PYTHON CODE AUTO-COMPLETER USING CODELLAMA

ABSTRACT:

Modern software development relies heavily on code autocompletion tools to enhance productivity and accuracy. Developers often use integrated development environments (IDEs) or text editors with code autocompletion features to speed up the coding process. There exists a market demand for such services as they work as a helping hand for developers.

In this project, we aim to design and implement a code auto-completer using the Llama2 model. The auto-completer will be capable of predicting the next token or sequence of tokens in each code snippet, thereby aiding developers in writing code more efficiently and accurately.  Our evaluation will focus on the accuracy of the predicted tokens in terms of their syntactic and semantic appropriateness in the given code context. Through this project, we aim to push the boundaries of AI's capabilities in understanding and generating code and contribute to the development of more intelligent and useful coding tools.

PROBLEM STATEMENT:

With AI-tools such as ChatGPT taking over the world due to its ability to work as a helping hand to countless programmers and tech geeks the demand for AI-code assistants is highest than ever before, from students to professors to working professionals everybody is now using these tools. Despite having a large market only, a limited number of suppliers who fulfil the needs of the users exist, which creates an empty space in the supply of such services. So, with this project we aim to build an industry worthy product which might be the next big thing in the tech space. Developing a code autocompletion system using LLMs (Large Language Models, in this case LLAMA2) to aid software developers in writing code more efficiently and accurately. So, we decided to fine-tune the pretrained code-llama by Meta to complete code for us.

PROPOSED SOLUTION:

For code auto completer we will be finetuning the code Llama on a programming dataset.

Code Llama is Meta's refined Llama 2 variant for code generation. According to Meta, Code Llama is an evolution of Llama 2 that has been further trained with 500 billion code tokens and code-related tokens from Llama 2's code-specific datasets. To train Code Lama, Meta used more code data over a longer period.

Code Llama displays enhanced programming capabilities and can generate code corresponding to a natural language prompt.

3. APPROACH:

Fine-tuning Code Llama for to complete python code snippets.

3.1 MODEL USED:  Code Llama (AI model built on top of Llama2)

CodeLlama offers various models of different sizes. You can choose one depending on your task and hardware limitations.

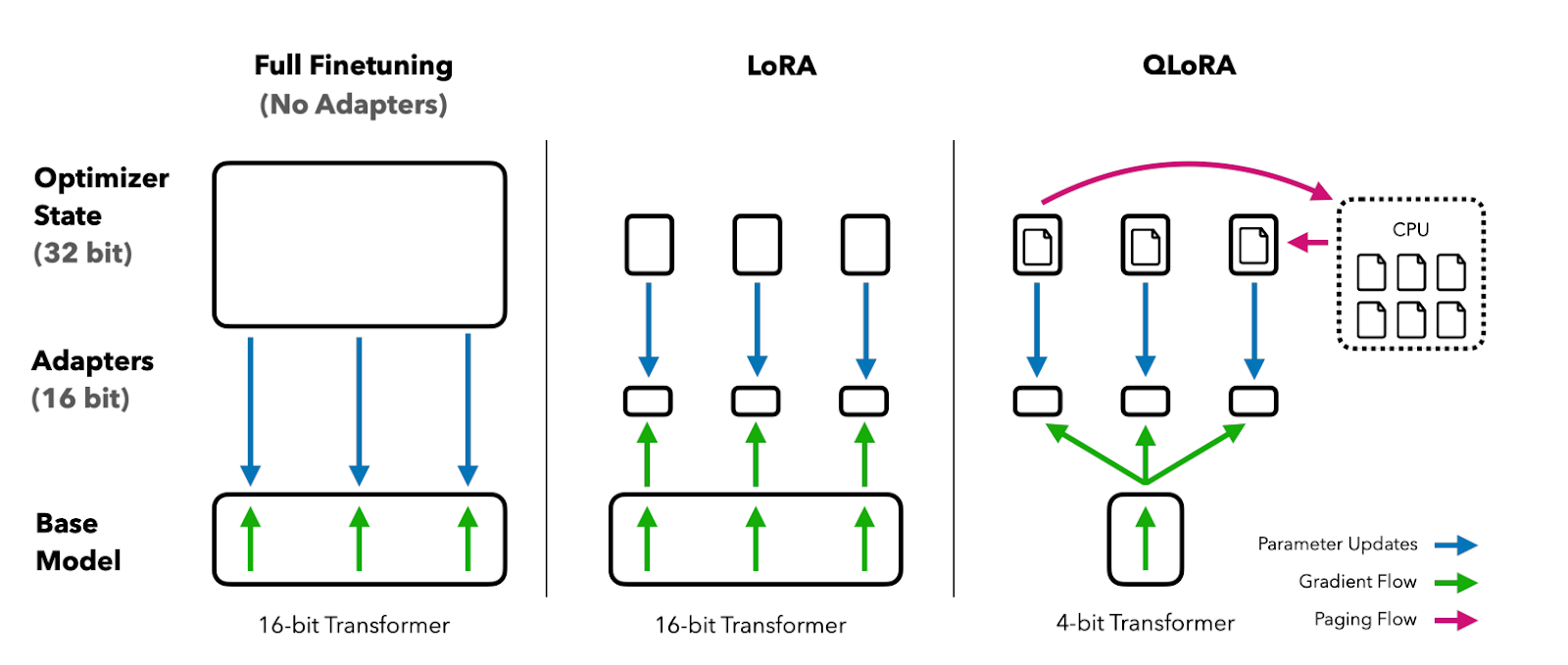
Here we are using the “codellama/CodeLlama-7b-hf” due to limited hardware as the 7B parameter model is ideal for tasks demanding low latency. Also, it can run on a single GPU.

