Deep Learning Project

Fine-Tuning On Essence

1 Introduction

The primary goal of this project is to fine-tune the pre-trained T5-small model for the complex task of Text-to-SQL generation within the MAC-SQL multitasking framework. Text-to-SQL is a challenging natural language processing (NLP) task that involves translating user questions into structured SQL commands executable by relational databases. This ability has become increasingly important, as it empowers non-technical users to interact with databases intuitively, enhancing data accessibility and retrieval.

Beyond simply applying fine-tuning, this project aims to explore the essence and key strengths of fine tuning itself—namely, the ability to adapt a general-purpose model to a specialized task using very limited data. One of the central questions we investigate is whether a very lightweight model can still yield competitive results when fine tuned on only a few hundred examples per task. In an era where massive models dominate, we challenge this paradigm by putting a small, efficient model into play and evaluating how far it can go in terms of accuracy, generalization, and real-world applicability.

To address the inherent complexity of Text-to-SQL generation, we use the MAC-SQL framework, which decomposes the problem into three subtasks: Selector (schema selection), Decomposer (query breakdown), and Refiner (query correction). Each task is treated independently, with a dedicated dataset containing source (the input prompt) and target (the expected SQL or response). By fine-tuning a compact model—t5-small-awesome-text-to-sql—on this multitask setup, we aim to assess its potential in a resource-constrained environment while gaining insight into the practical trade-offs between model size, data availability, and task performance.

2 State of the art

2.1 Text-to-SQL

Enterprises especially often face a gap between data accessibility and decision-making. While vast amounts of structured data exist, querying it typically requires SQL knowledge, limiting access to technical users. Text-to-SQL systems aim to bridge this divide by enabling natural language access to databases—empowering business users to ask questions directly. However, enterprise settings pose unique challenges such as complex schemas, evolving data models, and domain-specific language.

Evolution of Text-to-SQL Systems: From Rule-Based Interfaces to Large Language Models

In the early decades of computing (1970s–1980s), accessing structured data from relational databases was a task reserved for trained professionals. Business users were excluded from direct interaction, relying instead on periodic reports generated by IT staff. This gap limited real-time, data-informed decision-making across organizations.

To address this, Business Intelligence (BI) tools such as SAP BI, Oracle BI, and Tableau gained traction, offering users graphical dashboards with predefined metrics. While these tools improved data accessibility, they lacked flexibility—any ad hoc analysis or custom query still required technical intervention.

Early Natural Language Interfaces: Rule-Based and Template Systems

By the early 2000s, Natural Language Interfaces to Databases (NLIDBs) began to appear, aiming to simplify access to databases by translating natural language questions into SQL queries using handcrafted rules and template-based systems. For example, a question like “How many employees…” would trigger a rule mapping it to a COUNT SQL clause. These systems were useful but rigid, domain-specific, and costly to adapt to new schemas or vocabularies, limiting their scalability and generalization capabilities [1].

Sequence-to-Sequence Models: The Deep Learning Alternative

The rise of deep learning brought a shift toward Sequence-to-Sequence (Seq2Seq) models based on Recurrent Neural Networks (RNNs), which enabled learning the mapping between natural language inputs and SQL outputs from data rather than rules. This approach reduced manual engineering and introduced greater flexibility. However, Seq2Seq models struggled with generalizing to unseen schemas, and their performance was strongly tied to the size and quality of labeled training datasets [1, 2].

To mitigate training challenges such as vanishing gradients, Long Short-Term Memory (LSTM) networks were introduced, improving the ability to handle long-range dependencies. Still, these models fell short when faced with the complex hierarchical and relational structures inherent in SQL queries.

Transformer Models and Pretrained Language Models (PLMs)

A major breakthrough arrived with the Transformer architecture, introduced in Attention Is All You Need (2017), which eventually powered models like BERT, GPT-2, and T5. These Pretrained Language Models (PLMs) enabled transfer learning, contextual embeddings, and few-shot adaptation, substantially improving Text-to-SQL performance. Models could now be adapted to downstream tasks with fewer labeled examples, allowing broader generalization across domains and schemas.

By leveraging massive corpora, PLMs not only improved syntax and fluency in SQL generation but also dramatically reduced reliance on domain-specific rule sets or templates. Between 2018 and 2020, PLMs became the state of the art in the field.

LLM-Powered Text-to-SQL Systems

Today, Large Language Models (LLMs) such as GPT-4, PaLM-2, and CodeLLaMA represent the current frontier in Text-to-SQL systems. These models, often fine-tuned or adapted using techniques like prompt engineering, instruction tuning, and retrieval-augmented generation (RAG), can generate accurate and executable SQL queries for complex and unfamiliar schemas[1].

Their capabilities are benchmarked on datasets like Spider and BIRD, which simulate real-world scenarios involving multi-table queries and diverse domains. Unlike earlier approaches, modern LLM-based systems can generalize across

databases with minimal fine-tuning and deliver strong performance even with limited supervision, making them increasingly viable in enterprise applications where flexibility and scalability are essential.

2.2 MAC-SQL Architecture [3]

Introduction

MAC-SQL demonstrates state-of-the-art results on the BIRD benchmark, achieving an execution accuracy of 59.59% on the test set which makes it an interesting approach.
As presented in [Wang et al., 2023] paper, the main contributions of MAC-SQL are the following:

1. Introduction of a Multi-Agent Collaborative Framework
2. Development of SQL-Llama: To bridge the gap between closed and open-source models, the authors introduce SQL-Llama, an instruction-tuned model based on Code Llama 7B. This model replicates the behavior of each agent in the MAC-SQL pipeline and serves as an accessible alternative to proprietary LLMs like GPT-4.

