Fine-tuning LLM for Multilingual Code Generation

Link to Github Repo: LLM-for-code-intelligence

Link to Google Doc to add comment and see latest version: click here

OVERVIEW

- Large Language Models of Code (Code LLMs) have become powerful and useful to many developers; but their ***High Computational Requirements*** and their focus on ***Only Popular Programming languages*** are major limitations. In this work will propose, develop and evaluate an approach to overcome both limitations.
- **Very** Large code LLMs (> 1B params) often have good performance in a variety of languages (**Salesforce/codegen-2B-multi -> C, C++, Go, Java, JavaScript, and Python**)
- **Small** code LLMs (< 1B params) are usually trained to work on one programming language to have good performance (**Salesforce/codegen-350M-mono -> Python**)
- ***Solution:*** Small code LLMs can be extended to cover more languages by a variety of methods:
    a. Full Fine-tuning:
        - Pros: Best performance in fine-tuned model
        - Cons: High computational requirements, catastrophic forgetting
    b. LoRa Fine-tuning:
        - Pros: Efficacy, Scalable, No catastrophic forgetting
        - Cons: Lower performance than full fine-tuning
    c. Knowledge Distillation: **(TBC)**
        - Pros: Proven to work in Natural Language LLMs
        - Cons: High computational requirements

GOALS

1. ***Main Experiment :*** Compare and evaluate mentioned methods to extend Small code LLMs to new programing language considering:
    a. Performance: in terms of Perplexity and HumanEval **(Done)**
    b. Efficiency: in terms of Memory and Time **(Done)**
2. ***Ablation Studies:*** Study the effect of different factors on the training, evaluation and inference of models:
    a. Training Data: Filtering, Distributions, Preprocessing **(Done)**
    b. Training Strategies: learning rate, batch size, LoRa Layers **(Done)**
    c. Evaluation: Stride and stop tokens **(Not Started)**
3. Qualitative Analysis of the results:
    a. Failure modes comparison in HumanEval between Python and Java (Exceptions, runtime errors and not passing test cases) **(Done)**
    b. Correlation between Perplexity, Evaluation Loss and HumanEval **(Not Started)**
4. Further objectives:
    a. Relation between Different Programming Languages **(Not Started)**
    b. Knowledge Distillation **(Not Started)**

Timeline:

1. Read papers on Code LLM and select appropriate baseline -> **Salesforce/codegen-350M-mono**
2. Read papers on Code datasets and select perplexity evaluation data -> **bigcode/the-stack-dedup**
3. Read papers on evaluation strategy to implement HumanEavl in python and Java -> follow standards of **CodeX** and **Multipl-E**
4. Setting up Evaluation:
    - Done: Perplexity Evaluation on Google-Colab and HumanEval on Google-Colan and Virtual Containers (for Java)
    - Challenges: Perplexity Dataset, Humaneval Stop tokens for java
5. Generate Baseline Results using baseline model
6. Setting up Training and Evaluation environments:
    - Done: Configure Full & LoRa fine-tuning on Google-Colab, Track training progress on WandB, Save checkpoints on HuggingFace.
    - Challenges: Resuming Training reset some variables(LR, number of steps), Memory to 15GB and duration to 3 hours
7. Generate Results for the full tuned model and LoRa tuned model.
8. Edit the training setup (learning rate , gradient accumulation and batch size) based on the literature (**bigcode/starcoder**) and the results of the first finetuning
9. Train the model again and notice the effect of modification on two models:
    - Full fine tuning: *Significant improvement* in Perplexity and HumanEval for **Java** and *significant drop* for **Python**
    - LoRa fine tuning: *Significant improvement* in Perplexity and *Slight* in HumanEval for **Java** and *no change* for **Python**
10. Plan Next steps: further modifications, larger models, inference and distillation
11. Setting up live inference environment:
    - Done: Live inference configured on HuggingFace spaces to generate code from tuned models base on given prompt -> Code Inference
    - Challenges: CPU limitation
12. Got the best parameters for each model to train and evaluate effectively, and generated final results of main experiment
13. Experiments on fine-tuning dataset ➔ got the final dataset after filtering , de-duplications, splitting, and maximizing number of tokens (more documentation required) Dataset
14. Experiment on LoRa Parameters ➔ Tested different Ranks and Layers and found the highest performing on the HumanEval
15. Basic Error Analysis based on type of error (runtime, exception, assertion...) and the cause of error (missing smi-colon, bad type definition, uncompleted code)
16. comparison with multi-lingual small code LLMs and multi-lingula large LLMs (base and fine tuned) ➔ more context about the results.

Results

We fine-tuned our baseline model **Salesforce/codegen-350M-mono(Python)** model using two methods (LoRa Fine-tuning and Full fine-tuning). We also Experimented with a number of different factors affecting both ***Performance*** (scores on Perplexity and HumanEval) and **Efficiency** (Required Memory and time to complete training). These factors include: batch size,

learning rate, data preprocessing, LoRa filtering, Model Quantization, Gradient Checkpointing. The Following Section Summarizes the most important finding .

