Pre-training Vs Fine-tuning an LLM for SQL text generation

Dibyanshu Chatterjee DC7017@RIT.EDU
Department of Computer Science
Rochester Institute of Technology
Rochester, NY 14623, USA

Apeksha Kulkarni AK3994@RIT.EDU
Department of Computer Science
Rochester Institute of Technology

Rishabh Arora RA8851@RIT.EDU
Department of Computer Science
Rochester Institute of Technology

Snehith Reddy SB3994@RIT.EDU
Department of Computer Science
Rochester Institute of Technology

Editor: None

Abstract

This research presents a comparative study focusing on the pivotal phases in the development of large language models (LLMs): pretraining and fine-tuning, all conducted within a local environment. We aim to shed light on the feasibility of these roles for the purpose of making a task specific LLM, (which in our case is to generate SQL like text) while keeping the scalability and efficiency in consideration. Our study centers on LLMs inspired by the GPT and llama-2 architecture and delves into the intricacies of pre-training, where LLMs acquire linguistic knowledge and also fine-tuning, where LLMs extends their linguistic learning based on the new data the LLM was exposed to. This pretraining phase serves as the foundation for our comparative analysis of the fine-tuning process. We employ the Qlora method, showcasing how it efficiently refines pre-trained Falcon 7B parameters models within the constraints of a local environment. This work underscores the harmonious interplay between pretraining and fine-tuning, providing valuable insights for the machine learning community. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Keywords: Pre-training LLMs, Fine-tuning LLMs, GPT, llama-2, Qlora

1 Introduction

The landscape of natural language processing has witnessed a profound transformation with the emergence of large language models (LLMs). The development of these models, inspired by the GPT architecture, entails two integral phases: pretraining and fine-tuning. Our research endeavors have been executed entirely within a local environment, emphasizing the scalability, efficiency, and real-world applicability of these phases.

In the initial stages of our research, we embarked on the development of a GPT model, which is a decoder-only model. This undertaking serves a dual purpose: to showcase the intricacies of how a GPT model operates and to shed light on its inherent limitations. By conducting a thorough pretraining evaluation, we aim to provide an in-depth understanding of the foundational knowledge acquisition process, which equips LLMs with the linguistic nuances essential for the comprehension and generation of language.

Building on this foundation, we proceed to refine the llama-2-7b model, within the same local environment, illustrating the successful reduction in effort and a simultaneous increase in outcome. Moreover, we undertake showcasing the reduction in loss during this fine-tuning process. These achievements emphasize the efficiency of our local environment-based work, demonstrating the potential for more cost-effective and resource-efficient developments in the field of natural language processing.

Through our research, we aim to not only provide valuable insights into the dynamics of LLM development but also to offer a practical demonstration of how an LLM model can be further enhanced and optimized. This work underscores the potential of fine-tuning in a local environment, reflecting the evolving landscape of machine learning and its application to real-world tasks and challenges.

2 Methodology

Our methodology is deeply rooted in the exploration and comparison of diverse practices in the realm of Large Language Models (LLMs), aiming to uncover the most suitable approaches for distinct scenarios. Within this extensive exploration, we navigate through two pivotal phases, each described in detail below.

2.1 Pretraining of a GPT-Based Decoder-Only LLM

In this phase, we lay the groundwork by meticulously collecting and preparing a domain-specific dataset. The GPT-based model is carefully configured to foster linguistic understanding of SQL. The model is pre-trained to predict the next token in a sequence, effectively arming it with the essence of SQL. The model retrieved after the pre-training process is named tinySQLGPT, the code for which is made available on github at `https://github.com/dibyanshuchatterjee/tinyGPTSQL`.

2.1.1 Dataset

The cornerstone of this phase is the meticulous collection and preparation of a domain-specific dataset. To facilitate our research, we have chosen the 'wikisql' dataset, sourced from Hugging Face. The 'wikisql' dataset contains English text that has been translated into SQL commands. For our purposes, we specifically extract and use the SQL texts from

the relevant columns. This approach allows us to create a corpus that primarily consists of SQL text, which serves as the main building block for our language model's pre-training. It is worth noting that we did not use the entire 'wikisql' dataset for the pre-training process because of the local environment constraints, we instead use '2252' rows from the training set of the 'wikisql' dataset.

