Mixture of Expert Model for Code Generation

TEAM 11
Adithya Kameswara Rao akameswa@andrew.cmu.edu
Santhoshkumar Panneerselvam spanneer@andrew.cmu.edu
Yihao Jia yihaoj@andrew.cmu.edu
Yirui Zhu yiruiz@andrew.cmu.edu

Abstract

This paper explores the efficacy of the Mixture-of-Experts (MoE) framework in the realm of code generation, presenting a compelling performance compared to monolithic model architecture. Central to this investigation is the application of MoE to code generation on the Mistral-7B base model, adeptly tailoring to handle distinct programming languages, including Python, C++, Java, and Javascript, where the tasks are described as text question to code snippets. By dividing the problem space and deploying experts only when required, the MoE architecture demonstrates a significant reduction in computational overhead while maintaining high precision in code generation. Preliminary results indicate that this framework not only achieves 50% performance improvement with CodeBLEU benchmark, but also offers a scalable and economically viable solution. This study underscores the potential of MoE to democratize machine learning applications by making advanced computational models accessible without prohibitive investments, thereby broadening the scope for innovation and application in various tech-driven domains.

1 Introduction

Deep learning, as a specialized branch within the machine learning domain, exhibits an increasingly commanding prowess in content generation. While the consensus suggests that complex models tend to achieve superior outcomes, the necessity for advanced GPUs and the associated costs render such endeavors viable only for behemoth corporations. In this context, the Mixture-of-Experts (MoE) [1] method emerges as a pioneer that democratizes deep learning by circumventing the prohibitive costs associated with simpler models but equivalent performance. Essentially, MoE segments the problem space into discrete sub-regions, each addressed by specialized, simpler models or 'experts,' rather than employing a monolithic model for the entire domain. With adept integration, such an MoE framework can achieve performance parity with larger counterpart models, albeit in a more economically feasible manner.

Taking the realm of code generation as a point of reference, and as depicted in Figure 1, we propose a MoE framework that deftly selects the C language expert model for output generation if the input code snippet is C language. Analogously, when presented with Python code, the system channels the task to the expert model versed in Python for processing. This sophisticated architecture empowers the system to harness the prowess of distinct expert models tailored to various programming languages, thereby significantly enhancing the precision of outputs over the conventional baseline model.

In our pursuit, we endeavor to meticulously examine the Mixture-of-Experts (MoE) paradigm as it pertains to code generation. Presently, Large Language Models (LLMs) tasked with code generation typically function as monolithic systems across a variety of programming languages. Although these models are capable of generating competent outputs, their efficacy may wane when tasked with specific programming languages. For instance, a sophisticated model might exhibit superior performance in generating code for Python as opposed to Java and C++, despite all these

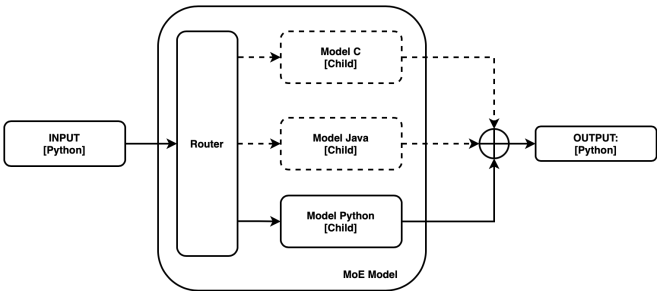

Figure 1: Applying MoE Architecture on Code Generation

languages falling well within its realm of expertise. Such observations compellingly suggest that adopting the MoE framework for code generation can be both intellectually sound and potentially advantageous. Given that code generation naturally divides into discrete sub-domains—namely, different programming languages—the assignment of specialized experts within an MoE model for each language offers a significant opportunity to surpass the capabilities of a singular, complex model in handling multiple programming languages.

2 Literature Review

MoE stands as a cutting-edge approach, leveraging a sophisticated model ensemble to handle diverse and complex tasks efficiently. The powerful fusion of MoE and a code-gen model directs specialized expert models, elevating its capabilities in code generation for different programming languages.

