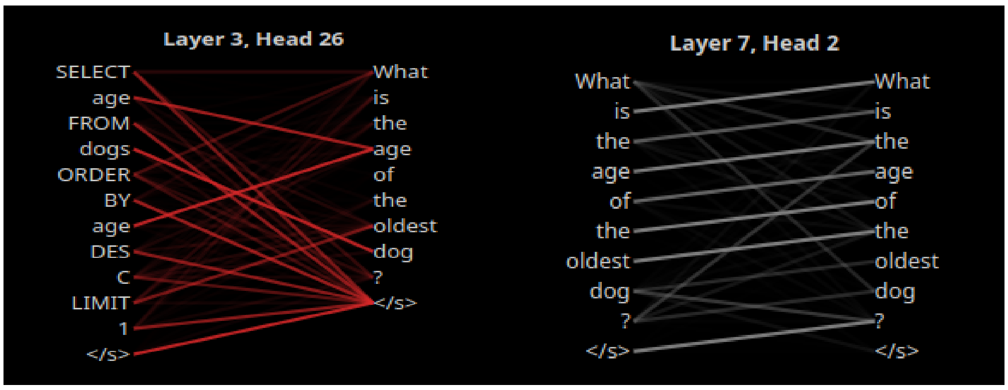
(a) Attention heads of the first 24 layers of the encoderlayer



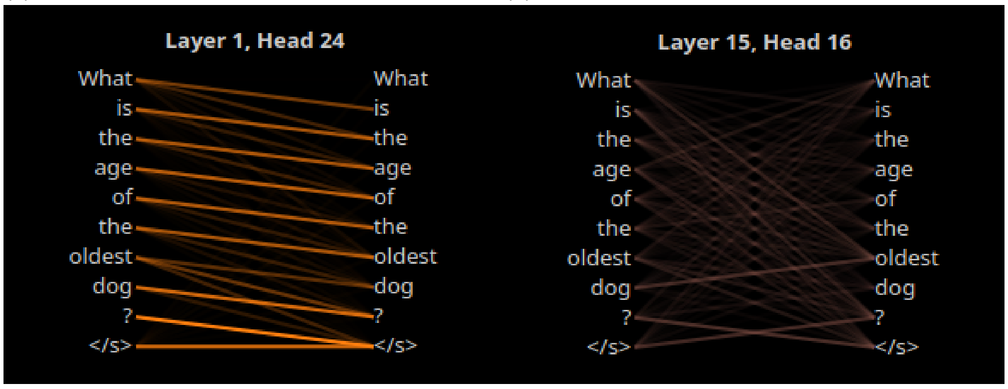
(b) Single attention head of encoder-decoder attention

Figure 4.4 Visualization of attention heads and weights

In figure 4.4, we can see all 32 attention heads per layer for the first 24 layers. We can observe that the attention heads are learning to focus on different parts of the input. Notably, in figure 4.4b, we can see the visualization of a single attention head in layer 0 of the encoder-decoder attention layer (cross-attention layer) for an example input and target sequence pair. This showcases what tokens from the input the model pays attention to for a given token from the target sequence. For example, we can see that there is a clear relation between the ”age” token from the input and the ”age” column in both the SELECT and ORDER BY clauses. We will delve deeper into the attention heads and different patterns for the same example sequence.



(a) Attention head of the encoder-decoder attention **(b)** Attention head of the encoder attention



(c) Attention head of the encoder attention **(d)** Attention head of the encoder attention

Figure 4.5 Visualization of single attention heads

Figure 4.5 contains 4 different common patterns of the attention heads.

Attention to identical/related tokens

The first pattern is the attention to identical/related tokens. In figure 4.5a, we can see that the tokens of the target sequence (the SQL query) are paying attention to related words in the input sequence (NL question). One example was the ”age” token described above, other examples can be token ”dogs” to ”dog” or ”LIMIT” to ”oldest” since querying the oldest implies limiting the SQL result to one row.

Attention to previous token

The second pattern is the attention to the previous token. In figure 4.5b, we can see that the tokens pay the most attention to the previous token. This is a common pattern in the encoder attention layers since the encoder is learning the sequential ordering of the tokens. This is loosely related to sequential forward RNNs.

