NLP ASSIGNMENT

Submitted To

Dr. Aruna Malapati

Submitted By:

Name	ID NO
AASHUTOSH A V	2021A7PS0056H
AASHISH CHANDRA K	2021A7PS0467H
MAHESWAR BORA	2021A7PS1505H

Birla Institute of Technology and Science, Hyderabad Campus

Abstract

This project presents a novel approach to solving simple Data Structures and Algorithms problems by harnessing the capabilities of open source Large Language Models. The primary objective is to develop a holistic solution that not only generates the correct code, but also provides human-readable explanations for the generated code. We found this problem statement to be important in teaching beginner coders algorithmic problems, and allow cohesive learning of coding and logic in DSA.

The project workflow unfolds in 2 key phases, administered by 2 LLMs.

1. Code Generation: A standard Algorithms problem is fed into the first LLM. The solution is generated and subsequently tested against manual test cases. A personalized question can also be fed into the model, as long as it is a Data Structures & Algorithms problem.

2. Code Explanation: The correct code generated by the previous LLM is passed into the next one, which generates human-readable explanations of the fed code, thus making the logic and functionality of the code transparent to the user. We aim with this part of the project to enhance the overall user understanding of the code.

By combining these 2 generative models, We built a project that streamlines the problem-solving process making it more accessible and educational to beginner programmers. The synergy between these models, we hope, will bridge the gap between automated code generation and human understanding, contributing to advancement in NLP-driven education.

TASK DEFINITION:

Generate and Elucidate Code for Algorithmic Problems

Dataset

We have created a custom dataset to finetune the Code Llama to solve Leetcode Questions. We obtained the dataset by scraping the LeetCode website using Puppeteer. We used Leetcode since we believed it was the best website for generic Algorithmic Questions.

We initially found upwards of 800 question-solution pairs to work with, but since some of them had multiple solutions, we realized the model got confused with contrasting outputs for the same given input. We then decided to do away with the repetitions, and ended up with 405 total unique (question, code) combinations.

We then split out 10% of this dataset for validation purposes, to understand the performance metrics of the model.

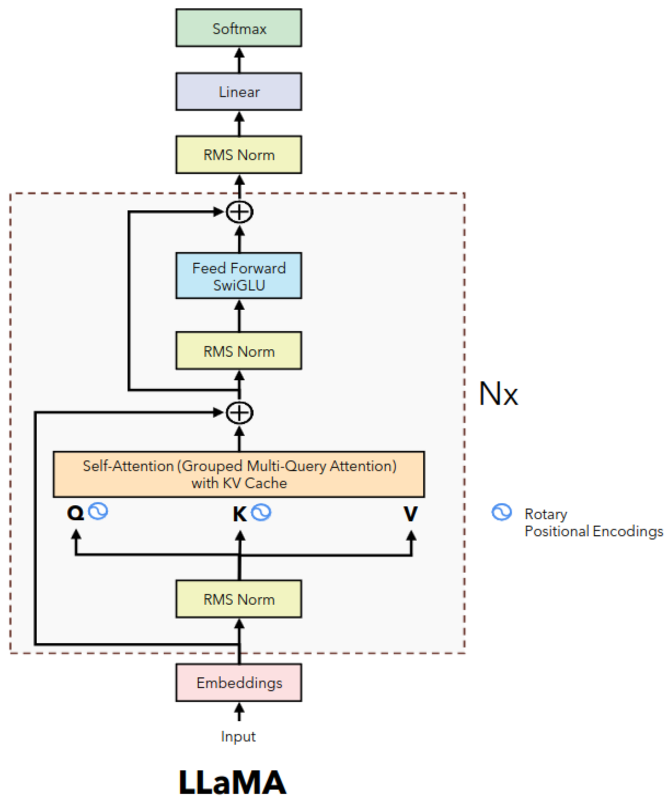
Since this is a Generation-based project, we didn't have to worry about ideas of class-imbalance, etc.

Structure of the Dataset:

A Json of (Question, Code, Solution) Triples, where the "Question" is a straightforward question input, "Code" is an incomplete tag, which we expect the Llama to fill with its solution of the problem, and "Solution", which is the actual solution for the problem. We use this Solution to benchmark the generated solution with, and then compute the metrics for evaluation of the performance of our model.

