Developing a Local Python Programming Interface Using Open-Source LLMs

Bin Zubair, Muhammed Saneens [1] **Pai, Pratiksha** [1]

Abstract

This project aims at a local interface for Python programming assistance, built on the cutting-edge LLaMa-2 7 Billion parameter model, leveraging open-source Large Language Models (LLMs). Incorporating advanced techniques such as supervised fine tuning (SFT), Parameter Efficient Fine Tuning (PEFT), and Quantized Low-Rank Adaptation (QLoRA), we aim to offer a solution that is not only efficient and cost-effective but also highly customizable to suit diverse programming needs.

The key to our interface is allowing interactions in natural language to give immediate python coding assistance. This approach significantly lowers the entry barrier for beginners, fostering a conducive learning environment, while for experts, it serves as a quick and effective tool to streamline their coding process. As a result, our system stands as a versatile and accessible platform, making advanced AI assistance more available and user-friendly.

Code can be found on our GitHub

Keywords: *Fine Tuning, Parameter Efficient Fine Tuning (PEFT), Quantized Low-Rank Adaptation (QLoRA), Supervised Learning, Transformer Models, Large Language Models (LLMs), LLaMa-7B.*

1. Introduction

Large Language Models (LLMs) like GPT-3 have significantly advanced the field of AI, especially in understanding and generating natural language. However, their reliance on external APIs for functionality and the need for a constant internet connection limit their wider use. Our project tackles these issues by developing a local interface for Python programming help, using the LLaMa-2 7 Billion parameter model, a leading model in the open-source LLM space. This approach offers a more user-friendly and efficient alternative, particularly useful for those who need reliable assistance without internet access.

[*]Equal contribution [1]Department of Eelectrical and Computer Engineering, Georgia Tech, US.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

We chose the LLaMa-2 (1) model for its state-of-the-art performance, although we were limited to the 7 Billion parameter version due to our project's hardware constraints. Despite this, LLaMa-2 proves to be highly effective for our needs, making it an ideal choice for Python programmers looking for quick and local help. With this project, we hope to pave the way for more local and accessible AI tools in the world of programming.

2. Methodology

Our methodology for fine-tuning the LLaMa-2 base model for Python programming assistance encompassed a series of strategically designed steps. Initially, we established a robust foundation using a pre-trained LLaMa-2 model, followed by employing supervised fine-tuning (2) with a curated Python 18k instruction dataset (3). The process was further refined by implementing of a quantized low-rank adapter, resulting in an optimized fine-tuning phase tailored to enhance the model's effectiveness in coding assistance tasks.

2.1. Base Model

For our project, we utilized the state-of-the-art LLaMa-2 7 Billion parameter model developed by Meta. This model, inherently designed as a document completer, is based on an optimized transformer architecture, making it an ideal foundation for our programming assistant. LLaMa-2 models are known for their auto-regressive nature, capable of generating coherent and contextually relevant text, which is crucial for programming assistance.

The version we adopted (4) had been pre-trained on chat-based instructions by NousResearch, aligning it closer to our use case. This pre-training was pivotal as it provided a base model already oriented towards interactive, chat-based scenarios, a key aspect of our programming assistant's functionality.

Incorporating this model into our system involved leveraging the Hugging Face transformers library (5). The library's compatibility and extensive support for transformer models, including various tools and features for model manipulation and deployment, significantly streamlined our development process.

2.2. Dataset & Supervised Fine Tuning

In the Supervised Fine Tuning (SFT) phase, we utilized the python_code_instructions_18k_alpaca dataset (3), distinguished by its clear structure with prompts containing instructions, tasks, and input/output samples, crucial for a Python-centric training approach. This dataset's concise yet comprehensive nature enabled us to refine the model's ability to understand and generate Python code without requiring an excessively large corpus.