Attention to the next token

The third pattern is the attention to the next token. In figure 4.5c, we can see that the tokens pay the most attention to the next token. This is almost the same principle as the previous pattern but based on backward RNNs, where states are updated from left to right.

”Bag of words” pattern

The fourth pattern is the ”bag of words” pattern. In figure 4.5d, we can see that the attention is divided relatively evenly across all words in the sentence. This is essentially computing a bag-of-words embedding by taking the (almost) unweighted average of the word embeddings.

4.4 Analysis of experimental results

In this section, we will analyze the results of our experiments. Then, we will discuss the performance of different models. Finally, we will compare our results with previous research.

4.4.1 Discussion of the performance of different models

Model Name	No. of Parameters	Exact Match (%)	Execution Accuracy (%)
T5-Base	220M	55.57%	56.29
T5-Large	770M	61.03%	64.9%
T5-3B	3B	70.41%	73.89%
T5-3B + PICARD	3B	**74.18%**	**78.72%**
GPT-2 Small	124M	21.5%	22.8%
GPT-2 Large	774M	31.3%	33.6%
GPT-2 XL	1.5B	42.4%	43.6%

Table 4.2 Model parameters, exact match, and execution accuracy on the Spider dev dataset

The performance of various models was evaluated on the Spider dataset dev split by using the official Spider evaluation script[22]. As shown in the table above, the performance of the models differs a lot. This can be attributed to several factors, including the number of parameters and the architecture of the model.

Starting with the T5 model, we can observe that the performance in both exact match and execution accuracy is improving significantly as the number of parameters increases. The 3B parameter model achieves an exact match score of 70.41% and an execution accuracy of 73.89%. However, the T5 model with PICARD, also with 3B parameters, outperforms all other models, reaching an exact match score of 74.18% and an execution accuracy of 78.72%. This suggests that processes outside of the training process of the model, such as constraining the output of the decoder with incremental SQL parser, may be beneficial for the text-to-SQL task.

On the other hand, GPT-2 models, despite their large number of parameters, perform significantly worse than the T5 models. The largest GPT-2 model with 1.5B parameters only achieved an exact match score of 42.4% and an execution accuracy of 43.6%. We also evaluated our 2 best models on the Spider dataset variants. We will analyze the results of the tables below.

Model Name	No. of Parameters	Exact Match (%)	Execution Accuracy (%)
T5-Large	770M	50.79%	52.95%
T5-3B	3B	61.42%	64.17%

Table 4.3 Model parameters, exact match, and execution accuracy on the Spider Realistic dataset

We can see that while the performance of the models on the Spider realistic dataset is worse only by about 10%, the performance on the Spider-DK dataset is significantly worse. This suggests that meanwhile our models can adapt to the question not matching the column names explicitly in the question and being more indirect, handling domain knowledge is very hard for these models.

Model Name	No. of Parameters	Exact Match (%)	Execution Accuracy (%)
T5-Large	770M	39.81%	50.28%
T5-3B	3B	47.1%	56.26%

Table 4.4 Model parameters, exact match, and execution accuracy on the Spider-DK dataset

We will look at some of the generated queries where the models went wrong and try to address some of the issues.

Analysis of generated queries

For the sake of conciseness, we will only look at the generated queries of the T5-3B model and the GPT-2 XL model. We will also omit the serialized schema, which is in every prompt, and only look at the question and query pairs.

Question: What is the age of the oldest dog?

```
-- T5-3B
SELECT age FROM dogs ORDER BY age DESC LIMIT 1
```

```
-- GPT-2 XL
SELECT age FROM dogs ORDER BY age DESC LIMIT 1
```

```
-- Gold
SELECT max(age) FROM Dogs
```

Here is the first type of issue, where the model generates a query that is semantically equivalent to the gold query but is not exactly the same. This can be seen as the difference between exact match and execution accuracies, which the Spider evaluation script takes into account.