1. Discussion:

a. Objective:

- We want to adopt a monolingual code LLM from one programming language (Python) to another (Java)
- In order to do so , we start with monolingual baseline model : **Salesforce/codegen-350M-mono(Python)**
- This is an autoregressive code completion model pretrained on The mono-lingual dataset BIG PYTHON
- To verify our claim, i.e, you can efficiently adopt a small LLM to work on a variety of programming languages, we will need to fine-tune this baseline on a Java corpus.
- For this we will use two fine tuning methods: Full fine-tuning (100% of the model parameters) and LoRa finetuning (5% of the model parameters)

b. Findings:

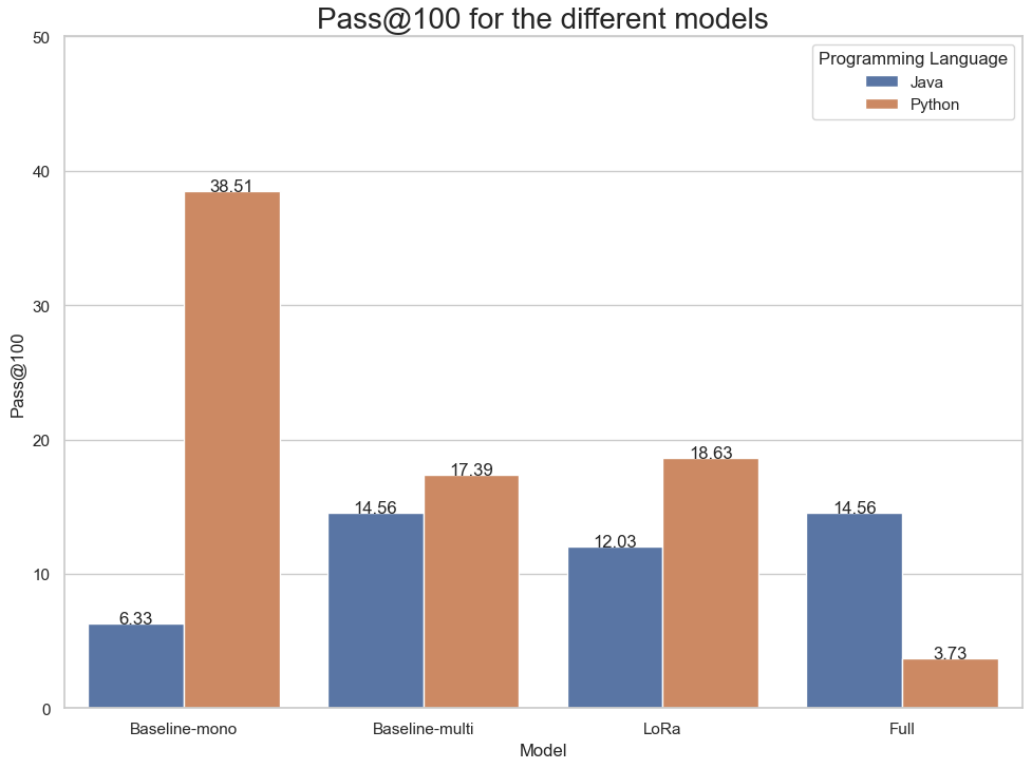

Figure 1 shows the Pass@100 for the different models. **Baseline-mono** and **Baseline-multi** are the two 350M models without any finetuning. The **LoRa** and **Full** are the results of the fine tuned models of the Baseline-mono.

Baseline-mono was pre-trained on the **BigPython dataset**. The data consists of **71.7B tokens** of the Python programming language.

Baseline-multi was pretrained on the **BigQuery dataset**. The data consists of **119.2B tokens** and includes C, C++, Go, Java, JavaScript, and Python.

The **LoRa model** was fine tuned on the **Stack dataset**. The finetuning was done for **24.5M tokens** of Java programming language.

The **Full model** was fine tuned on the **Stack dataset**. The finetuning was done for **8.2M tokens** of Java programming language.

We were able to finetune the LoRa model for 3K steps and the Full model for 1K steps, using the same compute resources for both models. That is because the LoRa parameters are only 10% of the Full model parameters. Of all the models, the Baseline-mono has the best pass@100 score for python at 38.51% , but the worst score for java at 6.33%. The Baseline-multi and the Full models have the best pass@100 score for java at 14.56% for both models. However, the Full model has the worst score for python at 3.73% followed by the Baseline-multi at 17.39%. The LoRa model has the second best pass@100 score for python at 19.21% and the third best score for java at 12.03%.

The important takeaways from theses results are:

1. For Smaller models , training on a single programming language is significantly better than training on multiple programming languages as shown by the Baseline-mono and Baseline-multi models.

2. Using Parameter Efficient Fine tuning (PEFT) enables us to finetune a smaller number of parameters on a larger number of tokens using the same compute resources and still get comparable results to using Full Parameter Fine tuning. More Results on Training times and Memory usage will be added.

3. The Full Model has the same pass@100 score as baseline-multi for java, even though it has seen far less tokens. This shows that smaller models are not suitable for pretraining on multiple programming languages.

4. The LoRa Model score on Python is 5 times better than the Full model, Proving that PEFT is a better approach to avoid catastrophic forgetting.

5. All the scores on Java are low compared to Python. This may indicate a problem with the evaluation as the problem prompts in Java are significantly longer and the stopping criteria is not well defined for Java. **(Not Started)**

6. The Fact that the LoRa model has good scores on both Python and Java could be explored further on the translation task of HumanEval. **(Not Started)**

2. Analysis:

a. Execution Based Error Analysis:

- Analysis of code generated by fine-tuned models to discover the most frequent and unexpected modes of failure**(Ongoing - Progress here)**

b. Correlation Between Metrics:

- So Far there seems to be a very small correlation between HumanEval and Peplexity Metrics. However, we still have more experiments to do with perplexity.
- The Validation Loss also seems unrelated to HumanEval Score, ax extra training improve HumanEval but the validation Loss stays the same (similar to perplexity)
- Finally, Further research into evaluation is required to find the optimal parameters for our task and implement further tasks like code-translation in the next steps.