Framework

MAC-SQL introduces a multi-agent architecture composed of three main agents: the Selector, the Decomposer, and the Refiner. Each plays a distinct role in facilitating robust and accurate SQL generation from complex natural language queries.
Roles:

- The Selector minimizes noise by pruning the database schema, focusing the attention of the LLM on relevant subsets of tables and columns.
- The Decomposer transforms intricate queries into simpler sub-questions, allowing for a stepwise resolution using chain-of-thought reasoning.
- The Refiner serves as an error-detecting and correcting mechanism, analyzing failed SQL executions and iteratively improving the output through feedback.

This division of labor enables the system to dynamically address complexity at multiple levels: schema size, question structure, and execution accuracy.

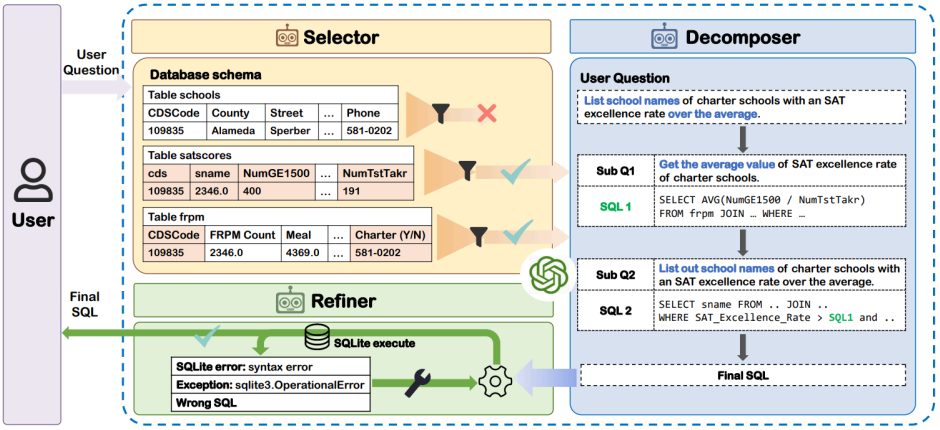

Figure 2.1: MAC-SQL framework example of execution see Annex II for a better understanding and explanation of the tasks and agents through this example

2.2 Fine-tuning in this framework

Fine-tuning is a subset of transfer learning. Which is a supervised learning technique where a pre-trained language model is adapted to a specific downstream task by training on labeled examples. This updates the model's weights to optimize performance on the target task. Modern deep learning libraries such as Hugging Face's Trainer API streamline this process by providing efficient tools for data loading, training loop management, evaluation, and checkpointing. In this project, we leverage these tools to effectively fine-tune our model on a multitask instruction dataset.

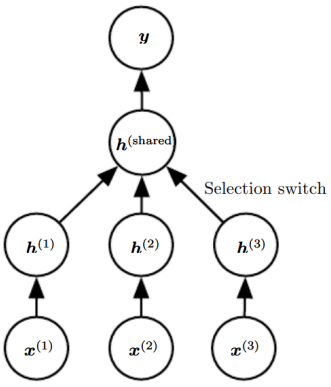

Figure 1.2: Transfer learning when the output variable y has the same semantics for all tasks while the input variable y has a different meaning (and possibly even a different dimension) for each task (or, for example, each user), called $x^{(1)}, x^{(2)}, x^{(3)}$ for three tasks. The lower levels (up to the selection switch) are task-specific, while the upper levels are shared. The lower levels learn to translate their task-specific input into a generic set of features.

Transfer learning, as illustrated in Figure 15.2, involves adapting a pretrained model to multiple related tasks that share the same output semantics y but have different input domains corresponding to three distinct tasks. In the MAC-SQL framework, these inputs $x^{(1)}, x^{(2)}, x^{(3)}$ represent the three agent tasks: the Selector, the Decomposer, and the Refiner. The lower levels of the model specialize in processing each task-specific input, while the upper levels share learned representations common to all tasks. This structure enables the model to translate diverse inputs into a unified feature space for better generalization [4,5].

In this framework, each input $x^{(i)}$ (Selector, Decomposer, or Refiner) is processed by a dedicated task-specific layer $h^{(i)}$ designed to capture the unique characteristics of that task. These task-specific representations are then passed to a shared upper layer $h^{(shared)}$, which integrates and transforms the features into a unified representation used to predict the output y.

Since the model - we will discuss this decision on the next section -, is already fine-tuned on SQL, the shared layer h^{shared} encodes general linguistic and semantic knowledge relevant to SQL translation. When downstream adapting to the Selector, Decomposer, and Refiner tasks, this shared representation allows efficient reuse of that knowledge. Fine-tuning updates both the shared and task-specific layers to better capture the unique nuances of each task, enabling effective multitask learning with minimal additional data [4,5].

During fine-tuning, both the task-specific and shared layers are updated to optimize performance across all tasks. This structure enables the model to specialize on the nuances of each agent task while simultaneously learning common patterns that generalize well.

By using this approach, fine-tuning leverages the pretrained model's general knowledge and adapts it to the specific requirements of the Selector, Decomposer, and Refiner tasks, improving efficiency and effectiveness on the multitask Text-to-SQL problem with limited labeled data.