Question: What is the name and country of origin of every singer who has a song with the word 'Hey' in its title?

```
-- T5-3B
SELECT t1.Name, t1.Country
FROM singer AS t1
JOIN singer_in_concert AS t2 ON t1.Singer_id = t2.Singer_id
JOIN song AS t3 on t2.Song_id = t3.Song_id
WHERE t3.Song_name LIKE '%Hey%'
```

```
-- GPT-2 XL
SELECT T2.Name, T2.Country
FROM singer_in_concert AS T1
JOIN song AS T2 ON T1.Song_ID = T2.Song_ID
WHERE T2.Song_Name LIKE '%Hey%'
```

```
-- Gold
SELECT name , country
FROM singer
WHERE song_name LIKE '%Hey%'
```

Another type of issue arises when the question is unclear, and synonyms are used. In this case, the question is too wordy, leading the model to assume the query is more complex and the joining of tables is needed. Additionally, the question specifically does not use the words ”song name”, but uses a synonym "title". Both models assume that `title` is included inside the `song` table, leading to the joins. However, the models correctly generate the `song_name` column in the end. We experimented with changing the structure of the question to better accommodate the models:

Question: What is the name and country for all singers whose song name contains the word 'Hey'?

```
-- T5-3B
SELECT name , country
FROM singer
WHERE song_name LIKE '%Hey%'
```

```
-- GPT-2 XL
SELECT name , country
FROM singer
WHERE song_name LIKE '%Hey%'
```

```
-- Gold
SELECT name , country
FROM singer
WHERE song_name LIKE '%Hey%'
```

As we can see, the models now correctly generate the SQL query. This suggests that the models are sensitive to the structure of the question. Also, the models lack domain knowledge to correctly interpret the synonyms.

Question: For the cars with 4 cylinders, which model has the largest horsepower?

```
-- T5-3B
SELECT t2.Model
FROM cars_data AS t1
JOIN model_list AS t2 ON t1.id = t2.Modelid
WHERE t1.Cylinders = 4
ORDER BY t1.horsepower DESC
LIMIT 1
```

```
-- GPT-2 XL
SELECT model
FROM cars
WHERE cylinders = 4
ORDER BY horsepower DESC
LIMIT 1
```

```
-- GOLD
SELECT T1.Model FROM CAR_NAMES AS T1
JOIN CARS_DATA AS T2 ON T1.MakeId = T2.Id
WHERE T2.Cylinders = 4
ORDER BY T2.horsepower DESC
LIMIT 1
```

An opposite issue of the question being too wordy is when the question is too short. In this case, the wanted query is complex. Meanwhile, the question is very concise and includes only the basic information. Therefore, the model is required to make assumptions based solely on the serialized schema in the prompt. The GPT-2 XL makes too simple assumptions that all information is in the `cars` table. Meanwhile, the semantics of the query are correct, it does not match the structure of the database because of missing information. The T5-3B model handled this problem marginally better, correctly joining two tables and selecting relevant columns. The lack of information about the tables causes the model to assume the wrong table `model_list`, which differs from the gold query, but it is correct. Table `model_list` contains the `Model` column with the same information as the `car_names` table, and the query retrieves the same data from the database. To make GPT-2 understand the requirements for the query, we made the following changes:

Question: Give me the model from car names, with 4 cylinders inside car data, that has the largest horsepower.

```
-- GPT-2 XL
SELECT T2.model
FROM cars_data AS T1
JOIN model_list AS T2 ON T1.model = T2.model_id
WHERE T1.cylinders = 4
ORDER BY T1.horsepower DESC
LIMIT 4
```

```
-- GOLD
SELECT T1.Model
FROM CAR_NAMES AS T1
JOIN CARS_DATA AS T2 ON T1.MakeId = T2.Id
WHERE T2.Cylinders = 4
ORDER BY T2.horsepower DESC
LIMIT 1
```

Even when enriching the question with table names that are relevant to the given column, the model was still not able to generate a correct query. This suggests that the model is not able to understand complex database schemas and make correct assumptions about the tables and columns. This heavily impacts the model's performance on larger databases, which are predominant in practice.