2.1 Mixture-of-Experts (MoE)

Mixtral [1]: The "Mixtral of Experts" paper introduces the Mixtral 8x7B model, an advanced sparse MoE language model built upon the Mistral 7B architecture. It features eight feedforward blocks per layer and selects two experts for each token. Overall, it boasts 47B parameters but utilizes only 13B during inference. Thus, this model excels in computational efficiency and performance, surpassing benchmarks set by models like GPT-3.5 and Llama 2 70B in areas including mathematics, code generation, and multilingual translation.

Gemini [2]: Gemini 1.5, boasting significantly improved performance and greater speed over its predecessor, Gemini 1.0, primarily attributes its advancements to the incorporation of the MoE framework into its foundational architecture. As claimed by Google, Gemini 1.5 outperforms 87% in benchmark experiments through the integration of the MoE approach within models such as GShard-Transformer, Switch-Transformer, M4, among others.

SegMoE [3]: SegMoE, introduced by Segmind, is an innovative open-source framework designed to enhance text-to-image generation. It combines multiple generative image models into more comprehensive and efficient systems, utilizing Stable Diffusion's architectural principles enhanced with the sparse MoE layers. This addition optimizes expert selection for each token, aiming to significantly boost image quality and prompt accuracy.

2.2 Code Generation Model

SantaCoder [4]: SantaCoder is a 1.1 billion parameter decoder-only transformer architecture designed for code generation tasks, trained on Java, JavaScript, and Python from The Stack dataset. Building upon earlier efforts in the BigCode community, the model integrates Multi Query Attention (MQA) and Fill-in-the-Middle (FIM) techniques, along with preprocessing filters. After 600,000 iterations of training, SantaCoder outperforms larger models on code generation and infilling tasks across Java, JavaScript, and Python programming languages.

Phi-2 [5]: Phi-2 is a large language model based on the Transformer architecture, comprising approximately 2.7 billion trainable parameters. Its training data encompassed the same sources utilized for the earlier Phi-1.5 model, supplemented by an additional corpus consisting of synthetically generated natural language processing (NLP) texts and filtered web content. Phi-2 demonstrated

performance nearing the state-of-the-art among models constrained to fewer than 13 billion parameters on commonsense reasoning, language comprehension, and logical inference capabilities.

3 Model Description

3.1 Architectural Details

The Transformer architecture has been widely used for natural language processing. It has become the de-facto standard for many sequence-to-sequence tasks, such as machine translation. Transformer makes use of two computational blocks, an encoder and a decoder, both implemented by stacking multiple Transformer layers. Transformer encoder layer consists of two consecutive layers, namely a self-attention layer followed by a position-wise feed-forward layer. Decoder adds third cross-attention layer, which attends over encoder output.

Mixtral is based on the Transformer architecture with the notable exceptions that it supports a fully dense context length of 32k tokens, and the feed-forward blocks are replaced by Mixture-of-Expert layers. The model architecture parameters are summarized in Table 1 on the right.

Parameter	Value
dim	4096
n_layers	32
head_dim	128
hidden_dim	14336
n_heads	32
n_kv_heads	8
context_len	32768
vocab_size	32000
num_experts	8
top_k_experts	2

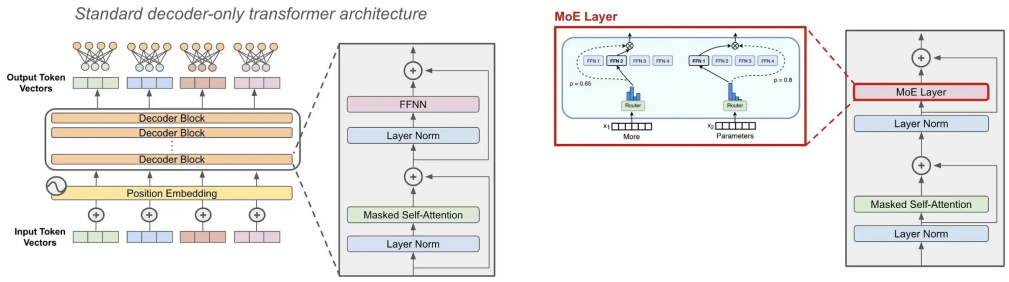

Figure 2: Illustration of scaling of Transformer Decoder with MoE Layers. The MoE layer replaces each feed-forward sub-layer and is comprised of several experts [6]