2.3 Models

MAC-SQL: SQL-Llama V0.5

The model that presents [Wang et al., 2023] is the following. Built upon the Code Llama 7B base model, SQL-Llama is fine-tuned specifically to perform multiple agent tasks crucial for effective SQL generation. Code Llama 7B is an open-source large language model developed by Meta AI, based on the Llama 2 architecture but specialized for code-related tasks. Code Llama 7B was trained on a diverse dataset of programming languages and natural language.

To train SQL-Llama, the authors created a comprehensive Agent-Instruct dataset by leveraging GPT-4 to generate multi-agent instruction data based on existing Text-to-SQL benchmarks such as BIRD and Spider. This dataset contains around 10,000 high-quality instruction examples spanning three agent-instruction tasks.

The instruction dataset was carefully filtered to ensure correctness and balanced coverage across different difficulty levels and database complexities, providing rich supervision for the multitask fine-tuning process which will be using supervised full finetuning.

T5-small-awesome-text-to-sql

Is a fine-tuned variant of the T5-small model designed specifically for the Text-to-SQL task. It is based on the original T5 (Text-to-Text Transfer Transformer) architecture, which frames all NLP problems as text-to-text generation, enabling a unified approach to tasks such as translation, summarization, and question answering. T5 is a sequence-to-sequence model. The original T5-small model consists of an encoder-decoder transformer architecture with approximately 60 million parameters, offering a compact yet powerful base. "Awesome-text-to-sql" is fine-tuned on datasets such as b-mc2/sql-create-context and Clinton/Text-to-sql-v1, which provide paired examples of natural language questions and their corresponding SQL queries. These datasets include multi-table queries with JOIN operations, enabling the model to handle complex relational database structures.

Comparison

The models explored in this study are primarily open-source, in contrast to ChatGPT-4, which is closed-source. As detailed in Annex I—particularly in the strengths and parameters rows—ChatGPT-4 stands out due to its significantly larger number of parameters. However, its closed nature imposes key limitations: developers cannot modify the model, inspect internal training metrics (such as loss curves), or perform any custom fine-tuning. These constraints make it unsuitable for our experimental objectives, which require transparency, control, and adaptability.

In contrast, SQL-Llama v0.5 effectively narrows the performance gap with GPT-4 by applying multitask instruction fine-tuning within the MAC-SQL framework. Despite its more modest parameter count, SQL-Llama achieves strong execution accuracy, illustrating the strength of instruction tuning combined with multitask learning in the Text-to-SQL domain.

Among the models available, t5-small-awesome-text-to-sql emerges as the most fitting choice for this project. Although smaller than other T5 variants, it offers several advantages critical to our goals:

- Lightweight architecture: Its reduced size means lower computational requirements, making it ideal for simulating training and deployment in resource-constrained environments. The smaller model size enables faster fine-tuning cycles and quicker deployment, allowing iterative experimentation and real-time applications where latency matters. Ideal for chatbot applications and this project where experimentation is key. Due to its relatively small size (number of parameters) requires significantly fewer computational resources for training and inference. This is beneficial for simulating a resource limited scenario. The smaller model size enables faster fine-tuning cycles and quicker deployment, allowing iterative experimentation and real-time applications where latency matters. Ideal for chatbot applications and this project where experimentation is key

- Fast experimentation: The smaller parameter count enables shorter fine-tuning cycles and quicker deployment, which is especially valuable in iterative, experiment-driven projects.

- Pre-fine tuned availability: Since we already have access to a version fine-tuned on relevant Text-to-SQL data, we avoid redundant training while gaining a strong baseline for further enhancement. Also the data which is available to finetune the model is the same data that was used to finetune SQL-Llama-v0.5 are going to be using for fine tuning the model

- Better task alignment: While larger models like Code LLaMA 7B were considered, their training focus is primarily code-centric. In contrast, T5-small—despite being pre-trained on text-to-text tasks—has already been fine-tuned for Text-to-SQL generation, making it more suitable for applying multitask refinement across Selector, Decomposer, and Refiner subtasks.

In summary, t5-small-awesome-text-to-sql is a practical and efficient model for exploring fine-tuning effectiveness under constrained conditions. It embodies a focused adaptation of the T5 architecture that supports natural language to SQL translation in real-world, low-resource environments—while offering a meaningful framework to assess how far small models can be pushed with the right strategy.

3 Methodology

3.1 Data Analysis

Data inspection

We worked with two .jsonl files derived from the dataset originally used to fine-tune the SQL-Llama model: processed and raw. While both files contain the same training instances, the processed version includes pre-tokenized fields such as input_ids, which are specific to the tokenizer used with SQL-Llama. Since we are not using the LLaMA tokenizer, these fields are not relevant to our setup. Therefore, we chose to work with the raw file, which contains the original text data without model-specific preprocessing.

The dataset comprises 3,375 examples, which we assume were curated by the dataset creators through filtering to ensure high-quality samples. For our experiments, we focus on two primary fields:

- source: the natural language input or prompt.
- target: the corresponding SQL query that serves as the expected output.

These input-output pairs form the core of our fine-tuning dataset.

We conducted an initial data analysis and cleaning process, which included checks for duplicate entries, null values, and anomalies. None were found, which aligns with the claims in *[Wang et al., 2023]*, stating that the dataset had been preprocessed and filtered prior to release from 10,000 instances to 3,375. To further explore potential irregularities, we plotted boxplots based on the lengths of the source and target fields, since the data is textual. While many outliers were observed, these are expected and can be attributed to variations in task type, schema complexity, and prompt structure—longer or more complex schemas naturally lead to longer input queries or generated SQL. Also are attributed to our main insight.