4.4.2 Comparison of results with previous research

This section will be dedicated into comparing models from our experiments with previous research. Since the Spider test dataset is not publicly available, we will compare our results on the dev split.

Model Name (Reference)	**Exact Match (%)**	**Execution Accuracy (%)**
Graphix-3B + PICARD[20, p. 2023]	**77.1**	**81.0**
DIN-SQL + GPT-4 (w/o fine-tuning) [35, p. 2023]	60.1	74.2
BRIDGE v2 + BERT[36, p. 2020]	71.1	-
RATSQL + GAP [37, p. 2021]	71.8	-
RATSQL + BERT [37, p. 2021]	69.7	-
SQLNet [38, p. 2017]	10.9	18.4
T5-3B (ours)	70.41	73.89
T5-3B + PICARD (ours)	74.18	78.72
GPT-2 XL (ours)	42.4	43.6

Table 4.5 Comparison of our results with previous research on Spider dev split.

Table 4.5 describes the results on the Spider dev dataset. The SOTA model (not including the test set) is the Graphix-3B + PICARD[20] achieving 81% execution accuracy and 77.1%

exact match. The model is based on the T5-3B model by enhancing the individual Transformer layers with graph-aware layers, as described in Chapter 2. This model is performing better than our best model T5-3B + PICARD, only by a slight margin.

Surprisingly, the third best model is the extremely large GPT-4 model using a technique called "Decomposed In-Context Learning with Self-Correction"[35], which does not require fine-tuning of the model and examples of queries and their answers are embedded inside the prompt itself. It also uses self-correction, a method that sends the generated query back to the model and instructs it to check for any potential issues with a specially engineered prompt. This allows the model "to learn" without undergoing the training process, and it is a very powerful technique.

Finally, we can see that SQLNet, a model which is based on LSTM encoder-decoder architecture, is heavily outperformed by all Transformer-based models. This showcases the power of the Transformer architecture and its ability to learn complex relationships.

Chapter 5

Conclusion and future work

Throughout this work, we explored the exciting and challenging field of NLP, explicitly focusing on the task of converting natural language to SQL queries. Our exploration took us through the history and evolution of the fields, the advancements brought by the introduction of neural networks and deep learning, and especially the introduction of the Transformer architecture.

We focused on the Transformer architecture and its types, specifically encoder-decoder (T5) and decoder-only (GPT-2) architectures. We experimented with both and evaluated the results on the Spider dataset, which is a comprehensive and complex benchmark that tests the ability of the models to handle complex database schemas, generalize to unseen schemas, and correctly understand the semantics of the natural language question.

5.0.1 Summary of key findings

Despite the challenges, we observed the remarkable performance of the models on the Spider dataset. The best model T5-3B + PICARD, achieved 74.18% exact match and 78.72% execution accuracy. This is an outstanding result, especially considering the complexity of the task. The models demonstrated a solid understanding of SQL query generation, although with some limitations, particularly on handling complex databases with multiple tables and tasks requiring specific domain knowledge to correctly infer the intent. We also noticed that the models were sensitive to the structure of the input question, where minor modifications in question phrasing led to significant differences in the output SQL query.

After inspecting the performance of the GPT-2 and T5 models, we concluded that the text-to-SQL task gains excellent benefits from the encoder thanks to its ability to capture the semantics of the input question. The differences were evident in schema linking, which means mapping relevant parts of the question to the columns and tables included in the prompt. GPT-2 model showed on numerous occasions to generate SQL queries with non-existing columns or tables, whereas the T5 model handled that significantly better. Therefore, the encoder-decoder architecture seems better suited for this task.

5.0.2 Limitations of the study and future directions

The main limitation of this thesis is the lack of computing resources. We found that training models larger than 3 billion parameters are not feasible on a single GPU without resorting to specific and complex techniques, which, however, slow down the speed of training (off-loading model to RAM) or degrade the performance of the models (reducing the size of the weights from 32 bits to 16 or less).

Looking forward, overcoming this constraint and exploring larger models is a promising direction for future research. Advancements in computational capabilities and the development of more efficient training techniques will allow future researchers to explore larger and more complex models, which will further push the performance in natural language to SQL translation.