Fine-tuning was conducted using a domain-specific training script that iteratively processed the dataset in batches, optimizing the model's parameters for the targeted task of Python programming assistance. The effectiveness of this process was quantified by a near-zero loss parameter at the completion of training cycles, indicating a high level of model precision. The resulting model displayed a marked improvement, consistently generating syntactically and logically correct Python code

2.3. Loading Pretrained model

For loading our pre-trained model for local deployment, we used a quantization strategy using the BitsAndBytes library to compress the LLaMa 2 model's weights into a 4-bit format. This process, known as NF4 quantization, normalizes and then quantizes the weights, significantly reducing memory requirements while maintaining a fine balance between storage efficiency and model precision. This technique allowed us to decrease the model's footprint from 14 GB to a mere 3.5 GB, making it viable for our A100 GPUs with restricted memory capabilities. The conversion of these quantized weights back to float16 during computation preserves the model's performance, ensuring that its capabilities remain the same even with the reduced precision.

2.4. Parameter Efficient Fine-Tuning

For fine-tuning, we employed Parameter-Efficient Fine Tuning (PEFT), which modifies a small subset of the model's parameters, allowing us to finely adjust the LLaMa 2 model for Python programming tasks without the overhead of full-model training. PEFT's selective updating process—particularly the alteration of specific delta weights, ensures that the model remains lean and manageable, with each set of adjustments resulting in a different fine-tuned model variant. This method conserves computational resources while retaining the model's extensive language understanding, making it a practical choice for specialized domains where large-scale LLMs must adapt to niche tasks with efficiency.

2.5. Block-wise k-bit Quantization

Quantization involves converting data from a format that holds a larger amount of information into one that holds less. This typically involves changing a data type with more bits to one with fewer bits, such as transforming 32-bit floating-point numbers into 8-bit integers. To make full use of the reduced bit range, the original data is usually scaled to fit within the range of the new data type. This scaling is done by normalizing the data based on its absolute maximum value. The data is often structured in a tensor. For instance, when a tensor in 32-bit floating-point (FP32) format is quantized, it is converted into an 8-bit integer (Int8) tensor, which can represent values in the range of $[-127, 127]$.

The quantization process is defined by the equation:

$$X_{Int8} = \text{round}\left(\frac{127}{\text{absmax}(X_{FP32})} \times X_{FP32}\right), \quad (1)$$

where $\text{absmax}(X_{FP32})$ is the absolute maximum value of X_{FP32}, and $\frac{127}{\text{absmax}(X_{FP32})}$ is the quantization constant c_{FP32}.

Dequantization, the inverse process, is represented by:

$$\text{dequant}(c_{FP32}, X_{Int8}) = \frac{X_{Int8}}{c_{FP32}} = X_{FP32}. \quad (2)$$

2.6. Low-rank Adapter Finetuning (LoRA)

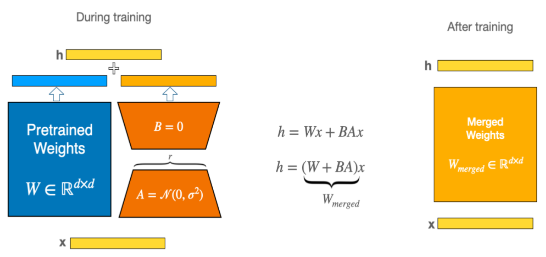

During training
h
Pretrained Weights
$W \in \mathbb{R}^{d \times d}$
$B = 0$
r
$A = \mathcal{N}(0, \sigma^2)$
x
$h = Wx + BAx$
$h = \underbrace{(W + BA)}_{W_{merged}}x$
After training
h
Merged Weights
$W_{merged} \in \mathbb{R}^{d \times d}$
x

Figure 1. LoRA

We decided to build on LoRA (8). According to the research by Aghajanyan et al. (2020)(6), pre-trained language models maintain a low "intrinsic dimension" and remain efficient in learning, even when projected into a smaller subspace. This suggests that during adaptation, the updates to the weights in a neural network might also exhibit a low "intrinsic rank". Consider a pre-trained weight matrix $W_0 \in \mathbb{R}^{d \times k}$. We propose constraining its updates via a low-rank decomposition:

$$W_0 + \Delta W = W_0 + BA, \quad (3)$$

where $B \in \mathbb{R}^{d \times r}$, $A \in \mathbb{R}^{r \times k}$, and rank $r \leq \min(d, k)$.