Main insight: the identification of task imbalance across the dataset. The dataset contains three distinct tasks, and a histogram of instance distribution revealed a significant skew:

- Decomposer: 2,029 instances
- Selector: 1,009 instances
- Refiner: 337 instances

This imbalance has implications for the model's ability to generalize across tasks, particularly for the underrepresented Refiner task. The distribution of task types is shown in Figure 3.1.

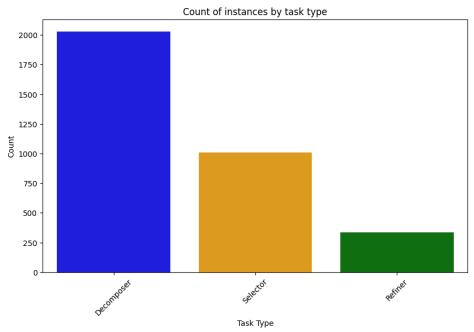

Figure 3.1 Histogram distribution grouped by task type

Trainer API

We are going to use the Trainer API from Hugging Face's Transformers library, which is a high-level interface designed to simplify the process of fine-tuning and evaluating transformer-based models. It abstracts away much of the boilerplate code involved in training, making it easier to implement supervised learning workflows on top of pre-trained models. The API manages key components such as data loading, training loops, evaluation, logging, checkpointing, and distributed training, all while allowing for customization through arguments and callback functions.

It is especially useful in scenarios involving large language models (LLMs) and is fully compatible with datasets formatted as Dataset objects or standard PyTorch DataLoaders. One of its key advantages is that with minimal configuration, users can define training arguments, pass in a model and dataset, and execute end-to-end training, logging and saving best-performing checkpoints.

Preprocessing

Training data

With the objective to do the training splits according to the task imbalance and to get insights separately from each of the tasks we add a new column "type" which we will get from the static start of the prompt which corresponds to column "source".

For Trainer API

To use Hugging Face's Trainer efficiently, proper preprocessing of the data is essential. We define a custom SQLDataset class that inherits from PyTorch's Dataset to seamlessly integrate with data loaders and training utilities.
Previously having loaded the model and tokenizer according to the model. The dataset is responsible for tokenizing both the input text 'source' and the 'target' text using the provided tokenizer. It applies consistent padding and truncation to a fixed maximum length– the one according to the model – , guaranteeing that each example is converted into properly formatted tensors ready for model consumption. The formatted tensors corresponding to:

- **input_ids**: Tokenized representation of the input text (source), converted into integer IDs using the T5 tokenizer.
- **attention_mask**: Binary mask indicating which tokens are actual input (1) and which are padding (0), so the model ignores padded positions.
- **labels**: Tokenized target text (target), used for supervision—T5 tries to generate this output given the input_ids.

Additionally, including the type field in the returned data allows the training process to differentiate between task categories, facilitating multitask learning.
To prepare batches during training, we use a sequence-to-sequence (seq2seq) data collator. A data collator is a utility that dynamically pads token sequences within each batch so that all inputs have the same length, which is necessary for efficient batch processing. It also formats input tensors—such as token IDs and attention masks—into the exact structure expected by the model. Using a collator improves training efficiency by minimizing unnecessary padding, simplifies the data pipeline by automating batch preparation, and ensures that inputs and targets are properly aligned for accurate loss computation and overall model performance.

3.2 Models and evaluation

Figure 3.2 How by fine tuning we aim to obtain different models on a more downstreamed task
The model architecture has already been explained progressively throughout the report, alongside the theoretical background of the models. See Figure 3.2 for a summary and a better visual description.
In this section focused on feasibility, we will instead concentrate on the loss function, initial hyperparameters, and evaluation metrics. This change in structure reflects the experimental approach: we first conducted Experiment 1 based on initial data analysis, and subsequently developed further experiments informed by the results obtained in previous runs. This step-by-step progression allows us to get insights based on the necessities and what to improve from the previous experiment.

Initial hyperparameters

For the first experiment, we designed a training setup aimed at balancing efficiency, training stability, and adaptability to a low-resource scenario. The function run_exp_1() initializes the training using Hugging Face's Trainer API along with a carefully selected set of hyperparameters via the TrainingArguments configuration.
We used a learning rate of 5×10^{-5}, a common choice for fine-tuning pretrained models, as it allows the model to gradually adapt to the new task (Text-to-SQL) while retaining its general language understanding. This is particularly important when training on small, domain-specific datasets.
The model was trained over 10 epochs with a batch size of 1 per device, both for training and evaluation. The small batch size ensures fine-grained updates to the model parameters and fits within the memory limits of typical GPUs. Although training with small batches can be noisier, it enables more frequent gradient updates and is suitable for small datasets.
To monitor training progress, the model is evaluated and saved at the end of each epoch (eval_strategy="epoch", save_strategy="epoch"), and only the best-performing checkpoint is retained (save_total_limit=1, load_best_model_at_end=True, metric_for_best_model="eval_loss"). This setup avoids overfitting by ensuring that the final model corresponds to the lowest evaluation loss observed.
We also enabled mixed precision training using bfloat16 (bf16=True), which speeds up training and reduces GPU memory usage without compromising numerical stability. Additionally, training logs are recorded every 50 steps (logging_steps=50) to keep track of progress.
Finally, a custom LossLoggerCallback is attached to the trainer to capture and store training/validation loss across epochs, enabling detailed post-training analysis of underfitting or overfitting behavior.
These choices form the initial hyperparameter configuration, while the actual model parameters (i.e., the internal weights and biases of the t5-small-awesome-text-to-sql model) are learned through backpropagation during training. This setup establishes a strong baseline for evaluating the effect of task balancing and later tuning experiments.