Another promising direction for future research lies in the development of methods independent of the training process. For example, self-correction techniques, where the model is fed back the generated SQL query and instructed to look for potential mistakes. Prompt engineering techniques could be used to optimize the way natural language queries are posed to the model, maximizing the probability of generating correct SQL queries.

Finally, new techniques for encoding the database schema into the prompt could be explored, and on the other side, new techniques for decoding the SQL query from the model could be developed, such as generating an AST tree instead of text.

5.0.3 Potential applications and impact of LLM-based inference applications

LLMs have shown remarkable success in a wide range of NLP tasks, including the translation of natural language queries into SQL. This ability can be leveraged to develop robust applications that significantly enhance human-computer interaction.

One such application is the creation of user-friendly interfaces for database management systems. These interfaces can use LLMs to translate natural language queries from non-technical users into SQL, thus enabling a more comprehensive range of individuals to create data-driven insights without the need to learn SQL.

Another promising application is the field of data analytics and reporting. LLMs can be used to automate the generation of SQL queries for a wide range of reporting tasks, such as generating daily reports. This will allow all businesses to create exciting insights and reports from their data without the need to study computer science or hire data analysts.

Appendix A
Attachments

```
 readme.txt ................ brief description of attached media
 src
   impl .................... source codes of the implementation
   configs ................. configuration files for T5 model tranining
   thesis .................. source code of the thesis in LaTeX
 text ...................... text of the thesis
   thesis.pdf .............. text of the thesis in PDF format
```

Bibliography

1. CHIANG, Wei-Lin; LI, Zhuohan; LIN, Zi; SHENG, Ying; WU, Zhanghao; ZHANG, Hao; ZHENG, Lianmin; ZHUANG, Siyuan; ZHUANG, Yonghao; GONZALEZ, Joseph E.; STOICA, Ion; XING, Eric P. *Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90%* ChatGPT Quality*. 2023. Available also from: `https://lmsys.org/blog/2023-03-30-vicuna/`.
2. NADKARNI, Prakash M; OHNO-MACHADO, Lucila; CHAPMAN, Wendy W. Natural language processing: an introduction. *Journal of the American Medical Informatics Association*. 2011, vol. 18, no. 5, pp. 544–551. ISSN 1067-5027. Available from DOI: `10.1136/amiajnl-2011-000464`.
3. WIKIPEDIA CONTRIBUTORS. *Word embedding — Wikipedia, The Free Encyclopedia*. 2023. Available also from: `https://en.wikipedia.org/w/index.php?title=Word_embedding&oldid=1146771404`. [Online; accessed 23-April-2023].
4. GAUTAM, Hariom. Word Embedding: Basics - Hariom Gautam. *Medium*. 2020. Available also from: `https://medium.com/@hari4om/word-embedding-d816f643140`.
5. KAMATH, Uday; LIU, John; WHITAKER, James. *Deep Learning for NLP and Speech Recognition*. [N.d.]. ISBN 978-3-030-14596-5. Available also from: `https://link.springer.com/book/10.1007/978-3-030-14596-5`.
6. ÖZBAY, Erdal. USE OF DEEP LEARNING MODELS IN DIFFERENT APPLICATION AREAS. In: 2022, pp. 123–155. ISBN 978-625-8108-87-3.
7. ABBOU, R. DeepClassic: Music Generation with Neural Neural Networks. In: 2020.
8. [N.d.]. Available also from: `https://erdem.pl/2021/05/introduction-to-attention-mechanism`.
9. VASWANI, Ashish; SHAZEER, Noam; PARMAR, Niki; USZKOREIT, Jakob; JONES, Llion; GOMEZ, Aidan N.; KAISER, Lukasz; POLOSUKHIN, Illia. Attention Is All You Need. *CoRR*. 2017, vol. abs/1706.03762. Available from arXiv: `1706.03762`.
10. SCHOLAK, Torsten; SCHUCHER, Nathan; BAHDANAU, Dzmitry. *PICARD: Parsing Incrementally for Constrained Auto-Regressive Decoding from Language Models*. 2021. Available from arXiv: `2109.05093 [cs.CL]`.
11. BENGIO, Yoshua; GOODFELLOW, Ian; COURVILLE, Aaron. *Deep learning*. Vol. 1. MIT press Cambridge, MA, USA, 2017.
12. RADFORD, Alec; NARASIMHAN, Karthik. Improving Language Understanding by Generative Pre-Training. In: 2018.
13. RADFORD, Alec; WU, Jeff; CHILD, Rewon; LUAN, David; AMODEI, Dario; SUTSKEVER, Ilya. Language Models are Unsupervised Multitask Learners. In: 2019.

