SAPIENZA
UNIVERSITÀ DI ROMA

Do LLMs really need so much Attention?

Facoltà di Ingegneria dell'informazione, informatica e statistica

Corso di Laurea Magistrale in Data Science

Candidate

Matteo Candi
ID number 1884760

Thesis Advisor

Prof. Indro Spinelli

Co-Advisor

Dott. Marocchino Alberto

Academic Year 2023/2024

Thesis defended on INSERT EXAM DATE
in front of a Board of Examiners composed by:

Prof. Indro Spinelli (chairman)

Do LLMs really need so much Attention?
Master's thesis. Sapienza – University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: candi.1884760@studenti.uniroma1.it - mcandi79@gmail.com

Abstract

In the evolving landscape of software development, the translation of code between programming languages stands as a pivotal yet traditionally labor-intensive task. The advent of Large Language Models (LLMs) has heralded a transformative era, enabling the seamless conversion of code with unprecedented accuracy and efficiency. This thesis delves into the optimization of LLMs, specifically tailored for the intricate task of translating Java code into Python, a challenge of significant practical importance in the tech industry.

Our research used a novel approach to streamline the computational demands of LLMs, focusing on the strategic dilution of attention layers within the model architecture. By employing a technique based on entropy, we selectively bypass less critical attention layers, thereby reducing the model's complexity and computational footprint. This method not only aims to enhance the model's efficiency but also to maintain its high performance in code translation tasks.

Through a series of experiments, we evaluate the modified model against traditional benchmarks, including translation accuracy, model size, and resource consumption. Our findings promise to illuminate a path toward more accessible and sustainable LLM applications, particularly in resource-constrained environments. This thesis invites readers to explore the frontiers of AI in software development, offering insights that could redefine the boundaries of code translation and beyond.

Contents

Motivation

This thesis addresses a significant challenge encountered during my internship at KPMG, specifically focusing on the development of a web application designed for code translation from legacy programming languages to contemporary ones. The project started with an extensive exploration of various open-source language models (LLMs) that could fulfil the requirements of the task at hand. A critical aspect of this exploration involved striking a balance between performance and resource efficiency, particularly given the substantial computational power these models demand.

After conducting several tests on the mapping capabilities of different programming languages and evaluating the outcomes, we ultimately selected two models that demonstrated satisfactory performance for our specific needs. One of the models we decided to implement in this thesis is the *OpenCodeInterpreter-DS-6.7B*. This model has shown particularly impressive results in coding-related tasks, making it a suitable choice for our project. Despite its comparatively modest size, 6.7 billion parameters, relative to some of the larger models available in the market, it still necessitates significant GPU memory, with requirements reaching up to 25 GB when utilizing float32 precision. Research has indicated that while high precision is advantageous during the training phase of a model, it is not strictly essential for inference. Consequently, we opted to employ the model's float16 version, thereby reducing the memory consumption to approximately 12.5 GB. While this is relatively small compared to larger models, it still imposes considerable resource demands for personalized usage. Initially, I attempted to utilize a virtual machine equipped with multiple GPUs. However, this approach proved to be cost-prohibitive and resulted in suboptimal performance, as the distributed system significantly slowed down model inference. Therefore, the most effective solution was to use a virtual machine featuring a single, more powerful GPU, which allowed the model to be efficiently uploaded while still providing sufficient capacity for necessary operations during inference. Following the establishment of this infrastructure, the subsequent step involved integrating the web application with the new virtual machine hosting the models. This integration optimized the entire process and enhanced the web application's functionality, enabling it to better accommodate the utilization of the selected models. Additionally, we developed metrics to assess and report the performance statistics for each request, thereby facilitating a more robust user experience.

Once the system was operational, we turned our attention to the potential for cost reduction, as the expenses associated with the virtual machine were significant. We aimed to improve the speed of the processes in place while also exploring avenues for reducing costs. This exploration culminated in the project described in this thesis, which aims to investigate the feasibility of achieving our research objectives through the development of a faster and more lightweight model. This initiative not only

seeks to optimize resource usage but also aims to expand the applications of these advanced technologies, facilitating new and innovative utilizations.

Chapter 1

Introduction

The advent of LLMs has revolutionized the way we approach numerous complex tasks, offering solutions that were previously difficult to achieve with conventional methods. One notable example is code translation, which involves converting code from one programming language to another. Traditionally, this task required significant manual effort and domain-specific expertise. However, LLMs have dramatically simplified this process by leveraging their deep learning capabilities to understand and translate code with high accuracy and efficiency. Prior to the rise of this models, code translation often involved rule-based systems and heuristic methods, which could be cumbersome and limited in their ability to handle diverse and complex codebases. These early methods struggled with the variety of different programming languages and often required extensive customization to address specific language features. The introduction of LLMs, with their ability to learn from vast amounts of data and capture intricate patterns in code, has transformed code translation into a more streamlined and automated process.