Model

LLAMA 2 Model details -



Umar Jamil - https://github.com/hkproj/pytorch-llama-notes

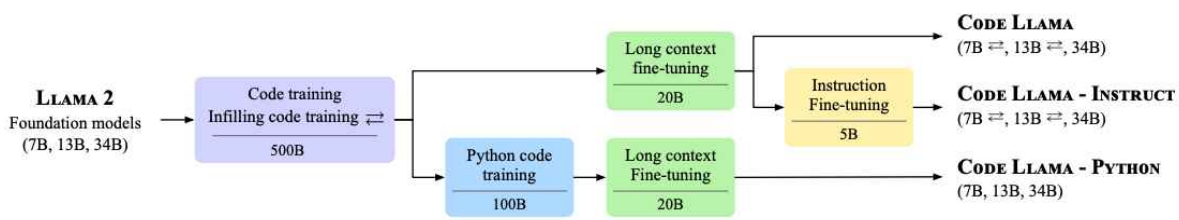
The Llama 2 architecture consists of -

1. **Standard Transformer Architecture**: Llama 2 continues to use the standard transformer architecture as introduced by Vaswani et al., 2017. This architecture typically includes layers of multi-head attention and position-wise feedforward networks.
2. **Pre-normalization using RMSNorm**: RMSNorm, introduced by Zhang and Sennrich in 2019, is used for normalization. This differs from the original transformer model, which used post-layer normalization. Pre-normalization involves applying normalization at the beginning of each sub-layer within the transformer blocks.
3. **SwiGLU Activation Function**: The SwiGLU (Switched Gated Linear Unit) activation function, proposed by Shazeer in 2020, is utilized in Llama 2. This is a variation of the GLU (Gated Linear Unit) that has shown to be effective in deep learning models.
4. **Rotary Positional Embeddings (RoPE)**: Introduced by Su et al. in 2022, RoPE is used for encoding positional information. This is a departure from the traditional positional

encoding methods used in transformers and is designed to provide more effective positional context.

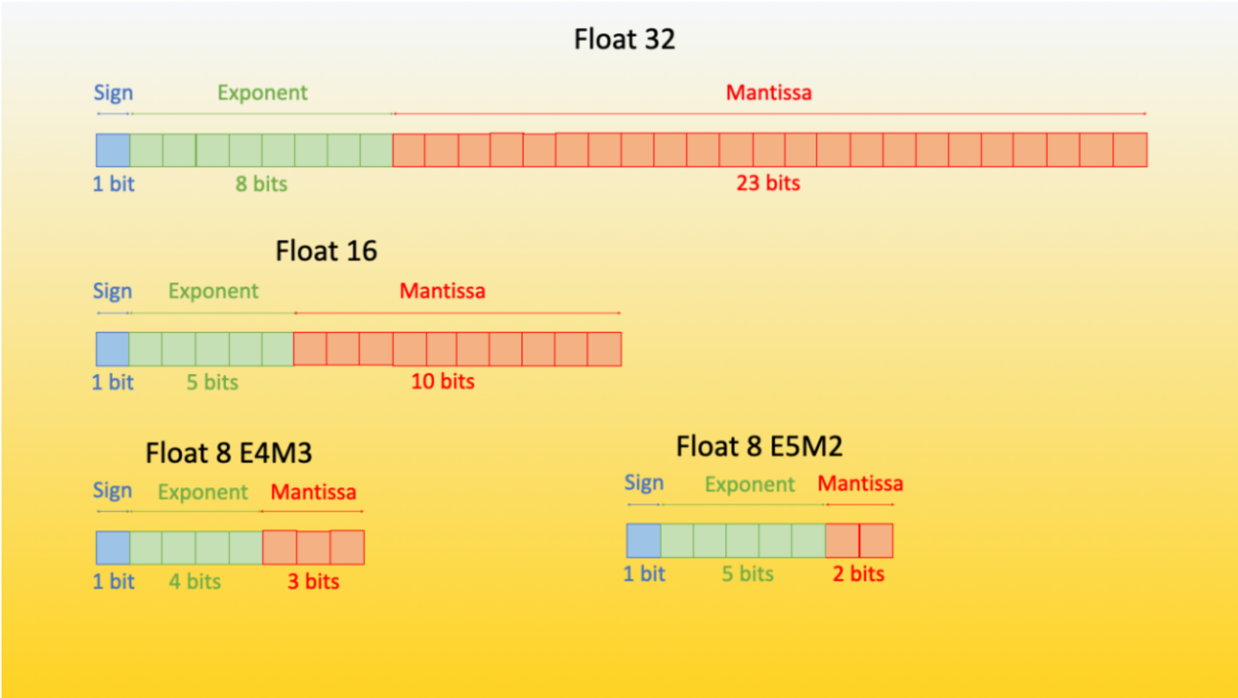
5. **Increased Context Length**: Llama 2 features an increased context length compared to Llama 1. This means the model can handle longer sequences of input data, which is crucial for understanding and generating more complex text.
6. **Grouped-Query Attention (GQA)**: This is a new feature in Llama 2, where the attention mechanism is modified to group queries. This change likely aims to improve the efficiency and effectiveness of the attention process.
7. The tokenizer used here is BPE(Byte Pair Encoding) model based on [sentencepiece]