14. DEVLIN, Jacob; CHANG, Ming-Wei; LEE, Kenton; TOUTANOVA, Kristina. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.* 2019. Available from arXiv: `1810.04805 [cs.CL]`.

15. RAFFEL, Colin; SHAZEER, Noam; ROBERTS, Adam; LEE, Katherine; NARANG, Sharan; MATENA, Michael; ZHOU, Yanqi; LI, Wei; LIU, Peter J. *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer.* 2020. Available from arXiv: `1910.10683 [cs.LG]`.

16. KIM, Hyeonji; SO, Byeong-Hoon; HAN, Wook-Shin; LEE, Hongrae. Natural Language to SQL: Where Are We Today? *Proc. VLDB Endow.* 2020, vol. 13, no. 10, pp. 1737–1750. ISSN 2150-8097. Available from DOI: `10.14778/3401960.3401970`.

17. ZHONG, Victor; XIONG, Caiming; SOCHER, Richard. *Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning.* 2017. Available from arXiv: `1709.00103 [cs.CL]`.

18. KATSOGIANNIS-MEIMARAKIS; KOUTRIKA. A survey on deep learning approaches for text-to-SQL. *The VLDB Journal.* 2023, pp. 1–32. Available from DOI: `10.1007/s00778-022-00776-8`.

19. WANG, Bailin; SHIN, Richard; LIU, Xiaodong; POLOZOV, Oleksandr; RICHARDSON, Matthew. *RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers.* 2021. Available from arXiv: `1911.04942 [cs.CL]`.

20. LI, Jinyang; HUI, Binyuan; CHENG, Reynold; QIN, Bowen; MA, Chenhao; HUO, Nan; HUANG, Fei; DU, Wenyu; SI, Luo; LI, Yongbin. *Graphix-T5: Mixing Pre-Trained Transformers with Graph-Aware Layers for Text-to-SQL Parsing.* 2023. Available from arXiv: `2301.07507 [cs.CL]`.

21. BUSBRIDGE, Dan; SHERBURN, Dane; CAVALLO, Pietro; HAMMERLA, Nils Y. *Relational Graph Attention Networks.* 2019. Available from arXiv: `1904.05811 [cs.LG]`.

22. YU, Tao; ZHANG, Rui; YANG, Kai; YASUNAGA, Michihiro; WANG, Dongxu; LI, Zifan; MA, James; LI, Irene; YAO, Qingning; ROMAN, Shanelle; ZHANG, Zilin; RADEV, Dragomir. *Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task.* 2019. Available from arXiv: `1809.08887 [cs.CL]`.

23. SHIVAKUMAR, Prashanth Gurunath; YANG, Mu; GEORGIOU, Panayiotis. Spoken language intent detection using confusion2vec. *arXiv preprint arXiv:1904.03576.* 2019.

24. ZELLE, John M.; MOONEY, Raymond J. Learning to Parse Database Queries Using Inductive Logic Programming. In: *Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 2.* Portland, Oregon: AAAI Press, 1996, pp. 1050–1055. AAAI'96. ISBN 026251091X.

25. DENG, Xiang; AWADALLAH, Ahmed Hassan; MEEK, Christopher; POLOZOV, Oleksandr; SUN, Huan; RICHARDSON, Matthew. Structure-Grounded Pretraining for Text-to-SQL. *CoRR.* 2020, vol. abs/2010.12773. Available from arXiv: `2010.12773`.