Today, they can analyze and generate code across multiple languages with remarkable ease, thanks to their sophisticated architecture and training on diverse code datasets. This advancement not only reduces the time and effort required for code translation but also improves the quality of the translated code by preserving its functionality and structure. The ability of LLMs to handle such tasks with minimal manual intervention represents a significant leap forward, making code translation more accessible and efficient than ever before. Despite these benefits, the growing complexity of large language models brings its own set of challenges. The need for substantial computational resources, large model sizes, and long inference times can limit the practical deployment of these models in resource-constrained environments. Addressing these challenges while retaining the models' effectiveness is a critical area of ongoing research and development. One of the key factors contributing to the high computational demands of LLMs is their extensive use of attention mechanisms. Attention layers enable the model to weigh different parts of the input

data differently, which is crucial for tasks that involve understanding and generating sequences. Despite their effectiveness, these layers add to the model's parameter count and computational overhead. For instance, the number of parameters in attention layers scales with the product of the input and output dimensions, leading to substantial resource requirements (self-attention mechanism present a quadratic complexity with the input dimension). To address these issues, this thesis explores an approach to optimizing LLMs by selectively diluting certain attention layers. The proposed method leverages a technique called NOSE (Entropy-based Selection Strategy), which identifies and targets attention layers that have a lesser impact on the model output. By configuring these layers to return their inputs directly rather than performing complex computations, we aim to simplify the model's architecture and reduce its computational footprint. The core of our approach involves modifying the attention layers so that they effectively bypass their typical processing. This modification allows us to retrain only the latter part of the decoder layer, which we hypothesize will be sufficient to retain the model's performance while achieving reductions in size and inference time. The rationale behind this method is that certain attention layers can be reduced to an identical mapping with a retrain of the subsequent MLP layer and the loss of performance can be very little, if any.

The model selected for this study is optimized for code translation, with a focus on converting Java code to Python. This task is chosen due to its relevance in the software development industry, where the ability to smoothly translate between programming languages can significantly enhance productivity and interoperability. To rigorously assess the effectiveness of our approach, we conduct experiments on the code translation task and evaluate the performance of the modified model using a range of metrics. These include translation accuracy, model size, parameter count, and resources consumption. By comparing these metrics with those of the original, unmodified model, we aim to determine whether our approach can achieve a balance between reduced computational demands and maintained performance quality.

The structure of this thesis is as follows:

- **Chapter 1: Introduction**
  This chapter provides an overview of the motivation behind the study, focusing on the challenges and opportunities presented by LLMs in the context of code translation. It outlines the specific problem addressed by the thesis and the approach taken to solve it.
- **Chapter 2: Literature Review**
  This chapter reviews the existing literature on transformers, focusing on their application in both NLP and Computer Vision. It explores previous efforts to optimize transformer models, particularly through methods like token pruning and attention layer reduction. The review also covers the potential applicability of these methods in LLMs.

- **Chapter 3: Theoretical Framework**
  This chapter delves into the theoretical foundations of the techniques employed in the study. It discusses the underlying principles of transformers architecture, attention mechanisms, entropy-based selection strategies (NOSE), and the rationale for their integration into the transformer architecture, with a focus also on the metrics used for evaluation.
- **Chapter 4: Methods and Results**
  This chapter details the experimental setup, including the design of the modified transformer model, the datasets used for training and testing, and the specific metrics employed for evaluation. The results of the experiments are presented, comparing the performance of the different versions with that of the original.
- **Chapter 5: Conclusions and Future Works**
  This final chapter summarizes the findings of the study, highlighting the contributions made to the field of LLM optimization. It also discusses potential areas for future research, including further refinement of the used approaches.

Chapter 2

Literature Review

The concept of transformers has significantly transformed both the domains of natural language processing (NLP) and computer vision (CV). for NLP, transformers have demonstrated superior performance in various tasks, largely due to their self-attention mechanism, which effectively captures dependencies across input sequences. Dosovitskiy et al. (2020) [1] extended this architecture to vision with the Vision Transformer (ViT), achieving state-of-the-art results in image classification by treating image patches as sequences of tokens.

Vision transformers have shown exceptional performance on image recognition tasks, particularly when applied to large datasets like ImageNet-1k. However, their success comes at the cost of high computational demands due to the quadratic complexity of the self-attention mechanism. This issue has led researchers to explore various strategies for reducing the computational burden without sacrificing performance. Techniques such as token aggregation and token pruning have been proposed, aiming to optimize the processing by focusing only on the most informative parts of the input data.