3.2 Structure of the Mixture-of-Experts Layer [1]

The Mixture-of-Experts (MoE) layer consists of a set of n "expert networks" $E_1, \ldots, E_n$, and a "gating network" G whose output is a sparse n-dimensional vector. The experts are themselves neural networks, each with their own parameters. Although in principle we only require that the experts accept the same sized inputs and produce the same-sized outputs, in our initial investigations in this paper, we restrict ourselves to the case where the models are feed-forward networks with identical architectures, but with separate parameters.

Let us denote by $G(x)$ and $E_i(x)$ the output of the gating network and the output of the i-th expert network for a given input x. The output y of the MoE module can be written as follows:

$$y = \sum_{i=1}^{n} G(x)_i \cdot E_i(x) \tag{1}$$

We save computation based on the sparsity of the output of $G(x)$. Wherever $G(x)_i = 0$, we need not compute $E_i(x)$.

3.3 Gating Network [7]

Softmax Gating: A simple choice of non-sparse gating function is to multiply the input by a trainable weight matrix W_g and then apply the Softmax function.

$$G_\sigma(x) = \text{Softmax}(x \cdot W_g)$$

Noisy Top-K Gating: We add two components to the Softmax gating network: sparsity and noise. Before taking the softmax function, we add tunable Gaussian noise, then keep only the top k values, setting the rest to $-\infty$ (which causes the corresponding gate values to equal 0). The sparsity serves to save computation, as described above. While this form of sparsity creates some theoretically scary discontinuities in the output of gating function, we have not yet observed this to be a problem in practice. The noise term helps with load balancing. The amount of noise per component is controlled by a second trainable weight matrix W_{noise}.

$$G(x) = \text{Softmax}(\text{TopK}(H(x), k))$$

$$H(x)_i = (x \cdot W_g)_i + \text{StandardNormal}() \cdot \text{Softplus}((x \cdot W_{noise})_i)$$

$$\text{TopK}(v, k)_i = \begin{cases} v_i & \text{if } v_i \text{ is in the top } k \text{ elements of } v \\ -\infty & \text{otherwise} \end{cases}$$

Training the Gating Network: We train the gating network by simple back-propagation, along with the rest of the model. If we choose $k > 1$, the gate values for the top k experts have nonzero derivatives with respect to the weights of the gating network. Gradients also back-propagate through the gating network to its inputs.

4 Dataset

We used the xlcost-text-to-code dataset offered by organization "codeparrot" on Hugging Face. The dataset offers pairs of natural language instructions and their correspondin code snippets in 7 programming languages: Python, C, C++, Java, JavaScript, and PHP.

Each language is represented by approximately 10,000 data points. The dimensions and sample size of the dataset have made it an ideal fit for our project's goal of fine-tuning a model that generates code from textual description.

To optimize the dataset for our model training focused on Python, Java, JavaScript, and C++, we performed a preprocessing step to filter out other programming languages. This filtering is crucial as it eliminates irrelevant data that could introduce noise and negatively impact the model's accuracy.

5 Evaluation Metric

We leverage a system called **CodeBLEU** [9] to benchmark our models. CodeBLEU is a novel automatic evaluation metric proposed for code synthesis tasks like text-to-code, code translation, and code refinement. The traditional BLEU metric, which is widely used for evaluating machine translation systems, essentially measures the n-gram overlap between the generated output and reference output. However, it fails to account for the unique characteristics and requirements of programming languages as opposed to natural languages.

To address these shortcomings, CodeBLEU is designed as a weighted combination of four components as a final CodeBLEU score: 1) BLEU score for n-gram matching; 2) Weighted n-gram match; 3) Syntactic AST match; 4) Semantic data-flow match. With these mechanisms, CodeBLEU achieved significantly higher correlation with human judgments across all three tasks compared to just using the standard BLEU metric or simple accuracy measures like exact string match.