Loss

Cross-entropy loss is the one used by default on the Trainer API and the one proposed and used on [Wang et al., 2023], for fine-tuning sequence-to-sequence models like T5. This loss measures the difference between the predicted token probabilities from the decoder and the ground truth target tokens, encouraging the model to generate accurate sequences. As stated on the paper with slight changes because of the model this would be loss on a whole execution then overall tasks:

Given the dataset with $N = 3$ instruction tasks $D = \{D_i\}_{i=1}^{N}$, we finetune the model to learn and complete these agent tasks. The supervised fine-tuning objective can be formulated as the minimization of the negative log-likelihood over all tasks:

$$L = -\sum_{i=1}^{N} \mathbb{E}_{Q,S_i,K,Y_i \sim D_i} \left[\log P\left(Y_i \mid Q, S_i, K; M\right)\right]$$

Where L is the total loss over all tasks, S_i represents the selected database schema and Y_i is the intermediate SQL query output for the i-th task. Here, M denotes the model.
Because T5 is a sequence-to-sequence transformer designed for conditional text generation, the loss corresponds to the cross-entropy between the predicted token sequence and the ground truth SQL query tokens.

Evaluation Metrics

To evaluate the performance of our Text-to-SQL model, we considered several metrics, each with its own strengths and limitations.

- Execution Accuracy (VES – Valid Execution Semantics):This metric checks whether the generated SQL query, when executed on the database, returns the same result as the ground truth query. It does not require the SQL strings to be identical—only that they produce equivalent outputs when run. This makes VES a realistic and practical metric, as the goal of a Text-to-SQL system is ultimately to retrieve the correct answer, not to mimic exact syntax.
  However, in our case, despite having downloaded the Spider and BIRD benchmark databases, the training data lacked information linking each instance to a specific database. As a result, we could not reliably execute the queries, making VES inapplicable for this experiment.
- Exact Match (EM):This metric checks whether the predicted SQL query exactly matches the reference query, token by token. It is very strict—even small, functionally irrelevant differences (e.g., extra parentheses, reordered conditions) will result in a failed match. Despite its limitations, EM was helpful for verifying whether the model followed the task-specific structure present in the training data.
- ROUGE-L:To introduce more flexibility, we used ROUGE-L, a metric based on the Longest Common Subsequence (LCS) between the predicted and reference queries. Originally designed for text summarization, ROUGE-L is useful here to assess partial correctness and structural similarity, especially when EM fails to reward nearly correct outputs.

In summary, while VES would have been the most meaningful evaluation metric under ideal conditions, the lack of schema linkage prevented its use. We therefore relied on EM and ROUGE-L to balance between exact syntactic matching and more forgiving semantic similarity.

4 Experiments

4.0 Pipeline

For each experiment, once a model configuration was defined, we carried out a full fine-tuning cycle. This included training the model with the corresponding dataset and hyperparameters, followed by the generation and analysis of training and evaluation loss curves to observe convergence behavior and diagnose potential overfitting or underfitting. After training, we selected the best-performing checkpoint based on validation loss and evaluated it using two key metrics: ROUGE-L, to assess semantic similarity, and Exact Match Accuracy, to measure exact token-level correspondence with the ground truth SQL.
Each subsequent experiment was built incrementally by taking the best-performing configuration from the previous experiment and introducing new adjustments—such as modified batch size, learning rate, or training strategies—allowing us to systematically test the impact of each design choice on model performance.

4.1 Experiment 1: Baseline vs. Task Balancing

Aim: To assess the impact of task imbalance on model performance while operating under a low-resource setting.
This experiment explores two undersampling strategies for training the model using a reduced subset of the original dataset, which contains 3,375 examples. Given our objective of fine-tuning with very little data, we limited both training configurations to approximately 1,000 samplesand as there were just 337 refiner instances and wanted to balance without having to do data augmentation.
The first configuration, referred to as the *baseline*, consisted of 1,011 randomly sampled instances that preserved the original task distribution (Decomposer being the majority, followed by Selector and Refiner). Although this setup involved global undersampling, it maintained the natural class imbalance. The resulting data was split into 80% for training and 20% for validation, using stratification to preserve proportions.

The second configuration applied class-aware undersampling to ensure balance across tasks. Specifically, we selected 337 examples for each task—the size of the minority class—resulting in a balanced dataset of 1,011 instances. This balanced set was also split into an 80/20 training-validation ratio, with equal representation from each task in both subsets.
Also to see if there is some difference between the loss in each of the tasks we add in this experiment the plot of training losses.

4.2 Experiment 2: Batch Size, Epochs & Early Stopping

Aim: To improve model stability and generalization by optimizing training dynamics under the balanced data setup.
The experiment tested two batch sizes (4 and 8) to compare the impact of gradient stability versus update frequency. Smaller batch sizes typically introduce more noise into the gradient updates, which can act as a form of implicit regularization and improve generalization, while larger batch sizes provide smoother, more stable updates but may lead to slower or less robust convergence, particularly in low-data regimes.
To allow sufficient training cycles for all configurations, we increased the maximum number of epochs to 20. However, to avoid overfitting in longer training runs, we enabled early stopping with a patience of 2 epochs, using validation loss as the monitoring metric. This ensured that training would terminate once performance plateaued, preserving only the best model checkpoint.
The use_liger_kernel=True flag was also used to enhance GPU throughput where supported and not have memory errors, especially useful when training with larger batch sizes.
This exercise will not get representation on results or discussion as it is already included in the combined version on experiment 3.