Llama 2 is further fine tuned to give the Code Llama models:



Out of the models existing with the Llama base, we used the Code Llama instruct model for both our tasks. We tested both the 13b and the 7b models, which are basically the 13 billion parameter, and 7 billion parameter models respectively.

Using QLora for fine-tuning, we loaded the 4-bit NormalFloat, an information theoretically optimal quantization data type for normally distributed data that yields better empirical results than 4-bit Integers and 4-bit Floats.



Having learnt about Parameter-Efficient Fine-Tuning (PEFT) in the class, we implemented that in our project, since it is impractical to re-work all the parameters of the model, with them being in the count of billions.

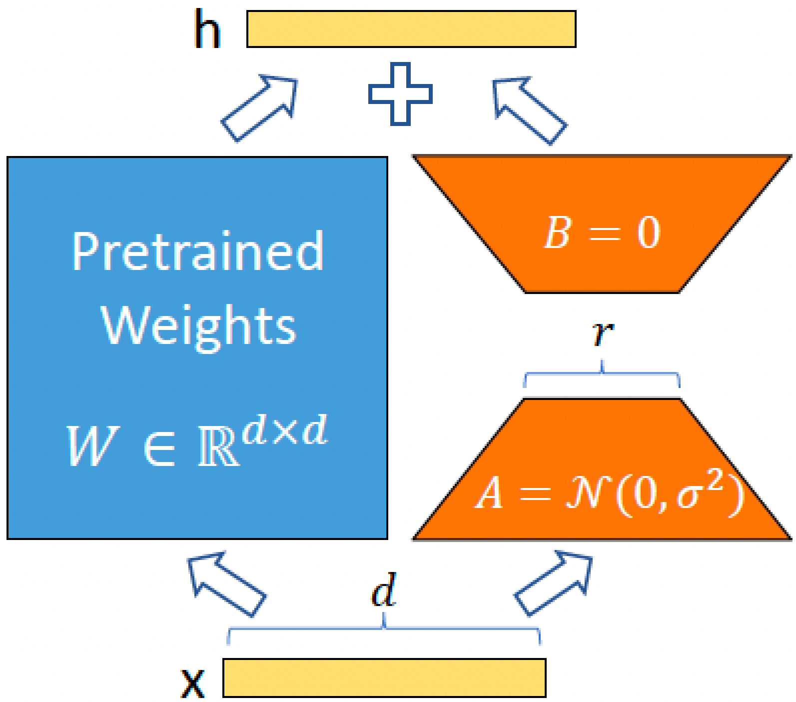


Figure 1: Our reparametrization. We only train A and B.

The parameters we worked on are mentioned under the Parameters sub-section of this report. All the parameter-tuning is happening in the first code-generation model, for we're doing prompting of the Llama Instruct model in the second task we have, of generating the explanations. Any code generated by the first Llama can be explained by prompting it to the second model, to generate a decent understandable explanation of the code generated prior.

Parameters

Our usage of Code Llama Instruct 7b and 13b implies the model is trained against 7 billion and 13 billion parameters. These models have been pre-trained on over 2 Trillion tokens, with a context length of 4096. Qlora finetunes on a fraction of these parameters.
The number of trainable parameters when fine tuning using Qlora is 262541312 or ~262 million for Llama 7 and 328258560 or ~328 million for Llama 13b.
During fine-tuning only the query and value matrices will be updated.

