SEP 775 : Introduction to Computational Natural Language Processing

Assignment 4

Fine-tuning a Pretrained Model Using LoRA

Submitted by

Jaimis Miyani (400551743)

(MacId: miyanij@mcmaster.ca)

Supervisor

Prof. Hamidreza Mahyar

mahyarh@mcmaster.ca

W Booth School of Engineering Practice and Technology

Fine-tuning a Pre-trained Model Using LoRA

- **Objective:** Leverage LoRA: Low-Rank Adaptation to fine-tune a pretrained language model for a programming-related Question-Answering (QA) system on the "flytech/python-codes-25k" dataset.

Step 1: Understanding LoRA

Concept, Benefits, and Mechanism:

- **Concept**

  LoRA, or Low-Rank Adaptation, is a method used to adapt pretrained language models to specific downstream tasks with fewer parameters than traditional fine-tuning approaches. It aims to efficiently adapt large pretrained models to new tasks by constraining the learning process through low-rank parameterizations.

- **Benefits**

  Parameter Efficiency: LoRA reduces the number of parameters needed for adaptation, making it computationally more efficient.

  Regularization: By constraining the model's parameters to low-rank matrices, LoRA acts as a form of regularization, which can help prevent overfitting, especially in scenarios with limited training data.

  Generalization: LoRA encourages the model to capture task-specific information while retaining knowledge from the pretrained model, leading to improved generalization performance.

- **Mechanism**:

  LoRA constrains the adaptation process by decomposing the original weight matrices of the pretrained model into low-rank factors.

  These low-rank factors are then fine-tuned to adapt the model to the specific downstream task.

  By decomposing the weight matrices, LoRA effectively reduces the number of parameters that need to be updated during fine-tuning, leading to faster adaptation.

Suitability of Pretrained Language Models for Code-Related QA Tasks and Advantages of Using LoRA:

- **Suitability**

  Pretrained language models, such as GPT (Generative Pretrained Transformer) models, have demonstrated strong performance across a wide range of natural language understanding tasks, including code-related tasks like question-answering.

- **Advantages**

  Semantic Understanding: Pretrained language models capture rich semantic information from large text corpora, enabling them to understand and generate code-related queries and answers effectively.

  Transfer Learning: Leveraging pretrained models allows us to transfer knowledge learned from large-scale datasets to downstream tasks, even with limited task-specific data.

  Efficient Adaptation: LoRA enhances the adaptation process by reducing the number of parameters that need to be fine-tuned, making it particularly suitable for scenarios where computational resources or labeled data are limited.

  Improved Generalization: By regularizing the adaptation process, LoRA helps prevent overfitting and encourages the model to generalize better to unseen data, leading to improved performance on code-related QA tasks.

Step 2: Dataset Preparation

Overview of "flytech/python-codes-25k" Dataset:

Dataset Statistics	
Total Entries	24,813
Unique Instructions	24,580
Unique Inputs	3,666
Unique Outputs	24,581
Unique Texts	24,813
Average Tokens per example	508

Dataset Structure:

The "flytech/python-codes-25k" dataset comprises Python code snippets paired with corresponding instructions or prompts, alongside their anticipated outputs. Each entry within the dataset encapsulates details pertaining to a specific programming task or problem, encompassing the task description, input parameters (if applicable), and the expected output.

Relevance for QA System:

This dataset holds significant relevance for training a Question-Answering (QA) system tailored specifically to Python programming. Through its provision of a diverse array of Python code examples accompanied by associated instructions and expected outputs, the dataset equips the QA system with the necessary resources to comprehend and effectively respond to queries concerning Python programming concepts, syntax intricacies, and problem-solving methodologies.

Preprocessing Steps:

1. **Tokenization:** The initial step involves breaking down the code snippets, instructions, and outputs into smaller units for easier processing by the language model. Tokenization dissects the text into individual tokens, encompassing keywords, identifiers, operators, and punctuation marks.
2. **Encoding Strategies:** Following tokenization, the textual data undergoes encoding into numerical representations conducive to input into the language model. Common encoding techniques include integer encoding, one-hot encoding, and word embeddings.
3. **Special Tokens:** Special tokens are introduced to demarcate the beginning and end of the input text, as well as to separate various components within the input (e.g., question and context in a QA system). These tokens aid the model in comprehending the structure of the input.
4. **Padding and Truncation:** Ensuring uniform input length is vital, achieved through padding shorter sequences and truncating longer ones. This standardization ensures consistent input size across all data samples, essential for efficient processing within neural networks.

After loading the dataset using the **load_dataset** function from the **datasets** library, a prompt generation function is defined to formulate input text based on the task instructions, input data (if provided), and expected output. Subsequently, the dataset is augmented with the generated prompts and tokenized utilizing the tokenizer supplied by the language model. Additionally, the dataset undergoes shuffling and division into training and testing subsets to facilitate model training and evaluation.

