SQL query generation from text

Bibi Hajira, Mahammada **Katia Oussar**

Abstract

With the availability of high-power computing machines, and the advancements in the fields of optimization, and deep learning, the field of Natural Language Processing has witnessed great developments in the last decade. Transformer, in particular, have been of significant importance due to their use of *attention*. With these developments many sequence-to-sequence tasks have seen improvements, such as in machine translation, abstraction, question and answering. In our work here, we try to employ the transformer(s) to *translate* a given English text to a corresponding SQL query. Our results show non-trivial performance of the model(s) and we try to show that having a pretrained model (trained on codes) could achieve better results with fine tuning, when compared to an untrained counterpart. Our code is made available in github [1].

1 Introduction

The field of Machine learning has had attracted lot of attention over the last decade with its near exponential developments. With the introduction of AlexNet by Alex Krizhensky et. al, with competitive results on ImageNet dataset, it was soon followed by other developments to support more and more complex models. Residual networks, introduced by Kaimin He et. al, solved the problem of exploding and vanishing gradients which was limiting the use of Neural networks for more complex tasks. Further advancements in optimization space such as Batch Normalization (Sergey Ioffe et. al.,), Adam Optimizer (Diederik P. Kingma et. al.,) helped increase the training speeds by better navigating the loss function landscape. These are but a few advancements which has made this field a success.

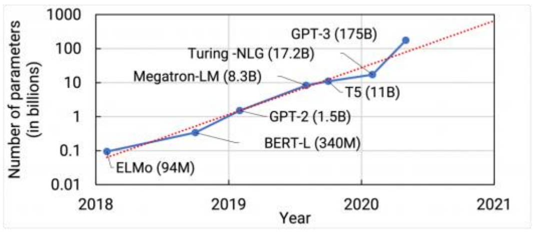

Figure 1: This image taken from Nvidia, shows how the models have evolved in recent years. This only shows models before 2021.

Natural Language, having a sequential aspect to it, had been using sequential networks such as Recurrent Neural Networks and its variants. One challenge was still with the quality of word embeddings as merely using static word vector which remains same regardless of the context was inherently limiting. Several other works came in which came up with techniques to better generate a word embedding, for example with surrounding context.

With the introduction of Transformers (Vaswani et. al.,) lot of the sequential work was eliminated as attention could be computed in parallel. With this attention the word embeddings now strongly leveraged the context of entire text.

Transfer learning with Transformers shown by BERT (Jacob Delvin et. al.,) was a major game changer, as it showed the power of transfer learning by beating many records in multiple tasks by finetuning a pre-trained model. This pre-trained model was trained easily with abundantly available

[1]https://github.com/BibiHajiraM/NLP_Final_Project

unlabeled text. More recent developments include even powerful models such as GPT, GPT3, which have demonstrated extraordinary performance.

One thing worth mentioning is the prohibitively expensive costs to train modern NLP models. With hundreds of GPUs running for months, the energy consumption is one problem, and the other is the inability of general public to ever be capable of developing an independent model themselves as it costs millions of dollars. Figure 1 tries to show such evolution in terms of model size [2].

Of the many sequence-to-sequence tasks, our focus is on code generation. In this line, there have been multiple models developed in the past. In particular we want to refer to CodeT5 by Salesforce (Yue Wang et. al.,). humble attempt is to *translate* English text to a corresponding SQL query. We detail our approach in section 2, experimental setup in section 3 and results in section 4.

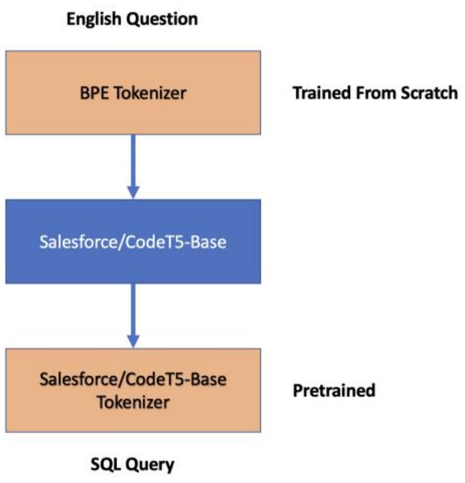

Figure 2: High level idea of the approach taken to train English-to-SQL model.