4.3 Experiment 3: Learning Rate

Aim: To identify the optimal learning rate and batch size combination.
This experiment focused on tuning one of the most critical hyperparameters in deep learning: the learning rate. The learning rate determines the step size at which the model updates its parameters during training. If set too high, the model may diverge or exhibit erratic behavior, overshooting the optimal weights. If too low, the model may converge too slowly or get stuck in suboptimal minima, resulting in underfitting. Hence, finding an appropriate learning rate is essential for model stability, efficient convergence, and generalization performance.
Either than excluding batch size and learning rate are This experiment aimed to tune the learning rate by testing combinations of: batch sizes: 4 and 8 and learning rates: 3e-5, 5e-5, 1e-4

4.4 Experiment 4: Final Model Evaluation

Aim: To qualitatively assess the best model's real-world output across task types.
Model evaluated on a small set of real examples randomly assigned from the complete dataset—two per task. This qualitative check provides insight into how well the model generalizes across the Decomposer, Selector, and Refiner tasks in practical inference scenarios and lets us see insights that are difficult to see by previous steps.

5 Results

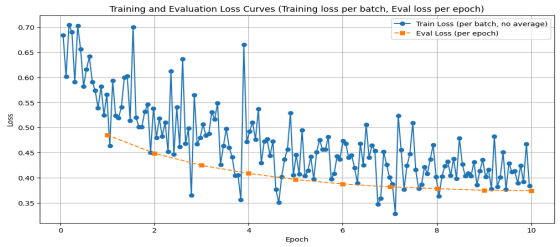

Figure 5.1 Train and evaluation loss curves on real distribution

Figure 5.2 Training loss per type of task on real distribution for 800 steps

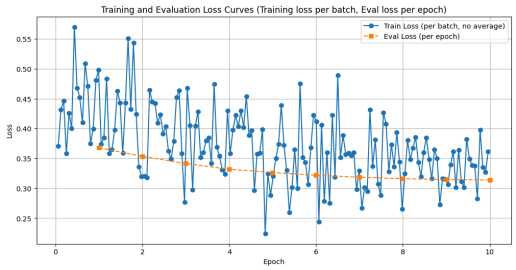

Figure 5.3 Train and evaluation loss curves on balanced distribution

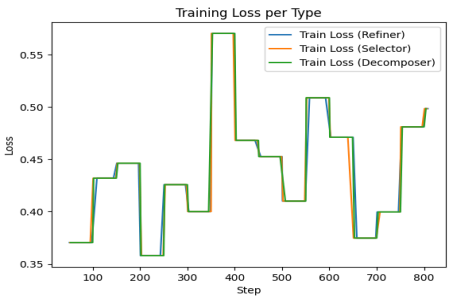

Figure 5.4 Training loss per type of task on real distribution for 800 steps

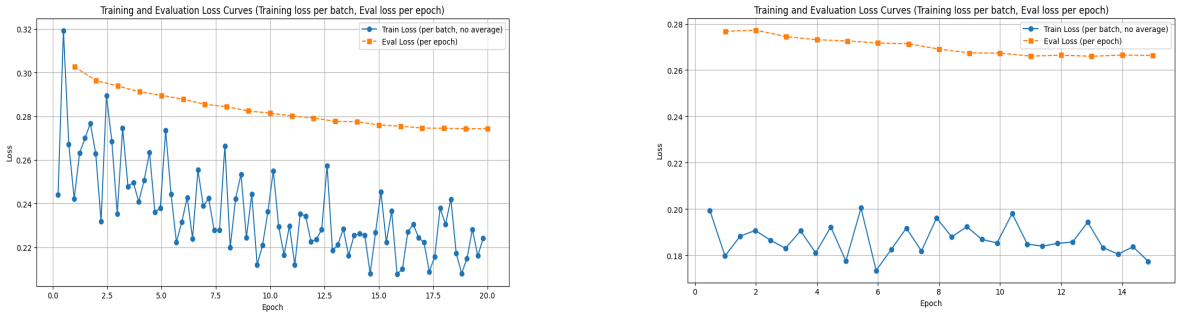

Figure 5.5 Train and evaluation loss curves for learning rate 0.0001 and batch size per device 4 (left) and 8 (right)

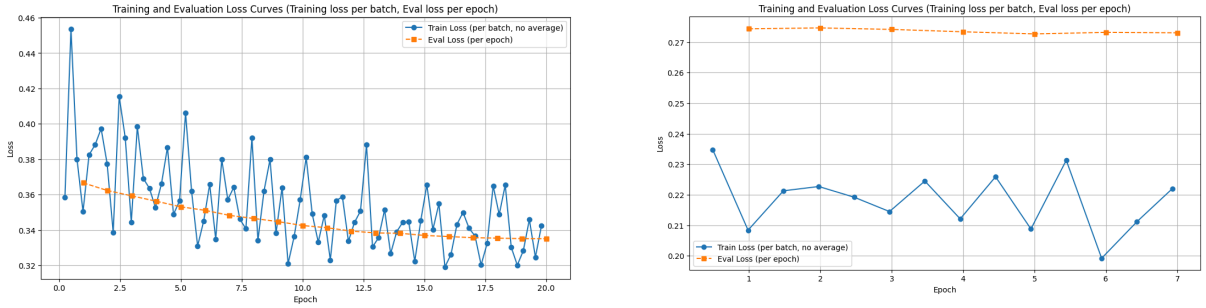

Figure 5.6 Train and evaluation loss curves for learning rate 3e-5 and batch size per device 4 (left) and 8 (right)