The Hyper-parameters used for fine tuning include:

Lora Config (PEFT config)-

1. r: the rank of the update matrices, expressed in int. Lower rank results in smaller update matrices with fewer trainable parameters. We used rank = 64 for our experiments.
2. target_modules: The modules (for example, attention blocks) to apply the LoRA update matrices. We set it to q_proj and v_proj (The query and value matrices).
3. alpha: LoRA scaling factor. We set alpha to 16 for our experiments.
4. bias: We set bias to None when finetuning
5. layers_to_transform: List of layers to be transformed by LoRA.. Set to Default, to allow all layers to transform
6. task_type: "CAUSAL_LM". Sets the task type to Casual language modelling
7. For required parameters like layers_pattern, rank_pattern, alpha_pattern, we decided to leave it at the default values defined in the original paper

Bitsandbytes config-

1. load_in_4bit=True
2. bnb_4bit_quant_type="nf4"
3. bnb_4bit_compute_dtype=torch.float16

Training hyperparameters-

per_device_train_batch_size = 1
gradient_accumulation_steps = 4
optim = "paged_adamw_32bit"
adam_beta1= 0.9
adam_beta2= 0.95
save_steps = 210
logging_steps = 10
learning_rate = 2e-4
max_grad_norm = 0.3
warmup_ratio = 0.03
lr_scheduler_type = "constant"
weight_decay= 0.001
num_train_epochs=2
fp16=True

Loss

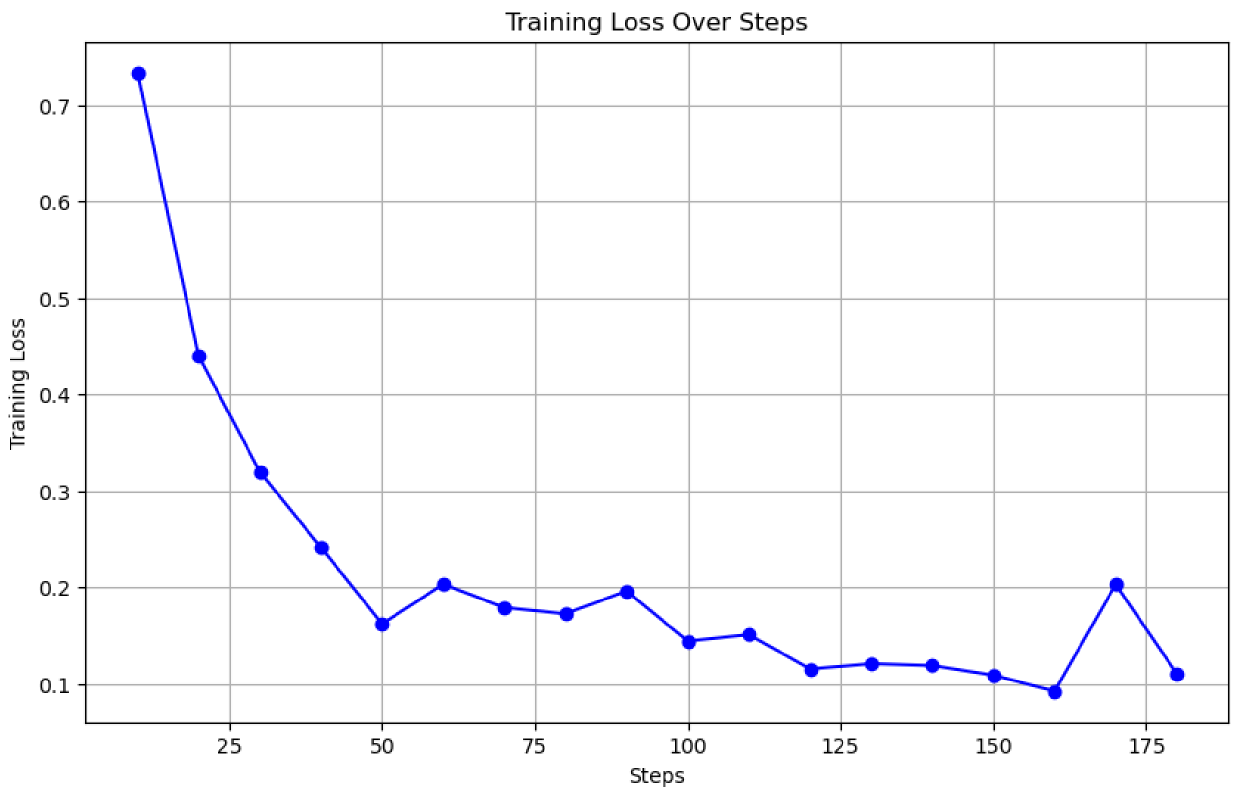
The Code Llama originally was trained using a concept called Ranking Loss:

$$L_{ranking} = -\log(\sigma(r_\theta(x, y_c) - r_\theta(x, y_r) - m(r)))$$

Here the margin "m(r)" is a discrete function of the preference rating. This works on a reward mechanism. "Yc" is the preferred response that is chosen by the annotators, and "Yr" is the rejected counterpart. The reward is given based on the difference in the output generated compared to the rejected part "Yr".

Loss after training for 180 Steps on CodeLlama-instruct-7b:

Loss after training for 180 Steps on CodeLlama-instruct-13b:



Results

CODE GENERATION EXAMPLE:

7b Generated Output:

```
class Solution {
public:
    vector<int> nextLargerNodes(ListNode* head) {
        vector<int> ans;
        stack<int> stk;
        for (ListNode* cur = head; cur != nullptr; cur = cur->next) {
            while (!stk.empty() && stk.top() < cur->val) {
                ans.push_back(stk.top());
                stk.pop();
            }
            stk.push(cur->val);
        }
        while (!stk.empty()) {
            ans.push_back(stk.top());
            stk.pop();
        }
        return ans;
    }
};
```

13b Generated Output:

```
class Solution {
public:
    vector<int> nextLargerNodes(ListNode* head) {
        vector<int> ans;
        stack<ListNode*> st;
        ListNode* curr = head;
        while (curr != NULL) {
            while (!st.empty() && st.top()->val <= curr->val) {
                st.pop();
            }
            if (!st.empty()) {
                ans.push_back(st.top()->val);
            } else {
                ans.push_back(0);
            }
            st.push(curr);
            curr = curr->next;
        }
        return ans;
    }
};
```

Our Prompt for Generating the Above 2 Solutions :

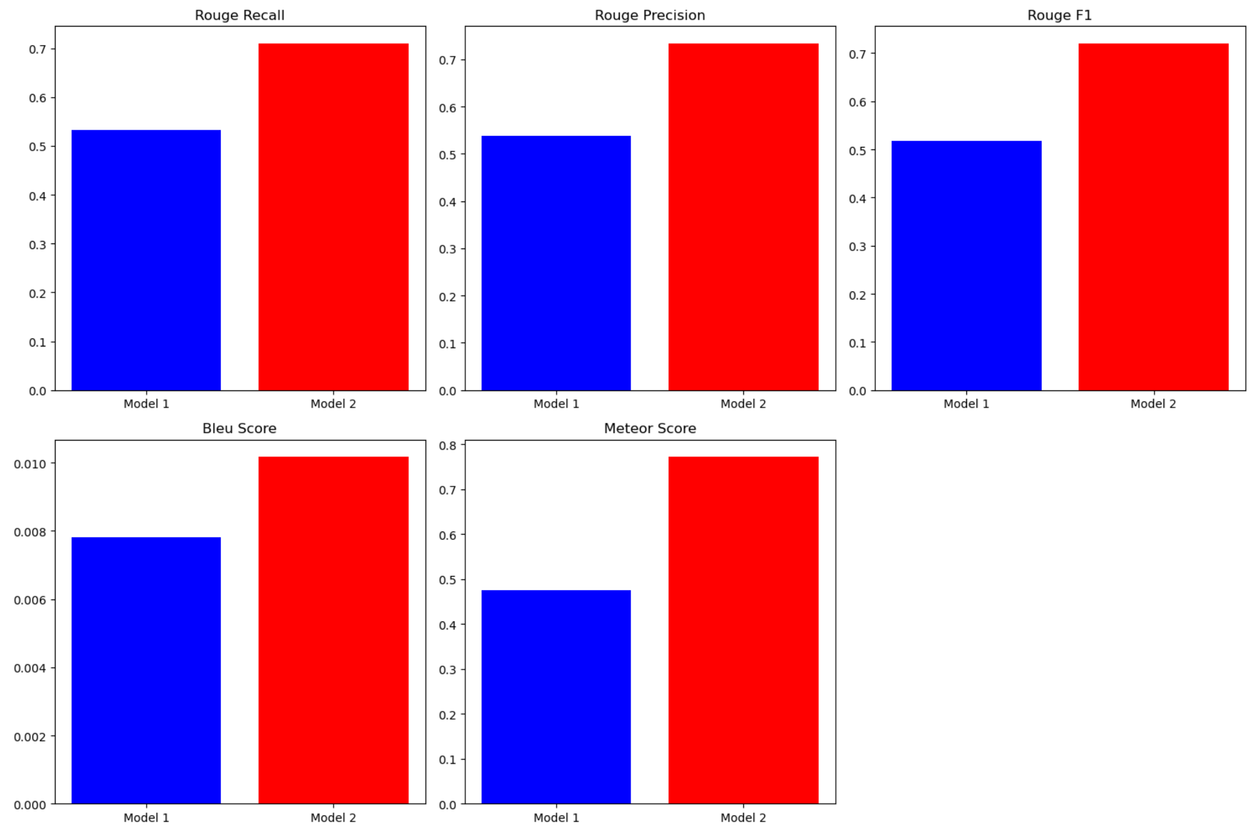
```
[INST] <<SYS>>
You are a smart assistant. You will solve the problem based on the following source
code and question.
<</SYS>>
<<QUESTION>>
Can you solve this real interview question? Next Greater Node In Linked List - You are
given the head of a linked list with n nodes.

For each node in the list, find the value of the next greater node. That is, for each
node, find the value of the first node that is next to it and has a strictly larger
value than it.

Return an integer array answer where answer[i] is the value of the next greater node
of the ith node (1-indexed). If the ith node does not have a next greater node, set
answer[i] = 0.

Example 1:
[https://assets.leetcode.com/uploads/2021/08/05/linkedlistnext1.jpg]

Input: head = [2,1,5]
Output: [5,5,0]

Example 2:
[https://assets.leetcode.com/uploads/2021/08/05/linkedlistnext2.jpg]

Input: head = [2,7,4,3,5]
Output: [7,0,5,5,0]

Constraints:
 * The number of nodes in the list is n.
 * 1 <= n <= 104
 * 1 <= Node.val <= 109
<</QUESTION>>
<<CODE>>
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next)
{}
 * };
 */
class Solution {
public:
    vector<int> nextLargerNodes(ListNode* head) {

    }
};
<</CODE>>[/INST]
<<SOLUTION>>
```