Compared to other bench-marking tools, CodeBLEU is a more comprehensive automatic evaluation approach for code synthesis systems, and is better correlated with actual human judgments of code quality compared to existing metrics like BLEU and exact match accuracy.

6 Loss Function

We have selected the Cross Entropy Loss Function (referred to herein as the Loss) for its adeptness in quantifying the congruence between a large language model's predicted probability distribution and the actual distribution of the target data. Mathematically, the Loss function is formalized as follows,

and for a batch of data, such function is calculated as the mean of the losses incurred for each data point within the batch.

$$L = -\sum_{c=1}^{M} y_c \log(p_c)$$

where:

- L is the loss for one training example.
- M is the number of classes (token types in the language model).
- y_c is the binary indicator (0 or 1) if class label c is the correct classification for the observation.
- p_c is the predicted probability of the class c given the input.

This Loss is instrumental during our fine-tuning process as it compels the models to reduce the divergence between their predictions and the actual token distributions. This loss encourages each specialist to elevate the likelihood of producing syntactically and semantically accurate code snippets in its designated language, thereby enhancing the collective performance.

7 Baseline Selection

In our quest to ascertain the efficacy of the MoE approach within code generation, three pivotal attributes must be meticulously considered while selecting a foundational LLM baseline:

Universality and Adaptability: To authentically validate MoE's universal applicability, selecting a model with broad utility is paramount. This approach minimizes extraneous variability, thereby providing a cleaner evaluation canvas. Mistral-7B, renowned for its exceptional performance among models of similar scale, emerges as an ideal candidate. Its widespread adoption for specialized fine-tuning further exemplifies its versatility and universal appeal.

Ease of Fine-tuning: In the domain of code generation where MoE's application is being tested, fine-tuning is not just beneficial but essential. The widespread acclaim of Mistral-7B has fostered a rich ecosystem of fine-tuning experiences, shared openly through comprehensive notebooks and insightful blogs detailing both successes and setbacks. This wealth of shared knowledge facilitates smoother and more effective fine-tuning processes.

Integration Maturity: Mistral-AI, the progenitor of Mistral-7B, is acclaimed for its flagship Mixtral model, an MoE construct powered by eight Mistral-7B units. The triumph of Mixtral has not only revitalized MoE discourse in academic and industrial spheres but also cemented its status as a mature, cost-effective, and leading-edge Proof of Concept in model enhancement.

Concisely, Mistral-7B, backed by extensive community support and a robust track record of successful implementations, stands out as the quintessential model for evaluating the MoE paradigm. Its proven adaptability, ease of fine-tuning, and integration maturity make it an exemplary baseline for this innovative venture.

8 Baseline Implementation Completeness

8.1 Finetuning

We adopted traditional fine-tuning techniques, specifically parameter-efficient fine-tuning (PEFT) with quantization using adapters (QLoRA) to cater to our resource-optimized requirements. The dataset comprises English text alongside its corresponding code translations. Each program is segmented into code snippets, with snippet-level subsets containing these snippets along with their associated comments. For program-level subsets, comments were concatenated into a single description. Additionally, the programs in all languages are aligned at the snippet level, with comments for each snippet being identical across all languages.

Example Dataset

```
Text: Sum of sum | Function to find the sum ; Driver code
```

```
Program:
def sumOfSumSeries ( n ) :
NEW_LINE INDENT return ( n * ( n + 1 ) * ( n + 2 ) ) // 6
NEW_LINE DEDENT N = 5 NEW_LINE print ( sumOfSumSeries ( N ) )
```

We began with the Mistral 7B model and fine-tuned it for Java, JavaScript, C++, and Python code generation. Our specialized models, tailored to each language, showed promising results in improving code generation capabilities.

8.2 Model Blending

Task vectors: Task vectors represent directional cues within a pre-trained model's weight space, guiding enhancements in performance for specific tasks. They are derived by subtracting the weights of a pre-trained model from those fine-tuned on a particular task. Arithmetic operations such as negation and addition on these task vectors modify the resulting model's behavior. Negating a task vector, for instance, can diminish performance on a target task while minimally affecting others. Conversely, adding task vectors enhances performance across multiple tasks. Moreover, combining task vectors based on analogy relationships between tasks can enhance performance on related tasks, even without utilizing their data during training.