26. SHAW, Peter; CHANG, Ming-Wei; PASUPAT, Panupong; TOUTANOVA, Kristina. Compositional Generalization and Natural Language Variation: Can a Semantic Parsing Approach Handle Both? In: *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers).* Online: Association for Computational Linguistics, 2021, pp. 922–938. Available from DOI: `10.18653/v1/2021.acl-long.75`.

27. GAN, Yujian; CHEN, Xinyun; HUANG, Qiuping; PURVER, Matthew; WOODWARD, John R.; XIE, Jinxia; HUANG, Pengsheng. Towards Robustness of Text-to-SQL Models against Synonym Substitution. In: *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Online: Association for Computational Linguistics, 2021, pp. 2505–2515. Available from DOI: `10.18653/v1/2021.acl-long.195`.

28. GAN, Yujian; CHEN, Xinyun; PURVER, Matthew. *Exploring Underexplored Limitations of Cross-Domain Text-to-SQL Generalization*. 2021. Available from arXiv: `2109.05157 [cs.CL]`.

29. SCHOLAK, Torsten; SCHUCHER, Nathan; BAHDANAU, Dzmitry. PICARD: Parsing Incrementally for Constrained Auto-Regressive Decoding from Language Models. In: *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2021, pp. 9895–9901. Available also from: `https://aclanthology.org/2021.emnlp-main.779`.

30. SHAW, Peter; CHANG, Ming-Wei; PASUPAT, Panupong; TOUTANOVA, Kristina. Compositional Generalization and Natural Language Variation: Can a Semantic Parsing Approach Handle Both? In: *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Online: Association for Computational Linguistics, 2021, pp. 922–938. Available from DOI: `10.18653/v1/2021.acl-long.75`.

31. WOLF, Thomas; DEBUT, Lysandre; SANH, Victor; CHAUMOND, Julien; DELANGUE, Clement; MOI, Anthony; CISTAC, Perric; MA, Clara; JERNITE, Yacine; PLU, Julien; XU, Canwen; LE SCAO, Teven; GUGGER, Sylvain; DRAME, Mariama; LHOEST, Quentin; RUSH, Alexander M. Transformers: State-of-the-Art Natural Language Processing. In: Association for Computational Linguistics, 2020, pp. 38–45. Available also from: `https://www.aclweb.org/anthology/2020.emnlp-demos.6`.

32. LOSHCHILOV, Ilya; HUTTER, Frank. *Decoupled Weight Decay Regularization*. 2019. Available from arXiv: `1711.05101 [cs.LG]`.

33. SHAZEER, Noam; STERN, Mitchell. *Adafactor: Adaptive Learning Rates with Sublinear Memory Cost*. 2018. Available from arXiv: `1804.04235 [cs.LG]`.

34. KINGMA, Diederik P.; BA, Jimmy. *Adam: A Method for Stochastic Optimization*. 2017. Available from arXiv: `1412.6980 [cs.LG]`.

35. POURREZA, Mohammadreza; RAFIEI, Davood. *DIN-SQL: Decomposed In-Context Learning of Text-to-SQL with Self-Correction*. 2023. Available from arXiv: `2304.11015 [cs.CL]`.

36. LIN, Xi Victoria; SOCHER, Richard; XIONG, Caiming. *Bridging Textual and Tabular Data for Cross-Domain Text-to-SQL Semantic Parsing*. 2020. Available from arXiv: `2012.12627 [cs.CL]`.

37. SHI, Peng; NG, Patrick; WANG, Zhiguo; ZHU, Henghui; LI, Alexander Hanbo; WANG, Jun; SANTOS, Cicero Nogueira dos; XIANG, Bing. Learning contextual representations for semantic parsing with generation-augmented pre-training. In: *AAAI 2021*. 2021. Available also from: `https://www.amazon.science/publications/learning-contextual-representations-for-semantic-parsing-with-generation-augmented-pre-training`.

38. XU, Xiaojun; LIU, Chang; SONG, Dawn. *SQLNet: Generating Structured Queries From Natural Language Without Reinforcement Learning*. 2017. Available from arXiv: `1711.04436 [cs.CL]`.