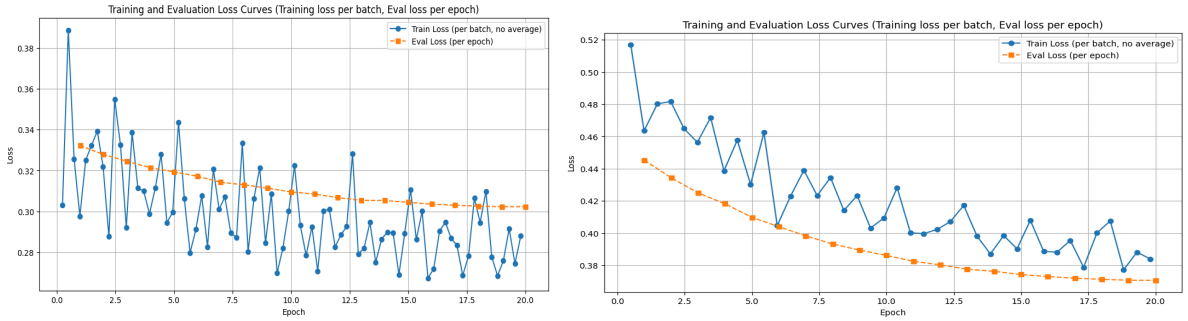

Figure 5.7 Train and evaluation loss curves for learning rate 5e-5 and batch size per device 4 (left) and 8 (right)

6 Discussion

Many challenges were encountered: model installation, allocating GPU in jupyter, memory limitations, finding metrics and including, understanding the whole framework etc. Nevertheless this is an issue that might be determining:

Loss plot: To monitor the learning behavior of each model, we used a custom visualization function that plots both training and evaluation loss curves. This function reads the trainer_state.json file automatically generated by Hugging Face's Trainer, which logs training metrics over time.

The training loss is recorded frequently—typically once every few steps—so it is plotted as a dense curve showing how the model's performance evolves within each epoch. While this curve may appear noisy, it offers a detailed view of how the model is adapting to the training data in real time. Maybe it was not a good decision and should get a balance on the level of detail and getting good insights.

In contrast, the evaluation loss is computed only once per epoch, based on the model's performance on the validation set. This is plotted as a cleaner, more stable curve, which allows us to observe whether the model is generalizing well or beginning to overfit.

Experiment 1

The model trained on the real distribution (see *Figure* 5.1) showed a decreasing validation loss overall, but the training loss was highly unstable, with frequent oscillations and sharp spikes. The task-specific breakdown (see *Figure* 5.2) confirmed that all task types followed similar trends, but due to the imbalance, the model was biased and the refiner we can see that it deviates from the trend because of underrepresentation. This setup reflects real-world usage frequencies but risks limiting the model's generalization across task types.

In contrast, the model trained on the balanced dataset (see *Figure* 5.3) exhibited more consistent and smooth learning behavior, with both training and evaluation losses decreasing steadily. The per-task loss analysis (see *Figure* 5.4) showed that the model learned each task at a comparable pace, indicating a more equitable and robust training process.
These results suggest that task balancing improves training stability and ensures fairer learning across subtasks, particularly benefiting those with fewer examples. Nevertheless this one had a ROUGE-L of 0.19 which is less than the one on the real distribution case.

Experiment 3

Observations from the Training and Validation Curves

- Right figures 5.5-5.7
- Batch Size 4, Learning Rate 1e-4 showed fast convergence and achieved the lowest evaluation loss (~0.267). However, the validation loss plateaued in the final epochs while the training loss continued decreasing, indicating overfitting.
- Batch Size 4, Learning Rate 5e-5 also showed stable learning but at a slower pace, with slightly higher validation loss (~0.301). The curves were smoother and more regularized, suggesting a safer but less efficient training process.
- Batch Size 4, Learning Rate 3e-5 led to underfitting: the model showed limited learning progress, with both losses plateauing early. This confirms that too low a learning rate can prevent the model from effectively updating its parameters.
- Left figures 5.5-5.7
- Across the board, Batch Size 8 configurations showed less favorable learning dynamics Figure 5.7 and 5.8 right really show overfitting. While a larger batch size offers smoother gradients, in this limited-data setting it led to higher evaluation losses, less flexibility, and in some cases early stopping (e.g., at 7 epochs with lr=3e-5) due to limited training signal.

This experiment demonstrates that both learning rate and batch size significantly impact model behavior:

- Lower batch sizes are more suitable for small datasets, enabling better adaptation and generalization.
- Higher learning rates can accelerate convergence but must be used carefully to avoid overfitting.

The analysis of loss curves reinforces the need for careful hyperparameter tuning, especially in multitask, low-resource NLP settings like Text-to-SQL

Experiment 3-4

We have decided to use Batch Size 4, Learning Rate 1e-4 as it was the only one that reached some minimal level of accuracy and had a high ROUGE-L – 0.4 –compared to the initial one 0.19. Meaning improvement.
In all the models accuracy being 0 this might be due to MAC-SQL put inside brackets column and table names to which the model is not used to, among other factors.