TIES-Merging (TrIm, Elect Sign & Merge) vs. SLERP (Spherical Linear Interpolation): While several merging techniques exist, we chose TIES-merging, which introduces three novel steps when merging models: (1) resetting parameters that underwent minimal change during fine-tuning, (2) resolving sign conflicts, and (3) merging only parameters aligned with the final agreed-upon sign. TIES-merging has demonstrated superior performance across diverse settings, encompassing various modalities, domains, tasks, model sizes, architectures, and fine-tuning scenarios.

In contrast, SLERP is a simpler merging technique that supports the merging of only two models at a time. It involves interpolating between the parameters of the two models along a spherical path. While SLERP offers simplicity and efficiency, its scope is limited compared to TIES-merging, which can handle multiple models simultaneously.

8.3 Creating Mixture of experts

To create the MoE model, we utilize the mergekit [8] library. This script combines the self-attention and layer normalization parameters from a "base" model with the MLP parameters from a set of "expert" models. Each expert in the MoE gating can be configured to be selected by positive prompts, while negative prompts can be used to ignore certain experts. These prompts represent examples that the router network uses to choose the appropriate expert based on the input. During inference, when users provide prompts similar to these examples, the router network activates the right expert model.

Training a MoE gating mechanism involves optimizing the parameters of the gating network during training to route inputs through the most suitable set of experts. This process enhances model specialization and performance. The training of MoE gating will be included in the final report.

9 Experiments

The experiments workflow involves 3 steps: 1) fine-tuning; 2) merging; 3) bench-marking. In the following subsection, we will deliberate the details, especially the settings of the experiments.

9.1 Fine-tuning

The experimental first fine-tuning base model into several models, resulting in the creation of several derivative models. This fine-tuning was tailored to each model's unique dataset for different programming language, ensuring optimal performance for different programming language context. As a output of this process, 4 different derivative model were generated, i.e., Mistral-Python-7B, Mistral-Java-7B, Mistral-Javascript-7B, and Mistral-C++-7B.

To smooth the process of fine-tuning and reduce cost in an acceptable level, several advanced techniques stand at the forefront of this transformative process.

Supervised Fine-Tuning (SFT) is employed to calibrate a pre-trained Large Language Model for a particular downstream application using a dataset annotated with relevant labels. By training on task-specific data, SFT guides the model to adjust its parameters to capture nuances pertinent to the target domain. This ensures that the predictive model not only retains its general linguistic capabilities but also acquires the ability to decipher and process domain-centric language, thereby significantly improving task-oriented performance.

Low-Rank Adaptation (LoRA) represents an innovative step towards efficient fine-tuning. By altering the weights in the adapter layers of the neural network while preserving the original model parameters, LoRA introduces a computationally frugal method that enables the model to adapt to new tasks without the need for extensive re-training. This technique is particularly valuable in deploying models in resource-constrained environments, where computational efficiency is as critical as model performance.

Quantization stands as a technique to reduce the model's demand on memory and computational power. In essence, it converts the model's parameters to a lower-bit representation, typically using only 4 bits. This remarkable compression allows for the deployment of powerful models even on devices with limited capacity, ensuring broader accessibility without compromising on speed or efficiency. By streamlining the computational load, quantization ensures that the benefits of advanced machine learning models can be enjoyed across a wide spectrum of devices, from high-end servers to mobile phones.

9.2 Merging the Models

Subsequently, the individually optimized models were amalgamated into an integrated framework, i.e., Mergekit. This integration was strategically engineered to leverage the strengths and mitigate the weaknesses of each constituent model, thereby enhancing overall accuracy and robustness. The seamless merging of models facilitates a comprehensive analytical capability, which is greater than the sum of its parts. This process requires careful preparation and execution, ensuring that the resulting MoE model leverages the specialized knowledge embedded in each expert while maintaining operational efficiency.