3.2 DATASET: "code\_x\_glue\_tc\_nl\_code\_search\_adv" dataset from Huggingface (a filtered version of the CodeSearchNet dataset)

Size of the dataset: 9599 rows

3.3 CHALLENGES FACED:

Since we are students and have access to limited resources finetuning models as big as Codellama is a task. Tho in theory code llama might run on a single gpu but in our case we encountered the “cuda out of memory” error multiple times , in order to deal with this error we decided to use QLoRA (Quantized Low-Rank Adaptation of Large Language Models) technique to minimize the number of trainable parameters which minimized the hardware requirements but this still wasn’t enough for training the model as the “out of memory “ error was consistent . Finally, we were able to finetune the model by adjusting the hyperparameters which decreased the performance and the accuracy of the finetuned model but due to hardware limitations this was the only option we had.



**Fine-tuning Code Llama without LoRA:**

In general, fine-tuning Code Llama without LoRA involves training the model on a specific task or domain to improve its performance for that particular use case. The fine-tuning process typically involves adjusting the model's hyperparameters, such as learning rate, batch size, and number of training epochs, to optimize performance. In this approach all the parameters of the parameters of the model are adjusted. This approach does provide good results but can prove to be computationally very expensive. As we are working with limited hardware this might not be the best option for us.

**Fine-tuning Code Llama with QLoRA:**

QLoRA stands for Quantized Low-Rank Adaptation, an even more efficient variant of LoRA (Low-Rank Adaptation). Instead of adjusting all the parameters of the model, QLoRA introduces quantized low-rank matrices that are fine-tuned instead. Which significantly reduces the number of trainable parameters which makes the finetuning process more memory efficient. Surprisingly, even after drastically reducing the number of trainable parameters QLoRA still delivers impressive results even with very small datasets it gives impressive results.

**TECHNIQUES:** we have used transfer learning i.e., fine tuning the Code Llama with using PEFT techniques (as we have limited hardware resources)

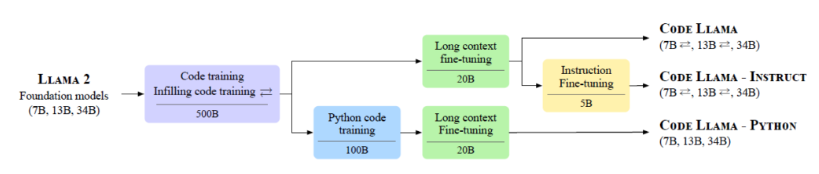
3.4 HARDWARE AND SOFTWARE REQUIREMENTS:

**Recommended Hardware:** The 7B parameter model is ideal for tasks demanding low latency such as real-time code completion. It can run on a single GPU (this might be true in theory but practically we did face the out of memory error)

**Hardware we used:** The v100 GPU, that is offered by google ColabPro.

**Software:** We used all the preinstalled dependencies from google Colab as well as other ones from HuggingFace.

4. THE FINE-TUNING PROCESS:



4.1 INSTALLING  [DEPENDENCIES](https://www.google.com/search?sca_esv=581232494&rlz=1C1ONGR_enIN1007IN1007&sxsrf=AM9HkKlBOtS1eA9ia6BbGNwmROgBE55-hQ:1699629668188&q=DEPENDENCIES&spell=1&sa=X&ved=2ahUKEwjPzPz03bmCAxUKZvUHHQaaCgIQkeECKAB6BAgJEAE):

For this specific fine-tuning task, we will use the Hugging Face ecosystem of LLM libraries: transformers, accelerate, peft, trl, and bitsandbytes.

4.2 LOADING MODEL AND TOKENIZER:

MODEL:

"codellama/CodeLlama-7b-hf"

Here we are loading the pretrained model from hugging face.

TOKENIZER:

tokenizer = AutoTokenizer.from\_pretrained("codellama/CodeLlama-7b-hf")

4.3 LOADING THE DATASET:

We used the "code\_x\_glue\_tc\_nl\_code\_search\_adv" dataset from Huggingface (a filtered version of the CodeSearchNet dataset) for finetuning the codellama model. Since we are applying the QLoRA technique to reduce the trainable parameters when went with a relatively small corpus of data as QLoRA yields good results even with less than 1000 rows of training data.

Size of the dataset: 1000 rows for finetuning.

Since we have access to limited hardware, it becomes crucial for us to save our computation power wherever possible. QLoRA has proven to yield good results even with less than 1000 rows of training data. So, we decided to shrink the size of the dataset to 1000 rows as we can still expect good results from it.

4.4 4-BIT QUANTIZATION CONFIGURATION:

bnb\_config = BitsAndBytesConfig

(

load\_in\_4bit=True,

bnb\_4bit\_quant\_type="nf4",

bnb\_4bit\_compute\_dtype="float16",

bnb\_4bit\_use\_double\_quant=True

)

Here we are creating a 4-bit quantization with NF4 type configuration using BitsAndBytes.

4.5 PEFT PARAMETES

peft\_config = LoraConfig

(

r=8,

lora\_alpha=16,

lora\_dropout=0.05,

bias="none",

task\_type="CAUSAL\_LM"

)

Here, r= is the rank of the low-rank approximation in Lora.

Lora\_alpha = scale factor for the Lora parameters. A larger lora\_alpha results in the Lora parameters having a larger effect on the fine-tuned model.

lora\_dropout: This is the dropout rate for the Lora parameters.

bias: This parameter determines how the bias parameters of the model are handled during fine-tuning.

task\_type: This parameter specifies the type of task the model is being fine-tuned for.

4.6 GENERATING PROMPT:

The prompt text(string) was generated using the “code” and the “docstring” column.

df["text"] = df[["docstring", "code"]].apply(lambda x: "[INST] Docstring: " + x["docstring"] + " [/INST] Code: " + x["code"] + "", axis=1)

4.7 TRAINING PARAMETERS:

Here is the list of hyperparameters that we used to optimize the training process.

training\_arguments = TrainingArguments

(

output\_dir="fine-tuned-codellama”,

per\_device\_train\_batch\_size=8,

gradient\_accumulation\_steps=4,

optim="paged\_adamw\_32bit",

learning\_rate=2e-4,

lr\_scheduler\_type="cosine",

save\_strategy="epoch",

logging\_steps=10,

num\_train\_epochs=1,

max\_steps=100,

fp16=True,

push\_to\_hub=True)

4.8 SUPERVISED FINE-TUNING(SFT):

The SFTTrainer class is a custom trainer class specifically designed for Soft Fine-Tuning (SFT), a technique used to fine-tune large language models. Here we are creating an instance of the SFTTrainer class.

trainer = SFTTrainer

(

model=model,

train\_dataset=data,

peft\_config=peft\_config,

dataset\_text\_field="text",

args=training\_arguments,

tokenizer=tokenizer,

packing=False,

max\_seq\_length=512

)

4.9 TRAINING:

trainer.train()

trainer.push\_to\_hub()

Number of recommended epochs for fine tuning code llama: 3.