2 Method

We see the SQL code generation task as machine translation, much like we translate from one natural language to another, such as English to German. In order to accomplish this, we consider using pre-trained model CodeT5-base from Salesforce. We use Huggingface library for all our coding work.

As shown in figure 2, we employ two tokenizers to work with our model. We chose BPE tokenizer and trained it from scratch to tokenize our English text. For SQL code on the other hand, we used pre-trained tokenizer from Huggingface corresponding to Salesforce's CodeT5 model, which could help us tokenize the SQL code easily.

For baseline comparison, we replace the CodeT5 model and the tokenizer with randomly initialized Transformer-Encoder-Decoder (the one implemented by us in Homework 5) and another BPE tokenizer trained on the available SQL text from scratch.

Since the CodeT5 model was specially trained with natural language and code, we did not need to introduce any additional layers, hence our block diagram does a fine job of simplifying the setting.

3 Experimental Setup

For our experimentation we used Spider dataset [3]. It is collected by 11 Yale University students who manually annotated the dataset. It consists of ~10,000 questions and ~5,700 SQL queries (ranging from simple to complex) that query 200 databases. For the purpose of our task, we source the dataset from huggingface instead, and it gives us 7,000 training samples and 1,000 validation samples. Each sample comes with multiple features such as database ID, question, query, question tokens, query tokens. Of these, we only pick query and question as they serve as target and input sequences respectively. The fields are detailed in Table 1.

Having decided on the model, dataset and the baselines, we trained both models on google cloud on A100 GPUs. We used wandb as a tool for better logging, tracking and exhaustive hyperparameter search.

We considered learning rate, batch size, epochs, and sequence length. Since we were dealing with database queries, which are shorter than natural language sentences or other languages such as python where sentences could be long.

To evaluate the performance of our model, we make use of BLEU score. It seemed like a logical

[2]https://developer.nvidia.com/blog/scaling-language-model-training-to-a-trillion-parameters-using-megatron/

[3] https://yale-lily.github.io/spider

choice for us as generated codes are not always

Fields	Description
db_id	Database ID
query	SQL Query
question	English text with question
query_toks	SQL query broken down into tokens
question_toks	Question text broken down into tokens
query_toks_no_value	SQL query broken down into tokens, BUT ignoring keywords
sql	Detailed breakdown of query according to the keywords such as FROM, SELECT, WHERE, GROUPBY, ORDERBY

Table 2: Spider data (sourced from Huggingface library) sample fields and their corresponding descriptions

accurate and hence could very well not be a valid SQL query that could be run against the database for validation. Using a BLEU score, we could at least push our model to generate as many tokens as possible that are a valid match.

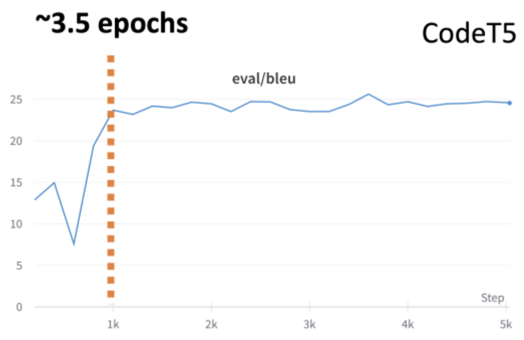

Figure 3: Train history of our model with pre-trained CodeT5 model with pre-trained tokenizer to process SQL queries. X-axis represents training steps and Y-axis represents BLEU score

4 Results

In this section we try to share our results. Of the two model versions, one with pre-trained CodeT5 components and one with our randomly initialized baseline model, figures 3 and 4 show the results of our best runs as part of our wandb sweeps.