9.3 Evaluation

CodeBLEU as the bench-mark in evaluating the performance of the base model against that of the fine tuned model, and the newly integrated models. The evaluation process not only highlighted the enhanced performance of the integrated models but also provided critical insights into areas for further refinement. Through this systematic evaluation, the study can unveiled the result that whether the merged models deliver substantial advancements over the initial base model.

9.4 Resources

The computational framework employed is the AWS A10G GPU, which provides robust processing capabilities. The system's RAM is provisioned at 26GB, ensuring sufficient memory for operations. For fine-tuning, the process spans 48 to 72 hours per language, contingent upon the dataset's size. Merging operations are conducted on a T4 x 2 setup in Kaggle platform with 12GB RAM, taking approximately 30 minutes. Lastly, the evaluation phase was conducted in the same machine of fine-tuning, taking approximately 12 hours for each language.

10 Results

Table 1 captures the CodeBLEU scores for each programming language separately on both the baseline and the fine-tuned Mistral model for comparison. As demonstrated, the fine-tuned model has improved as anticipated.

Figure 3 visualizes the performance distributions of the Mistral 7B model before and after fine-tuning. Generally, the baseline model peaks at lower CodeBLEU scores, while the fine-tuned models exhibit a broader range, peaking at higher CodeBLEU scores. This suggests significant improvements in generating higher-quality code and adapting to diverse coding tasks.

Table 1: Performance for fine-tuned models.

Language	Baseline	Finetuned
Python	0.1800	0.4000
Java	0.1860	0.4287
JavaScript	0.1909	0.4182
C++	0.2170	0.3340