Number of epochs we used: 1.

Since we had access to limited hardware so to avoid the “CUDA OUT OF MEMORY” error we trained it for only one epoch which indeed compromised the accuracy.

A screenshot of a computer

Description automatically generated

Training time: it took about 1hr and 15 minutes for 1 epoch.

After finetuning the codellama model we pushed the model onto the hugging faces using hugging faces API.

4.10 MODEL INFERENCING AND TESTING

from transformers import AutoTokenizer

from transformers import pipeline

import torch

tokenizer = AutoTokenizer.from\_pretrained("Vasanth/codellama2-finetuned-codex-fin")

pipe = pipeline(

"text-generation",

model="Vasanth/codellama2-finetuned-codex-fin",

torch\_dtype=torch.float16,

device\_map="auto",

)

sequences = pipe(

'def fibonacci(',

do\_sample=True,

temperature=0.2,

top\_p=0.9,

num\_return\_sequences=1,

eos\_token\_id=tokenizer.eos\_token\_id,

max\_length=100,

)

for seq in sequences:

print(f"Result: {seq['generated\_text']}")

A screenshot of a computer program

Description automatically generated

As we can see the fine-tuned model does help in completing the code but sometimes it does fail to provide good results. It can possibly be due to limited training data, changes made to hyperparameters and the number of training epochs we made due to our hardware limitations.

Given better hardware, revised hyperparameters and increased number of epochs this might work as expected.

5. RESULTS AND ANALYSIS:

**EVALUATION MATRICES USED:**

**Rouge Score:** ROUGE (Recall-Oriented Understudy for Gisting Evaluation) is a metric for evaluating machine-generated text by measuring the overlap of n-grams between the generated and reference text. Common types include ROUGE-1 (unigrams), ROUGE-2 (bigrams), and ROUGE-L (longest common subsequence), assessing precision, recall, and F1-score to gauge the quality of generated content.

**F1 Score**: The F1 score is a metric that combines precision and recall into a single value, providing a balanced measure of a model's performance. It is the harmonic mean of precision and recall.

**Experimental Results:**

The fine-tuned model was tested on about 25 examples.

|  |  |  |  |
| --- | --- | --- | --- |
| Rouge-1 | Rouge-2 | Rouge-L | F1 score |
| 0.045 | 0.045 | 0.045 | 0.210 |

The fine-tuned model has a Rouge-L of 0.045 and F1 score of 0.210.

The performance of the model based on results is fine, but it does not perform exceptionally well, increasing the number of epochs and switching hyperparameters might help to boost and better the results. As discussed above, since we had access to limited computational units we compromised on the number of epochs, hyperparameters and the size of the test and train data.

6. IMPROVEMENTS AND FUTURE WORK:

**Limitations:** The only limitation which held us back from fine-tuning the model to its full potential was limited access to GPU.

**Future work:**

If given access to the better hardware we can expect the model to work even better, we can fine tune it on a larger corpus of data and see which values of hyperparameters yield better results by constant experimentation. Also maybe working on a better dataset could help us increase accuracy as well.

7. TAKING A LOOK AT THE MODEL FROM BUSINESS PERSPECTIVE:

With the boom of AI tools like ChatGPT, programmers and people in tech are becoming more reliant on AI tools and assistants specially for coding purposes as it saves time and works as a personal tutor for many. People in the tech field are willing to spend money on such services as they work as a great helping hand and save time and effort all while clearing doubts of students and new programmers in the market.

There does exist a demand in the market and only a few suppliers exist for this service. If we can better the results for this product in terms of accuracy and contextual understanding this might be the next big tool in AI-space.

8. CONCLUSION  
With Rouge-L score of 0.045 and F1 score of 0.210 the model performs somewhat descent when it comes to code auto-completion. With better GPU and larger corpus of train and test data we can achieve even better results.

The model with better results can be used as an algorithm for code auto-generation services which could possibly be a profitable business as the demand for such services in today’s world is really high.