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

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.

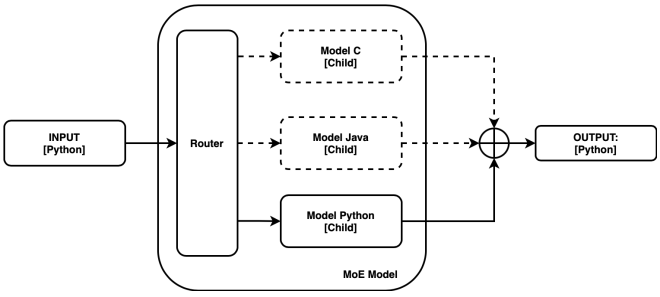

Figure 1: Applying MoE Architecture on Code Generation

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 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.

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

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.

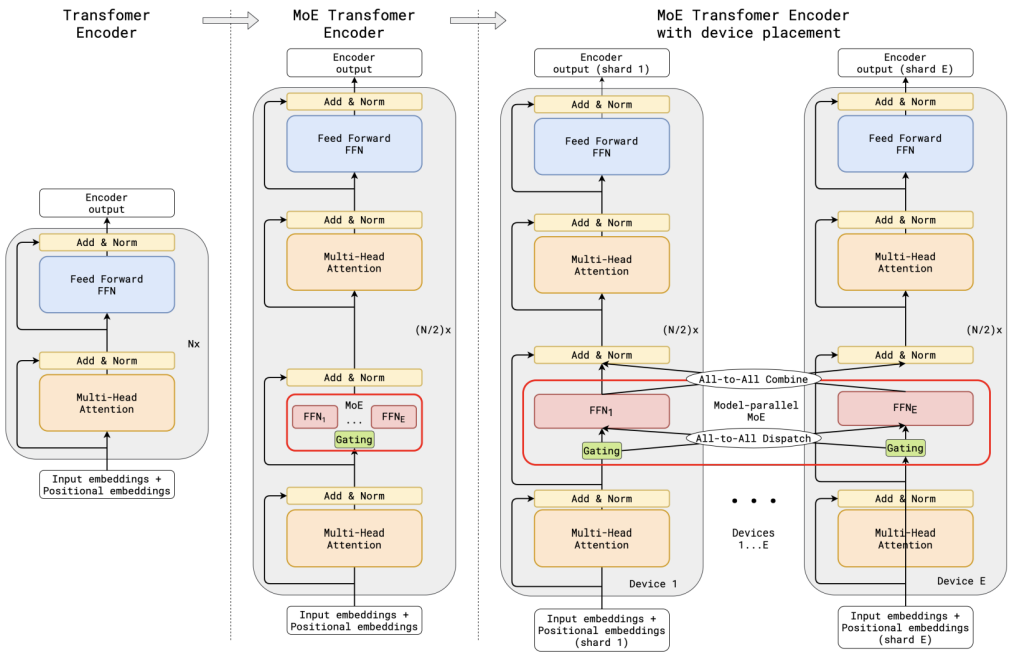

Figure 2: Illustration of scaling of Transformer Encoder with MoE Layers. The MoE layer replaces the every other Transformer feed-forward layer. Decoder modification is similar. (a) The encoder of a standard Transformer model is a stack of self-attention and feed forward layers interleaved with residual connections and layer normalization. (b) By replacing every other feed forward layer with a MoE layer, we get the model structure of the MoE Transformer Encoder. (c) When scaling to multiple devices, the MoE layer is sharded across devices, while all other layers are replicated. [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.

3.4 Evaluation Metrics

We leverage a system called **MultiPL-E** [8] to benchmark our model. MultiPL-E is a sophisticated and scalable framework specifically designed for evaluating neural code generation models over a variety of programming languages. This polyglot approach allows for a more comprehensive assessment of a model's ability to produce not only syntactically correct but functionally accurate code, which can be verified by passing predefined unit tests. This represents a significant advancement over existing benchmarks, which might focus on a narrower range of languages or primarily assess syntactic accuracy without considering functional correctness.

In MultiPL-E, models are tested across multiple programming languages, which broadens the applicability and relevance of the benchmark. The evaluation encompasses a set of diverse coding tasks, requiring models to generate code snippets that are then evaluated for their correctness through unit tests specifically designed for each task. This approach ensures that the code generated by the models meets real-world programming standards and performs the intended operations successfully.

The evaluation metrics within MultiPL-E, including pass@1, pass@10, and pass@100, are particularly designed to measure the functional accuracy of generated code:

pass@1: This metric assesses the probability that the first code completion generated by the model (at a lower randomness setting, temperature 0.2) passes all designated unit tests for a particular task. This measures the model's precision in producing correct solutions on the first try.

pass@10: This evaluates the likelihood that at least one of ten different completions (generated with a higher randomness setting, temperature 0.8 to encourage variety) passes all the unit tests. This metric provides insight into the model's ability to offer multiple valid solutions and its consistency in generating functionally correct code.

pass@100: Similar to pass@10, but it assesses the probability across one hundred different completions. This gives a broader understanding of the model's consistency and its capacity to generate a wide range of viable solutions.

These metrics focus on functional correctness, a critical aspect often overlooked in other benchmarks. By ensuring that the generated code not only compiles but also performs the specified task correctly, MultiPL-E provides a more realistic and rigorous evaluation of code generation models, reflecting their practical utility in real-world coding scenarios.

4 Baseline Selection

In our quest to ascertain the efficacy of the Mixture of Experts (MoE) approach within code generation, three pivotal attributes must be meticulously considered while selecting a foundational large language model (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.

5 Baseline Implementation Completeness

5.1 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.

5.2 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.

5.3 Creating Mixture of experts

To create Mixture-of-Experts (MoE) models, we utilize the mergekit [9] 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.

6 Proposed Extension

In our experiments, we explored two approaches for creating Mixture-of-Experts (MoE) models: model blending through merging techniques and expert model selection through a gating network. The baseline model was fine-tuned for Java, JavaScript, C++, and Python. One approach combined the fine-tuned models, while the other used a gating mechanism to select the best expert model for a given input.

We aimed to determine whether the Mistral-7B-Coder and Mistral-4x7B-Coder-MoE models performed better individually on various programming tasks across the four languages and how they compared to each other. We anticipated that Mistral-4x7B-Coder-MoE would learn the feature space of neural code generation tasks more effectively than the unified fine-tuned model Mistral-7B-Coder.

For evaluation, we will use the MultiPL-E system, a multi-programming language evaluation framework, to benchmark the models. MultiPL-E employs pass rates as a metric, assessing the probability of a model's output successfully solving a specified problem within a given number of attempts. This metric provides a quantifiable measure of a model's proficiency in generating functionally accurate code across diverse programming languages and tasks.

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] **GShard:** arXiv:2006.16668

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

[8] **MultiPL-E: Scalable and Extensible Approach to Benchmarking Code Generation** arXiv:2208.08227

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