In this methodology, W_0 remains unchanged during training, while A and B are trainable. The output for an input x in the modified forward pass is given by:

$$h = W_0x + \Delta Wx = W_0x + BAx. \quad (4)$$

This reparameterization is initially set with a random Gaussian initialization for A and zero for B, making $\Delta W = BA$ start at zero. The product ΔWx is scaled by α/r, with α being a constant. This approach, as discussed by Yang & Hu (2021) (7), simplifies the tuning of hyperparameters when varying the rank r. Therefore, α is set to the initial value of r and is not further adjusted.

The concept of fine-tuning in neural networks often involves selectively training certain pre-trained parameters. LoRA extends this idea by not insisting that gradient updates to weight matrices during adaptation be of full rank. Essentially, when LoRA is applied to all weight matrices and all biases are trained, it can approximate the effectiveness of complete fine-tuning. This is achieved by setting the LoRA rank r equal to the rank of the pre-trained weight matrices.

The final weight matrix W, which is the sum of the initial weight matrix W_0 and the product BA, can be precomputed and stored for use during inference. Both W_0 and BA have dimensions $R^{d\times k}$. If there's a need to adapt the model for a different downstream task, W_0 can be easily retrieved by subtracting BA and then incorporating a new product B_0A_0. This approach ensures that there is no extra latency during inference.

LoRA can be selectively applied to specific weight matrices within a neural network, effectively reducing the number of parameters that need to be trained. In the case of the Transformer architecture, which includes four weight matrices in the self-attention module (W_q, W_k, W_v, W_o) and two in the MLP module, this principle can be particularly useful. We consider W_q (as well as W_k and W_v) as a single matrix with dimensions $d_{\text{model}} \times d_{\text{model}}$, despite the fact that its output dimension is typically divided into several attention heads. For our research, we focus exclusively on adapting the attention weights for downstream tasks, while keeping the MLP modules static (i.e., they do not undergo training for these tasks).

2.7. QLORA

QLORA (9) achieves effective 4-bit fine-tuning by employing two novel techniques: 4-bit NormalFloat (NF4) quantization and Double Quantization. It uses two types of data: one for low-precision storage (typically 4-bit) and another for computations (commonly BFloat16). In practical terms, this means that when a QLORA weight tensor is needed, it's first converted from 4-bit to BFloat16, and then 16-bit matrix multiplication is performed.

The NormalFloat (NF) data type is an extension of Quantile Quantization, a method known for being optimally efficient in terms of information theory. It ensures an even distribution of values across quantization bins from the input tensor. This is achieved by estimating the input tensor's quantiles using its empirical cumulative distribution function.

Considering that pretrained neural network weights often follow a zero-centered normal distribution with a standard deviation of σ, we can standardize these weights to a uniform distribution. We do this by scaling σ to fit the weights within a predetermined range, set at $[-1, 1]$ for our purposes. This process involves aligning both the data type's quantiles and the network weights to this range.

QLORA is defined for a single linear layer in the quantized base model with one LoRA adapter as follows: the output in BFloat16, Y_{BF16}, is the sum of the product of the input in BFloat16, X_{BF16}, and the double dequantized weight matrix W_{NF4}, plus the product of two LoRA layers L_{BF16_1} and L_{BF16_2}. The double dequantization process, denoted as doubleDequant(), involves two stages of dequantization, first converting the 4-bit quantized weights W_{4-bit} to BFloat16, and then applying the same for the quantization constants c_{k-bit_2}.

In QLORA, NF4 is used for the weight matrix W and FP8 for the constant c_2. For achieving higher precision in quantization, a block size of 64 is used for W, and a block size of 256 is utilized for c_2 to save memory.

During parameter updates, only the gradients with respect to the error for the LoRA adapter weights ($\frac{\partial E}{\partial L_i}$) are needed, not the gradients for the 4-bit weights ($\frac{\partial E}{\partial W}$). However, computing $\frac{\partial E}{\partial L_i}$ requires the calculation of $\frac{\partial X}{\partial W}$, which is done using the initial equation with dequantization from the storage data type W_{NF4} to the computation data type W_{BF16}.