In figure 3, we show the CodeT5's performance as we fine tune with our Spider dataset. We can see that the model quickly achieves a decent BLEU score of 23+ in under 4 epochs. Here steps (shown in x-axis) depend on dataset length and batch size (of the run). The red line is taken as a reference point to compare this performance with the baseline model.

Similar to the CodeT5 model, in figure 4 we can see the performance of baseline model with each training step. With the reference line we can see that for the same ~3.5 epochs, the baseline model achieves close to ~16.8 BLEU score.

We noticed that if we kept baseline model to train for longer epochs, then the random

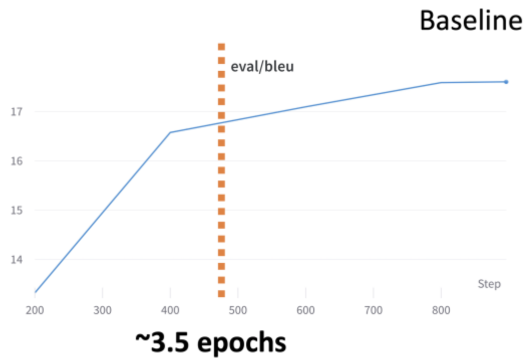

Figure 4: Performance of baseline model with respect to training steps. Baseline model is simply randomly initialized transformer model of nearly the same size as CodeT5. X-axis represents training steps and Y-axis represents BLEU score

initialization has less/no impact, and the model achieves much higher BLEU scores. Because of this we limited the epochs in our wand sweep for baseline model to be under 10.

Additionally, with our training data being so small, and our models being so heavy with significantly more number of weights (parameters), the model could very well be memorizing the samples, which could explain such high BLEU scores.

For our model with CodeT5, we achieved a best BLEU score of 25.7 in 24 epochs, batch size of 24, and learning rate of 0.00029.

For baseline model, we got a best BLEU score of 18, with batch size of 55, epochs 6, and learning rate of 0.000095. It is worth noting that although it may seem like we gave CodeT5 more number of epochs to learn, we can see in Figure 3 that the curve had come close to saturation levels early on, which wasn't the case with our baseline.

Additionally, we noticed that sequence length really mattered here. We need not keep higher lengths like 200-300. In both our best runs, the maximum sequence lengths ended up being 15 and 14 for CodeT5 and Baseline models respectively.

5 Conclusion and future work

From the perspective of a college project meant to give experience molding existing toolkit to try generating code, this is satisfactory. We could see that decent BLEU scores could be achieved quickly. But this being an interesting problem where accuracy really matters, perhaps there is ample room for improvement. Additionally, we only experimented with a smaller dataset, which is one option to pursue. We could also consider exploring other hyperparameters as well to see if we can achieve better results.

Acknowledgments

We would like to thank the Teaching Assistant, Vlad Lialin, for his responsible presence throughout the course and project with helpful tips and directions.

References

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. *ImageNet Classification with Deep Convolutional Neural Networks*. NIPS 2012: Neural Information Processing Systems, Lake Tahoe, Nevada.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. *Deep Residual Learning for Image Recognition*. https://doi.org/10.48550/arxiv.1512.03385

Sergey Ioffe, and Christian Szegedy. 2015. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. https://doi.org/10.48550/arxiv.1502.03167

Diederik P. Kingma, and Jimmy Ba. 2014. *Adam: A Method for Stochastic Optimization*. https://doi.org/10.48550/arxiv.1412.6980

Ashish Vaswani, Noam M. Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser and Illia Polosukhin. 2017. *Attention is All you Need*. ArXiv, abs/1706.03762.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. *BERT: Pre-training of Deep Bidirectional Transformers for Language*. https://doi.org/10.48550/arxiv.1810.04805

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. *CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation*. https://doi.org/10.48550/arxiv.2109.00859

A Appendices

We had shared contributions throughout and didn't have clear separation of tasks. Codes being relatively similar to that of HW5, we only had to fix some errors we got while introducing new pre-trained components. Report writing was a shared effort as well. As far as code on github is concerned, only one person added code there as it was done at the last minute for documentation.