One innovative approach in this area, discussed by Lin et al. (2024) [3], involves simplifying ViTs by selectively removing non-essential attention layers. This method, guided by entropy considerations, integrates the information from these attention layers into the subsequent multi-layer perceptron layers, effectively reducing the model's complexity and memory footprint. The authors argue that certain MLP layers in ViTs possess sufficient capacity to replace the functions of some attention layers, particularly in the lower blocks where attention maps are less informative. This insight is leveraged to create more efficient models that maintain performance while using fewer resources. The *MLP Can Be A Good Transformer Learner*[3] paper suggests that by integrating uninformative attention layers into MLP layers, the resulting architecture can achieve comparable performance to full transformer models with a significant reduction in parameters and computational load. The proposed method demonstrates its effectiveness on various benchmarks, including

ImageNet-1k and CIFAR-100, showing that it is possible to remove up to 40% of the attention layers without compromising accuracy.

This finding is particularly relevant in the context of applying transformer techniques from Computer Vision to Natural Langauge Processing. The potential of MLPs to substitute parts of the attention mechanism in transformers suggests that similar strategies might be applicable in the NLP domain. By exploring how these techniques, originally developed for visual transformers, could be adapted for language models, it may be possible to create more efficient NLP models that retain the benefits of transformers while reducing their computational costs.

Another critical aspect of the study presented in the paper is the examination of the generalization capabilities of the modified transformer models. The authors conducted transfer learning experiments on CIFAR-100, revealing that the models retained strong performance even when a significant portion of the attention layers were removed. This suggests that the features learned by the remaining layers were robust and generalized well to different tasks. Such findings highlight the potential for applying these techniques in NLP, where generalization across various language tasks is crucial.

The exploration of whether techniques used in visual transformers can enhance the performance and efficiency of large language models (LLMs) represents a promising research direction. The work by Lin et al. (2024) provides a strong foundation for this inquiry, demonstrating that MLPs can effectively replace parts of the attention mechanism in ViTs without losing performance. Extending this approach to NLP could lead to more efficient LLMs, potentially revolutionizing the field by making transformer models more accessible and easier to deploy in resource-constrained environments. Further research will be needed to fully understand the implications and optimize these techniques for the specific challenges posed by language data.

Chapter 3

Theoretical Framework

3.1 LLMs and Transformers Architecture

Large Language Models (LLMs) have revolutionized the field of natural language processing (NLP) by providing sophisticated and effective mechanisms for text generation. The architecture of these models, particularly those based on the Transformer architecture, has played a critical role in their success. This subsection outlines the general architecture of LLMs, emphasizing their components and functionalities in the context of text generation.

The foundation of most modern LLMs is the Transformer architecture, introduced by Vaswani et al. in 2017 [9]. This architecture consists of an encoder-decoder structure (Figure 3.1), although many LLMs used for text generation utilize only the decoder part. Large language models are constructed from multiple interconnected blocks of attention layers, which include sophisticated mechanisms such as self-attention, multi-head attention, and feed-forward neural networks. The self-attention mechanism allows the model to weigh the significance of different words within a context by computing attention scores, thereby capturing relationships between words regardless of their distance in the text. Multi-head attention further enhances this capability by running several attention operations in parallel, enabling the model to attend to different aspects of the information simultaneously. Feed-forward networks, positioned after the attention layers, transform the aggregated information to capture more complex patterns and nuances. These components, coupled with layer normalization and residual connections to ensure stable and efficient training, work in concert to process and generate coherent, contextually relevant human-like text. This intricate architecture empowers LLMs to perform a wide array of natural language processing tasks with remarkable accuracy and versatility, from text generation and translation to question answering and beyond. The key components of the Transformer architecture are listed and described below.