2.1.2 Model Configuration

In line with the GPT [**?**] paper, our approach strictly adheres to a decoder-only model architecture, where multiple decoder layers are stacked on top of each other. This architectural choice mirrors the GPT model design, ensuring our language model focuses on generating text and understanding language, much like the original GPT model. The training environment for this phase is set on an M1 chip MacBook, utilizing 'mps' as the selected GPU. This decision reflects our commitment to exploring efficient training methods, consistent with the resource-efficient approach championed in the GPT paper.In order to ensure a close adherence to the GPT architecture the following points were considered:

- Transformer Block: The Transformer block (Block class) is a key component of the model. It consists of a multi-head self-attention mechanism (sa) and a feed-forward neural network (ffwd). The block applies self-attention to the input, adds the result to the original input (residual connection), normalizes the sum, and then applies the feed-forward network.

- GPT Model: The GPT model (GPT class) is the main model class. It includes:

  - Token embedding table: This is a lookup table that transforms input tokens into embeddings.
  - Position embedding table: This is a lookup table that provides embeddings for the position of each token in the input sequence.
  - Blocks: These are the Transformer blocks. The model contains a sequence of these blocks.
  - Final layer normalization: This is applied after the last Transformer block.
  - Language model head: This is a linear layer that transforms the output of the model into logits for each token in the vocabulary.

  The forward method of the GPT model computes the forward pass of the model. It adds token and position embeddings, applies the Transformer blocks, applies the final layer normalization, and computes the logits with the language model head. If targets are provided, it also computes the cross-entropy loss.

  The generate method generates new tokens conditioned on the provided input (idx). It repeatedly applies the model to the current input, samples a new token from the output distribution, and appends the new token to the input.

- Training Loop: The training loop iterates over a specified number of iterations. In each iteration, it gets a batch of training data, computes the model's logits and loss,

performs backpropagation, and updates the model's parameters using the AdamW optimizer.

The model's state is saved to a file after training.

- Model Hyperparameters:
  - Number of batches (16): This is the number of independent sequences to process in parallel.
  - Context size (32): This is the maximum context length.
  - Maximum iterations (5000): This is the total number of training iterations the model will perform.
  - Evaluation interval (100): This is the number of training iterations between each evaluation of the model.
  - Learning rate (1e-3): This is the step size used when updating the model's parameters during training.
  - Evaluation iterations (200): This is the number of iterations for which the loss is estimated during evaluation.
  - Embedding size (64): This is the size of the vector used to represent each token in the input.
  - Number of heads (4): This is the number of attention heads in the multi-head attention mechanism.
  - Dropout rate (0.0): The probability of zeroing out activations in the dropout layers for regularization.
- Text Generation After training, the model can generate new text. The generate method is used for this purpose. It takes an initial context and generates a sequence of new tokens. The generated tokens are then decoded back into text.

2.2 Fine-tuning an existing LLM

In this phase we choose an existing LLM (TinyPixel/Llama-2-7B-bf16-sharded) model to analyze and understand when and how finetuning could be really useful to consumers and on what grounds should a consumer consider choosing finetuning over pretraining an LLM for a specefic task from scratch.

2.2.1 Model Selection

To begin this phase, we opt for a substantial language model, the llama-2 7B parameters model. This model choice was made based on the preliminary analysis and experimentation done to ensure fine-tuning of the biggest possible LLM, considering the environmental resources available. The chosen model was sourced from the HuggingFace library and a conscious choice of using a sharded llama-2 model was made, as it would make it efficient to optimize the training process across multiple GPUs.

2.2.2 Dataset Selection

For the fine-tuning process, we select the same dataset as chosen during the pre-training phase (wikisql), sourced from Hugging Face. This dataset will prove instrumental in further refining the responses generated by the language model, making it particularly suitable for our objective.