Experiment 4

The model did not perform correctly in any of the tasks. Nevertheless it has downstream its initial task to the one on the Overall, the model demonstrated:

- A basic structural understanding of the tasks (e.g., decomposing, identifying joins, or relevance)
- But suffered from hallucinations, schema inconsistency, and formatting errors, especially in Refiner and Selector tasks. Decomposer responses followed the expected format but were disconnected from the input schema, which undermines their usability.

This behavior may be attributed to several factors, including overfitting due to limited training data and the inherent constraints of using a lightweight model. In particular, the model's maximum output length of 512 tokens proved insufficient for handling long and structured responses. As a result, we frequently observed truncated outputs or overly minimal responses, especially in tasks involving long prompts or multi-step SQL generation.

7 Conclusion

The aim of this project was to assess whether a lightweight model like T5-small-awesome-text-to-sql could be effectively fine-tuned for multitask Text-to-SQL generation under low-resource conditions. Despite the model's limitations in handling long outputs and complex queries, the results show that fine-tuning with few examples per task is feasible and can yield usable, partially accurate outputs, especially for simpler decomposition tasks.
We observed that:

- Task balancing improves generalization and learning stability.
- Lower batch sizes and higher learning rates boost convergence but require caution to avoid overfitting.
- Even with a small model, ROUGE-L scores improved notably, showing meaningful learning.

While Exact Match remained low and outputs were sometimes truncated or hallucinated, the model did show evidence of task-specific behavior. In this sense, the core objective—testing fine-tuning feasibility on a compact model in a multitask setup with limited data—was successfully achieved. Further improvements would require longer output capacity, prompt refinements, or light data augmentation.

8 References

[1] Shi, L., Tang, Z., Zhang, N., Zhang, X., & Yang, Z. (2024). *A Survey on Employing Large Language Models for Text-to-SQL Tasks*. http://arxiv.org/abs/2407.15186

[2] Wiesinger, J., Marlow, P., Vuskovic, V., Huang, E., Xue, E., Sercinoglu, O., Riedel, S., Baveja, S., Gulli, A., Nawalgaria, A., Mollison, G., & Haymaker, J. (2024) *Google Whitepaper on AI Agents : Google : Free Download, Borrow, and Streaming : Internet Archive*. (n.d.). Retrieved May 4, 2025, from https://archive.org/details/google-ai-agents-whitepaper

[3] Wang, B., Ren, C., Yang, J., Liang, X., Bai, J., Chai, L., Yan, Z., Zhang, Q.-W., Yin, D., Sun, X., & Li, Z. (2023). *MAC-SQL: A Multi-Agent Collaborative Framework for Text-to-SQL*. http://arxiv.org/abs/2312.11242
[4] Goodfellow, I., Bengio, Y., & Courville, A. (n.d.). *Deep Learning*.
[5] Tunstall, L., von Werra, L., & Wolf, T. (n.d.). *Natural Language Processing with Transformers Building Language Applications with Hugging Face*.

Annex I: Example of combination of tasks execution and combination

Figure 2.1 illustrates the end-to-end SQL generation process under the MAC-SQL framework using a concrete example. The user provides a natural language question, which triggers collaborative reasoning between three agents: the Selector, Decomposer, and Refiner.

The Selector identifies only the relevant tables and attributes needed for answering the question, rather than passing the entire database schema. In the example shown, the schools table is excluded due to irrelevance, and only the necessary columns from the frpm and satscores tables are retained. This selection reduces complexity and improves focus.

Next, the Decomposer breaks the question into sub-questions. The first sub-question computes the average SAT excellence rate (SQL 1), and the second filters schools above this average (SQL 2). Each sub-question builds logically on the previous, allowing the model to reason incrementally.

Finally, the resulting SQL query is validated and executed. If it fails (e.g., due to a syntax or operational error), the Refiner agent iteratively adjusts the query using feedback from the database engine.

Annex II: Comparison of models

Feature	GPT-4	SQL-Llama V0.5	t5-small-awesome-text-to-sql
Base Architecture	Proprietary Transformer-based LLM (OpenAI)	Code Llama 7B (Llama 2 variant specialized for code)	T5-small (Text-to-Text Transfer Transformer)
Parameters	Estimated 100+ billion (not publicly confirmed)	~7 billion	~60 million
Training Data	Massive, diverse corpora including code and text	Multi-agent instruction dataset (~3,370 examples) generated using GPT-4, based on BIRD and Spider benchmarks	Fine-tuned on b-mc2/sql-create-context and Clinton/Text-to-sql-v1 datasets with multi-table queries and JOINs
Training Method	Pretrained and instruction-tuned with RLHF	Supervised full multitask fine-tuning	Supervised fine-tuning on paired NL-to-SQL data
Capabilities	State-of-the-art language understanding and generation across domains, including Text-to-SQL	Handles database simplification, question decomposition, SQL generation, and error refinement via multi-agent framework	Generates SQL queries from NL, supports multi-table queries with joins
Strengths	Best-in-class performance on [Wang et al., 2023] paper	Large-scale, multitask instruction tuning; close performance to GPT-4 on execution accuracy on [Wang et al., 2023]	Lightweight, computationally efficient; suitable for limited-resource environments
Limitations	Closed-source	More computational resources	Limited capacity for very complex queries or extensive schemas due to smaller size