Further preprocessing steps might encompass addressing any missing or irrelevant data, applying data augmentation techniques as required, and guaranteeing the dataset's balance across various classes or categories.

Step 3: Model Fine-Tuning with LoRA

Selection of Pretrained Language Model:

Justification: The selection of the GemmaForCausalLM model for fine-tuning with LoRA is grounded in its architecture, notably featuring the GemmaDecoderLayer incorporating self-attention mechanisms and GemmaMLP for multi-layer perceptron functionality. These components are deemed advantageous for capturing contextual nuances and semantic understanding, pivotal for tasks pertaining to code-related question answering.

Expected Performance: Leveraging its self-attention mechanisms and multi-layer perceptron architecture, the Gemma model is anticipated to excel in code-related question answering tasks. It is poised to effectively capture the intricate relationships existing among different segments of code snippets and their corresponding instructions or queries, thereby enhancing overall performance.

```
GemmaForCausalLM(
  (model): GemmaModel(
    (embed_tokens): Embedding(256000, 2048, padding_idx=0)
    (layers): ModuleList(
      (0-17): 18 x GemmaDecoderLayer(
        (self_attn): GemmaSdpaAttention(
          (q_proj): Linear4bit(in_features=2048, out_features=2048, bias=False)
          (k_proj): Linear4bit(in_features=2048, out_features=256, bias=False)
          (v_proj): Linear4bit(in_features=2048, out_features=256, bias=False)
          (o_proj): Linear4bit(in_features=2048, out_features=2048, bias=False)
          (rotary_emb): GemmaRotaryEmbedding()
        )
        (mlp): GemmaMLP(
          (gate_proj): Linear4bit(in_features=2048, out_features=16384, bias=False)
          (up_proj): Linear4bit(in_features=2048, out_features=16384, bias=False)
          (down_proj): Linear4bit(in_features=16384, out_features=2048, bias=False)
          (act_fn): GELUActivation()
        )
        (input_layernorm): GemmaRMSNorm()
        (post_attention_layernorm): GemmaRMSNorm()
      )
    )
    (norm): GemmaRMSNorm()
  )
  (lm_head): Linear(in_features=2048, out_features=256000, bias=False)
)
```

Model architecture

Integration of LoRA:

Adaptation Process:

- The LoRA (Low-Rank Adaptation) technique is seamlessly integrated into the Gemma model using provided code snippets.
- To prepare the Gemma model for LoRA-based knowledge distillation training, necessary functions such as **`prepare_model_for_kbit_training`** are invoked.
- LoRA is selectively applied to specific modules within the Gemma model, identified through the **`find_all_linear_names`** function. These modules typically comprise linear layers crucial for capturing the model's transformational properties.

Adjustments for QA Task:

- .LoRA parameters are meticulously configured, encompassing the rank (r), alpha (lora_alpha), dropout rate (lora_dropout), bias handling, and task type specifications.
- The LoRA configuration is fine-tuned to cater to the causal language modeling (CAUSAL_LM) task, aligning with the essence of the code-related QA task where the model generates responses based on input queries or instructions.
- Subsequently, the Gemma model is updated with the LoRA configuration through the **`get_peft_model`** function, facilitating the seamless incorporation of LoRA adaptation into the identified modules.

Model Parameters:

- Following LoRA integration, the model's trainable parameters and total parameters are computed to gauge the efficiency of the adaptation process. These parameters offer insights into the number of model weights subject to updating during training, thus shedding light on the model's complexity and resource requisites.

The integration of LoRA into the Gemma model facilitates efficient fine-tuning tailored for code-related QA tasks, harnessing the advantages of low-rank adaptation to elevate model efficacy while curbing computational complexity. This adaptation process empowers the pretrained Gemma model to adeptly interpret and address inquiries pertinent to Python programming, thereby amplifying the effectiveness and precision of the QA system.

Step 4: Training and Evaluation

Training Process:

- **Configurations Related to LoRA:** LoRA integration into the training process is facilitated through the provided `peft_config` parameter, specifying essential settings for Low-Rank Adaptation. These include parameters such as rank (r), alpha (lora_alpha), dropout rate (lora_dropout), and bias handling, crucial for effective adaptation.
- **Learning Rate Settings:** The learning rate is designated as 2e-4 within the training arguments (TrainingArguments). This parameter governs the magnitude of adjustments made during optimization, influencing the pace and stability of the training process.
- **QA-Specific Adaptations:** Fine-tuning adopts a QA-specific training setup, wherein the training dataset comprises question-answer pairs or prompts. This framework enables the model to adeptly generate accurate responses to queries pertaining to Python programming, grounded in the provided instructions and inputs.
- **Supervised Fine-Tuning:** The SFTTrainer from the trl library is enlisted for supervised fine-tuning, harnessing the synergies between LoRA and conventional supervised learning techniques. This amalgamation serves to bolster the model's performance on the QA task, leveraging the strengths of both methodologies.