The Generated Explanation using 7b Model:
<<EXPLANATION>>
The solution is based on the idea of using a stack to keep track of the nodes that have a smaller value than the current node. The stack is used to find the next greater node for each node in the list.
The algorithm works as follows:
1. Initialize an empty stack and a pointer to the head of the list.
2. Iterate through the list and for each node, do the following:
a. If the stack is empty or the top element of the stack has a smaller value than the current node, push the current node to the stack.
b. If the stack is not empty and the top element of the stack has a larger value than the current node, pop the top element of the stack and add its value to the answer array.
3. Once the iteration is complete, if the stack is not empty, add the value of the top element of the stack to the answer array.

The Generated Explanation using 13b Model:
<<EXPLANATION>>
The solution uses a stack to keep track of the nodes in the linked list. The stack is initially empty. The algorithm iterates through the linked list, and for each node, it does the following:
1. It pops all the nodes from the stack that have a value less than or equal to the current node. This is done by using a while loop that continues until the top of the stack is a node with a value greater than the current node.
2. If the stack is not empty after popping the nodes, it means that the top of the stack is a node with a value greater than the current node. The value of this node is the next greater node for the current node. The algorithm pushes the current node onto the stack.
3. If the stack is empty after popping the nodes, it means
that the current node does not have a next greater node. The algorithm pushes the current node onto the stack and sets the next greater node

Evaluation Metrics:

Comparing the Metrics of the Fine-Tuned 13b and 7b Models for Code Generation:



Bibliography

Fine Tuning was done on kaggle 2xT4 16GB: https://www.kaggle.com/

Model weights downloaded from: https://huggingface.co/codellama

Dataset: https://leetcode.com/

Llama Recipes: https://github.com/facebookresearch/llama-recipes.git

4 bit quantization and qlora: https://huggingface.co/blog/4bit-transformers-bitsandbytes

Llama 2 paper: -https://arxiv.org/pdf/2307.09288

Code Llama paper: https://arxiv.org/abs/2308.12950

Qlora paper: https://arxiv.org/abs/2305.14314