2.8. Training

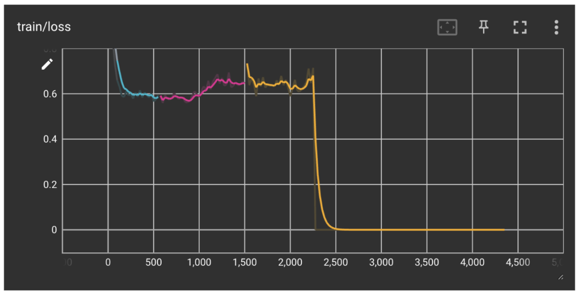


Figure 2. Learning rate decay

For fine-tuning of the LLaMA model we used 1 NVIDIA A100 GPU for 8 hours. A critical aspect was the learning rate, set initially at $2e^4$. This rate, managed by a cosine

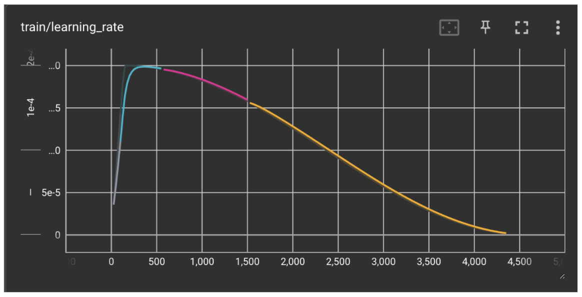


Figure 3. Change in Loss during Fine tuning phase

learning rate scheduler, facilitated rapid early learning and finer adjustments over time, as evidenced by the learning rate graph's decay pattern. The loss graph, showing an initial sharp decrease, indicates effective learning, assisted by QLoRA parameters like `lora_r = 64` and `lora_alpha = 16`, which target the model's adaptation to the Python codebase.

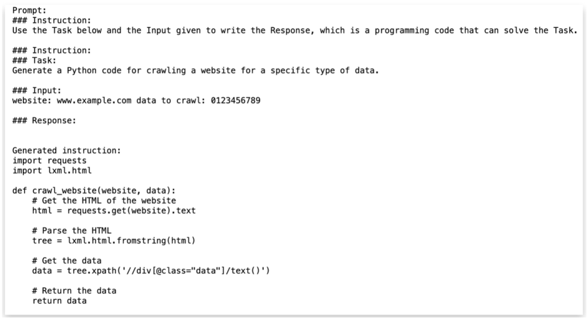
During the hyperparameter tuning stage, settings like a training batch size of 4 and `max_grad_norm` of 0.3 were key in stabilizing training and preventing gradient exploding. The use of 4-bit quantization (`use_4bit = True`) with specific compute and quantization types (`bnb_4bit_compute_dtype = "float16"` and `bnb_4bit_quant_type = "nf4"`) enhanced the model's computational efficiency. These choices were instrumental in refining the model's performance to grasp the intricacies of the Python codebase.

Finally, the fine-tuned model was obtained by merging the base model and the finetuned weights. This version was then uploaded to Hugging Face so it's easier to infer the model later.

2.9. Inference & Examples

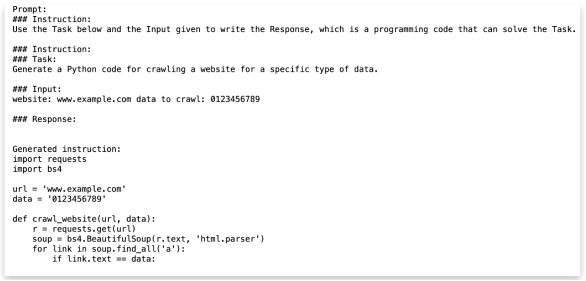
After the model has been fine-tuned and uploaded to Hugging Face, it becomes available for inference for coding tasks. The process involves downloading the model for local inference and running it using python script. A script `generate.py` loads the fine-tuned LLaMA model (10).

The generation script takes a prompt in natural language to produce Python code. The parameters, such as `max_new_tokens`, `do_sample`, `top_p`, and `temperature`, are set to control the length and variability of the generated code. Following are a few examples that show model's versatality and use.