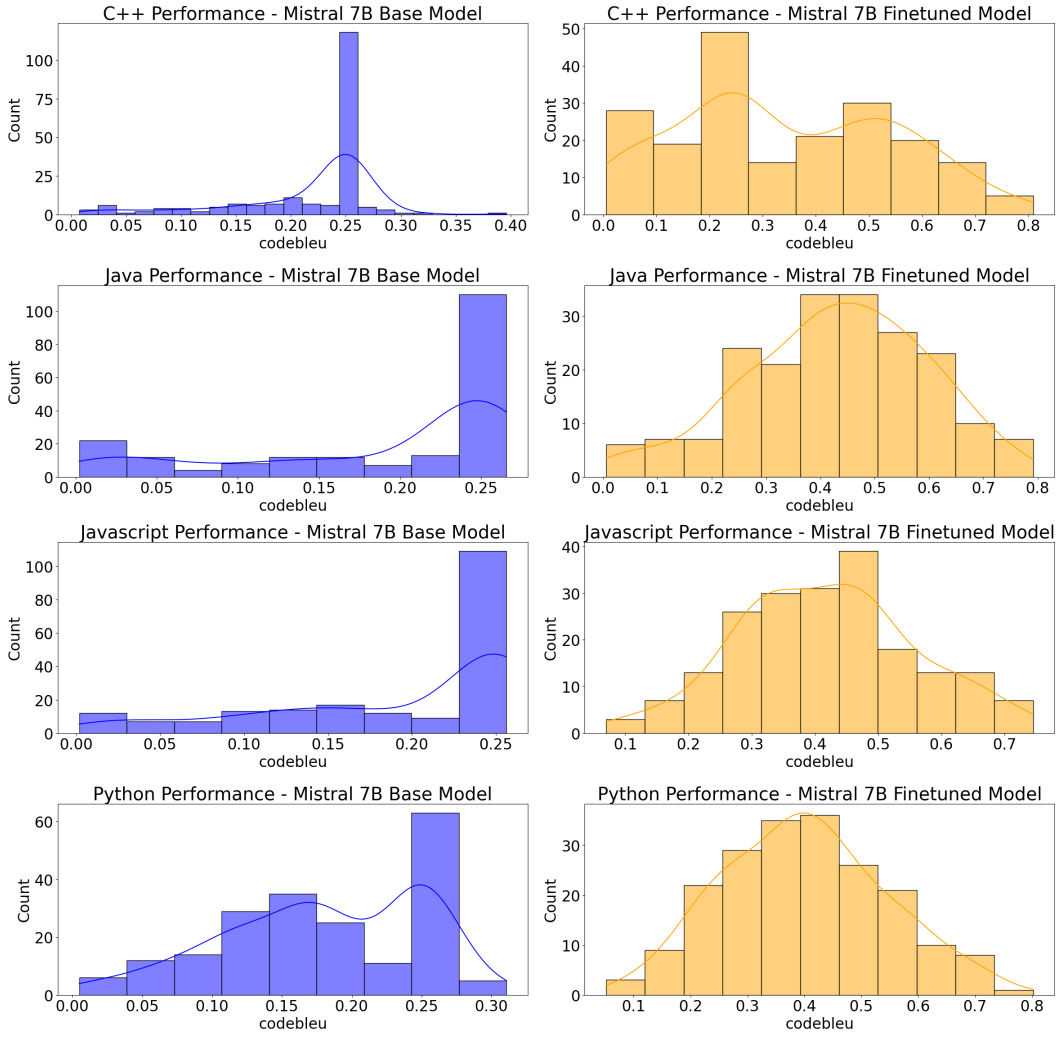

Figure 3: Performance distribution for base and fine-tuned model.

To better evaluate model inference with zero-shot prompting, a shuffled dataset with fixed 50 data points are selected from each of the four test datasets. As shown in Table 2, the MoE model introduces considerable improvement by 51% based on the test dataset across all four programming languages.

Table 2: Performance for MoE.

Language	Mistral 7B	Mixture-of-Experts
Python, Java, JavaScript, C++	0.1660	0.2500

Figure 4 presents the performance distribution of the Mistral 7B Base Model and the Mistral 4x7B MoE model. The concentration of MoE metrics in a narrow range suggests a more consistent output, indicating a higher level of specialization or efficiency in processing programming languages as

compared to the base model, and potential advantages in terms of uniformity and peak performance in coding-related tasks.

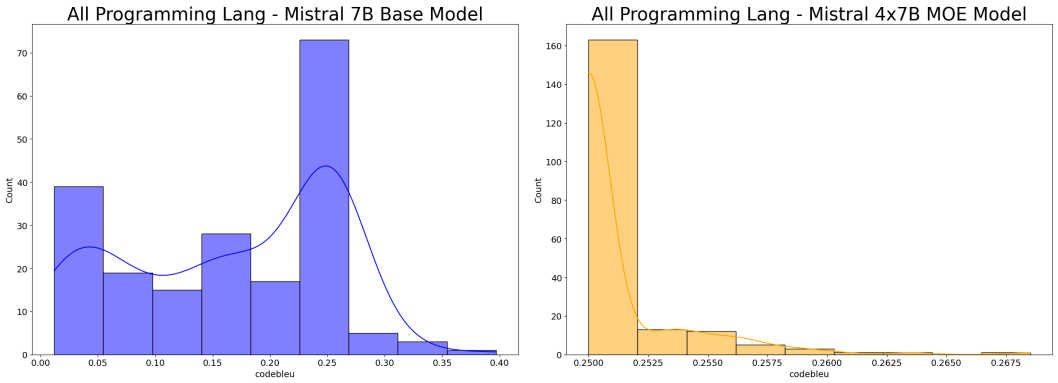

Figure 4: Performance distribution for MoE.

11 Discussion

Model Architecture and Expert Selection: Combining all expert models can provide reliable performance improvement, but we still observe some incorrect expert selection for certain problems. Java and JavaScript tokens could share similar hidden representations, making expert selection challenging. Our hard routing gating strategy needs further revision and improvement.

Evaluation Metrics and Benchmarks: Since we limited the maximum output length of new tokens to 256, it may be insufficient for programming languages like C++/Java to express solutions that can be achieved in fewer lines in Python and JS. For future evaluations, we should set the maximum output length to 650, as it is the default configuration used by the BigCode community.