2.2.3 QLORA Fine-Tuning Implementation

In the implementation phase, we employ the state of the art QLORA fine-tuning technique on the chosen Falcon 7B parameters model. To facilitate this process, we make use of the Hugging Face Transformers library. Specifically, we harness the newly introduced BitsAndBytesConfig class from the Transformers library, which complements the fine-tuning process.

- Configuration: The model 'TinyPixel/Llama-2-7B-bf16-sharded' is loaded with a 'BitsAndBytesConfig' for 4-bit precision.
- Tokenizer: Loaded from the pretrained model with padding token set to EOS token.
- Trainable Parameters: A function prints the number of trainable parameters.
- Model Preparation: The model is prepared for k-bit training and gradient checkpointing.
- Lora Configuration: Set up with parameters 'lora_alpha=16', 'lora_dropout=0.1', and 'r=64'.
- Generation Configuration: Set up with parameters 'max_new_tokens=80', 'temperature=0.7', 'top_p=0.7', 'num_return_sequences=1', 'pad_token_id=tokenizer.eos_token_id', and 'eos_token_id=tokenizer.eos_token_id'.
- Prompt Generation and Model Inference: A prompt is generated and model inference is run.
- Data Preparation and Training: Data is prepared, tokenized, and model is trained with specific arguments.
- Model Saving and Loading: The trained model is saved and loaded from the saved location.

3 Outcomes and evaluation

3.1 Pre-training

he model has approximately 0.229466 million parameters. This is a relatively small number, which means the model is lightweight and can be trained more efficiently, especially when using MPS for GPU acceleration.

The final training loss is 0.8993 and the validation loss is 1.2195. Cross entropy loss measures the performance of a classification model whose output is a probability value

between 0 and 1. In the context of a language model, a lower cross entropy loss means the model is better at predicting the next token in the sequence.

To put it in perspective, if we convert these loss values into perplexity, we get a training perplexity of approximately 2.46 and a validation perplexity of approximately 3.38. This means that, on average, the model was "uncertain" among about 2-3 choices per word during training and validation.

When prompted with "SELECT COUNT", the model generates text that resembles SQL statements. However, the generated text does not form valid SQL queries and appears to be gibberish. This suggests that while the model has learned some structure of SQL, it may need further fine-tuning on SQL data or a more specific prompt to generate valid SQL queries.

3.2 Fine-tuning

The fine-tuning of the Language Model (LLM) TinyPixel/Llama-2-7B-bf16-sharded using QLORA resulted in significant improvements in the model's performance. Prior to fine-tuning, the LLM already had a basic understanding of SQL text. However, the fine-tuning process enhanced the model's contextual understanding, enabling it to generate actual SQL text from the training set when provided with English text.

The effectiveness of the fine-tuning process was quantitatively demonstrated by a reduction in cross entropy loss from 1.419 to 0.756. This substantial decrease in loss indicates that the model's predictions became more accurate after fine-tuning, suggesting that the model had indeed gained a deeper understanding of the context.

The training was conducted on a freely available T4 GPU utilizing CUDA, demonstrating the feasibility of this approach even with limited resources.

To evaluate the performance of the fine-tuned model, a function generate_response was implemented. This function takes an English question as input, generates a prompt, and runs model inference on this prompt. In the test case where the prompt was "Tell me what the notes are for South Australia", the model was able to generate a satisfactory response, further validating the success of the fine-tuning process.

3.3 Pre-training vs Fine-tuning

Upon examining the generated SQL text, it can be concluded that fine-tuning method could be useful for creating a task specific LLM, given the limited resources constrains; whereas, the pre-training process takes a significant computational power and does not generate very accurate SQL text.

4 Contributions

- Pre-training architecture implementation: Dibyanshu Chatterjee and Rishabh Arora
- Fine-tuning architecture implementation: Apeksha Kulkarni and Snehith Reddy
- Fine-tuning evaluation: Dibyanshu Chatterjee and Rishabh Arora
- Pre-training evaluation: Apeksha Kulkarni and Snehith Reddy