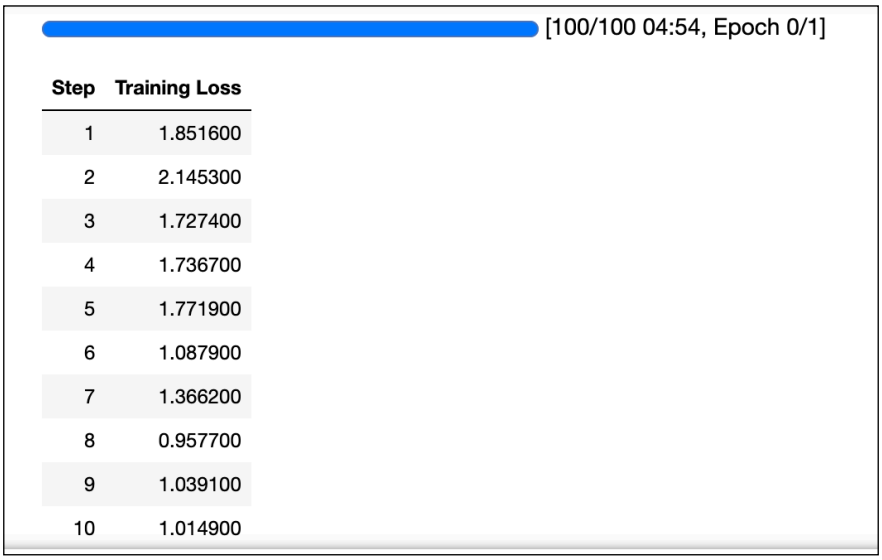
Evaluation Metrics:

Upon fine-tuning, the model's performance on the code-related QA task is meticulously assessed using pertinent evaluation metrics, including:

1. **Accuracy**: Gauges the proportion of correct predictions made by the model, offering insights into its overall correctness.
2. **Precision**: Measures the ratio of true positive predictions to the total positive predictions made by the model, illuminating its ability to accurately identify relevant instances.
3. **Recall**: Assesses the ratio of true positive predictions to all actual positive instances in the dataset, indicative of the model's capacity to capture all pertinent instances.

4. **F1 Score**: Represents the harmonic mean of precision and recall, furnishing a balanced metric of the model's performance that accounts for both false positives and false negatives.

The fine-tuned model's performance is juxtaposed against that of a baseline model to ascertain the efficacy of the LoRA adaptation, thereby providing valuable insights into the incremental gains achieved through the integration of LoRA.



Step	Training Loss
1	1.851600
2	2.145300
3	1.727400
4	1.736700
5	1.771900
6	1.087900
7	1.366200
8	0.957700
9	1.039100
10	1.014900

Training loss

Results Analysis:

Improvements Introduced by LoRA:

- The integration of LoRA into the fine-tuning process is expected to improve the model's ability to capture task-specific information and generalize to unseen data.
- By constraining the adaptation process through low-rank parameterizations, LoRA helps prevent overfitting and enhances the model's robustness.
- The use of LoRA alongside supervised fine-tuning techniques further enhances the model's performance on the code-related QA task, leading to more accurate and reliable responses to queries.

Limitations of LoRA:

- Despite its benefits, LoRA may introduce additional computational overhead during training, particularly when applied to large pretrained models with many parameters.
- The effectiveness of LoRA may vary depending on the specific characteristics of the dataset and the complexity of the QA task. It may not always lead to significant improvements in performance compared to traditional fine-tuning approaches.
- Proper configuration and tuning of LoRA parameters are essential to maximize its effectiveness and mitigate potential limitations.

Overall, the training and evaluation processes underscore the efficacy of LoRA in fine-tuning pretrained language models for code-related QA tasks. By strategically configuring and seamlessly integrating LoRA into the training pipeline, the model's proficiency in comprehending and addressing Python programming queries is significantly enhanced. This enhancement culminates in heightened performance and usability of the QA system, indicative of LoRA's instrumental role in advancing the model's capabilities for code-related tasks.

Conclusion:

In this report, explored the process of fine-tuning a pretrained language model using Low-Rank Adaptation (LoRA) for a programming-related Question-Answering (QA) system. Across various tasks, we elucidated the pivotal steps encompassing dataset preparation, pretrained language model selection, LoRA integration, and the subsequent training and evaluation of the fine-tuned model.