Dataset Quality and Size: Volume of data does not necessarily translate to higher accuracy in output. Quality datasets are essential for code generation tasks, requiring textbook-level quality to fully understand syntax and semantic representations. Finetuning on a larger dataset of 100K-500K text-to-code samples did not improve model performance, as most of the data was noisy, worsening the base model's performance. Switching to a smaller, quality 10K text-to-code direct program-level dataset has shown significant improvement in accuracy.

Finetuning: Finetuning with snippet-level text descriptions for the same dataset, akin to data augmentation, is anticipated to further improve model accuracy and conversational abilities through instruction tuning. We ran finetuning at 88% GPU utilization with optimal hyper-parameters but encountered out-of-memory issues for other parameters, indicating good resource utilization. Despite these challenges, we believe that the MoE approach has advantages over traditional fine-tuning by fine-tuning the base model without negatively impacting other aspects.

12 Future Works

Enhanced Dataset Curation: Expanding the training data with diverse programming languages, coding styles, and task complexities is crucial. Incorporating domain-specific code snippets and leveraging program-level and snippet-level datasets can improve the model's understanding of complex syntax and logical data flows.

Refinement of Model Architecture: Exploring alternative gating mechanisms, such as softmax gating and noisy top-k gating, can enhance expert selection and performance. Experimenting with different MoE configurations, like varying the number of experts or architectures like dense MoE and switch transformers, can optimize efficiency and accuracy.

Expert Selection at Attention Layers: Currently, only the MLP layers utilize expert concepts. Exploring expert selection at the attention layers, where experts focus on different attention tasks,

could improve accuracy further. This allows leveraging specialized experts for attention-based operations, potentially leading to more accurate and context-aware code generation.

Evaluation Metrics Expansion: Incorporating additional metrics like MultiPL-E and MBPP with CodeBLEU can provide a more holistic evaluation of model performance. MultiPL-E measures functional correctness, and MBPP evaluates the ability to solve programming problems.

Instruction Tuning: Fine-tuning the model on specific instructions or prompts through prompt engineering and instruction-based fine-tuning can enhance its ability to understand and follow complex natural language descriptions accurately.

Alignment: RLHF (Reinforcement Learning from Human Feedback) or DPO (Direct Preference Optimization) techniques can improve the alignment between natural language descriptions and code snippets. RLHF aligns the model using a reward model trained on human feedback, while DPO directly optimizes the model's policy for preferred responses.

13 Conclusion

In addressing the challenge of low-quality code created by current multilingual code generation techniques, our research introduced a robust Mixture-of-Experts (MoE) framework. This framework decomposes a complex baseline model into smaller and specialized models for each programming language.

Our findings revealed that fine-tuning within the MoE framework significantly enhanced performance in Python, Java, and JavaScript, languages known for their consistent structural and explicit syntactical nature. However, the C++ expert model did not show the expected level of improvement, likely due to the language's inherent complexity and error-prone syntax.

Overall, the MoE approach led to a 51% improvement in performance over the original model, substantiating its effectiveness in enhancing code generation across multiple languages and thus meeting our primary goal.

The significance of our work lies in its potential to streamline code generation processes, offering substantial time and cost efficiencies in software development. The broad implementation of MoE models could drive innovation across various tech-driven industries. Future developments should focus on refining underperforming languages such as C++ and expanding the MoE framework to encompass a wider array of programming languages.

References

[1] **Mixtral of Experts**, arXiv:2401.04088

[2] **Gemini 1.5:** https://blog.google/technology/ai/google-gemini-next-generation-model-february-2024

[3] **Segmind's SegMoE** https://blog.segmind.com/introducing-segmoe-segmind-mixture-of-diffusion-experts/

[4] **SantaCoder: don't reach for the stars!**, arXiv:2301.03988

[5] **Phi-2:** `https://huggingface.co/microsoft/phi-2`

[6] **MoE Layer in Transformer Decoder:** `https://stackoverflow.blog/2024/04/04/how-do-mixture-of-experts-layers-affect-transformer-models/`

[7] **Outrageously Large Neural Networks:** arXiv:1701.06538

[8] **mergekit:** `https://github.com/arcee-ai/mergekit`

[9] **CodeBLEU: a Method for Automatic Evaluation of Code Synthesis**, `arXiv:2009.10297`