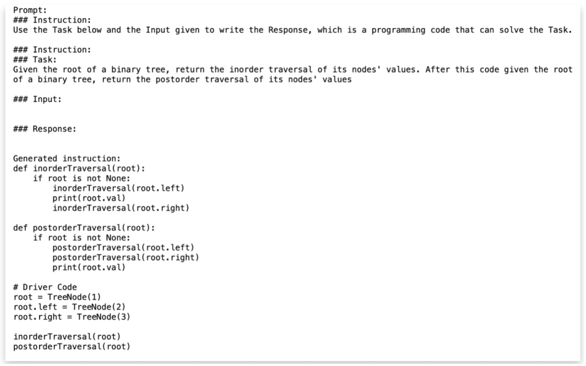
```
Prompt:
### Instruction:
Use the Task below and the Input given to write the Response, which is a programming code that can solve the Task.

### Instruction:
### Task:
Generate a Python code for crawling a website for a specific type of data.

### Input:
website: www.example.com data to crawl: 0123456789

### Response:


Generated instruction:
import requests
import lxml.html

def crawl_website(website, data):
    # Get the HTML of the website
    html = requests.get(website).text

    # Parse the HTML
    tree = lxml.html.fromstring(html)

    # Get the data
    data = tree.xpath('//div[@class="data"]/text()')

    # Return the data
    return data
```

Figure 4. Example 1: with one task, one input



```
Prompt:
### Instruction:
Use the Task below and the Input given to write the Response, which is a programming code that can solve the Task.

### Instruction:
### Task:
Generate a Python code for crawling a website for a specific type of data.

### Input:
website: www.example.com data to crawl: 0123456789

### Response:


Generated instruction:
import requests
import bs4

url = 'www.example.com'
data = '0123456789'

def crawl_website(url, data):
    r = requests.get(url)
    soup = bs4.BeautifulSoup(r.text, 'html.parser')
    for link in soup.find_all('a'):
        if link.text == data:
```

Figure 5. Example 1: with one task, one input, regenerated



```
Prompt:
### Instruction:
Use the Task below and the Input given to write the Response, which is a programming code that can solve the Task.

### Instruction:
### Task:
Given the root of a binary tree, return the inorder traversal of its nodes' values. After this code given the root
of a binary tree, return the postorder traversal of its nodes' values

### Input:


### Response:


Generated instruction:
def inorderTraversal(root):
    if root is not None:
        inorderTraversal(root.left)
        print(root.val)
        inorderTraversal(root.right)

def postorderTraversal(root):
    if root is not None:
        postorderTraversal(root.left)
        postorderTraversal(root.right)
        print(root.val)

# Driver Code
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)

inorderTraversal(root)
postorderTraversal(root)
```

Figure 6. Example 2: with multiple tasks

3. Results & Conclusions

This project successfully fine-tuned the LLaMa 7b model for local Python programming assistance. The use of open-source LLMs and innovative fine-tuning techniques like PEFT and QLoRA underlines the potential of AI in specialized domains. The project not only demonstrates the technical feasibility of such an endeavor but also provides valuable insights into the practical application of AI models in programming assistance, paving the way for more personalized and efficient AI tools.

References

[1] Meta AI. Llama 2: Open Foundation and Fine-Tuned Chat Models. *arXiv preprint arXiv:2307.09288*, 2023.

[2] Hugging Face. SFT Trainer Documentation. Available at: https://huggingface.co/docs/trl/main/en/sft_trainer.

[3] Hugging Face. Python 18k Instruction Dataset. Available at: https://huggingface.co/datasets/iamtarun/python_code_instructions_18k_alpaca.

[4] Nous Research. LLaMa-2 7B Chat Model. Available at: https://huggingface.co/NousResearch/Llama-2-7b-chat-hf.

[5] Hugging Face. Hugging Face Transformers Library. Available at: https://huggingface.co/docs/transformers/index.

[6] Armen Aghajanyan, Luke Zettlemoyer, and Sonal Gupta. Intrinsic Dimensionality Explains the Effectiveness of Language Model Fine-Tuning. *arXiv preprint arXiv:2012.13255*, 2020.

[7] Greg Yang and Edward J. Hu. Feature Learning in Infinite-Width Neural Networks. *arXiv preprint arXiv:2011.14522*, 2021.

[8] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-Rank Adaptation of Large Language Models. *arXiv preprint arXiv:2106.09685*, 2021.

[9] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, Luke Zettlemoyer. QLoRA: Efficient Finetuning of Quantized LLMs *arXiv preprint arXiv:2305.14314*, 2023.

[10] Hugging Face. Finetuned model. Available at: https://huggingface.co/pratikshapai/llama-2-7b-int4-python-code-20k.