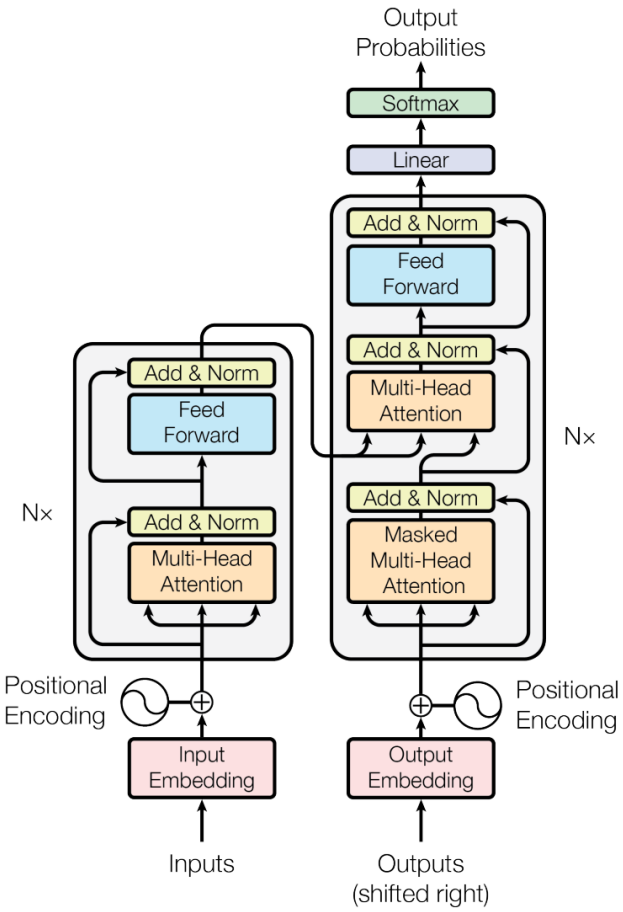

Figure 3.1. Transformers Architecture

Tokenization Tokenization is a fundamental preprocessing step in the pipeline of large language models (LLMs). It involves converting raw text into smaller units, called tokens, which are the basic building blocks that the model processes. Effective tokenization is crucial for the performance of LLMs as it directly impacts the model's ability to understand and generate text. There are several approaches to tokenization, each with its own advantages and trade-offs:

- Word Tokenization
- Subword Tokenization
  - Byte Pair Encoding (BPE)
  - WordPiece
- Character Tokenization

Each of these techniques has its own method for splitting text and is chosen based on the specific requirements of the language model and the characteristics of the

input data.

Embeddings Embeddings are a crucial component in the architecture of large language models, as they provide a dense representation of tokens that capture their semantic meanings. Embeddings convert discrete tokens, such as words or subwords, into continuous vectors that the model can process effectively.

The base strategy is the Word Embedding that map each word in the vocabulary to a fixed-size vector. Early techniques like Word2Vec and GloVe created static embeddings, where each word has a single vector regardless of context. These embeddings capture semantic similarities, meaning that words with similar meanings have vectors that are close in the embedding space.

Modern LLMs use Contextualized Embeddings, which vary depending on the context in which a word appears. This approach allows the model to handle polysemy (words with multiple meanings) effectively. Techniques like ELMo, BERT, and GPT generate embeddings dynamically during the model's forward pass, ensuring that the representation of a word adapts to its surrounding words.

In the Transformer architecture, an embedding layer is the first layer of the model. It converts the input tokens into embeddings and adds positional encodings to incorporate information about the token positions within the sequence. The combined embeddings are then fed into the subsequent layers of the model. Mathematically, the embedding process can be represented as:

$$\mathbf{E}(x_i) = \text{EmbeddingMatrix}(x_i) + \text{PositionalEncoding}(i)$$

where $\mathbf{E}(x_i)$ is the embedding of the i-th token x_i, and PositionalEncoding(i) adds positional information.

Embeddings are essential for transforming discrete text data into a form that neural networks can process. They capture syntactic and semantic relationships between tokens, enabling the model to understand and generate coherent and contextually appropriate text. By leveraging embeddings, LLMs can perform various NLP tasks, such as language modeling, translation, and text classification, with high accuracy and efficiency.

Positional Encoding In the transformer architecture, unlike recurrent or convolutional neural networks, there is no inherent notion of the order of the input tokens. However, understanding the order of words in a sentence is crucial for capturing the meaning and context of the text. To address this, positional encoding is introduced, providing the model with information about the position of each token in the sequence.

The positional encoding vectors are added to the input embeddings to incorporate the positional information. The encoding is designed to allow the model to learn the relative positions of the tokens.

The positional encodings PE are generated using sine and cosine functions of different frequencies. The formula for computing the positional encoding for a position pos and a dimension i is given by:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right) \quad PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right)$$

where pos is the position of the token in the sequence, i is the dimension, and d_{model} is the dimensionality of the model. These functions produce unique encodings for each position, and the values ensure that tokens at different positions receive distinct positional encodings. The alternating sine and cosine functions help the model to distinguish between even and odd positions, capturing the sequential nature of the data.

The intuition behind using sine and cosine functions is that they provide a smooth, continuous encoding of positions, which can generalize to sequences of different lengths. Additionally, these functions allow the model to learn to attend to relative positions, as the positional encodings exhibit predictable patterns. By adding these positional encodings to the input embeddings, the Transformer model can leverage both the content of the tokens and their positions within the sequence. This combination enables the model to understand the order and structure of the text, which is essential for generating coherent and contextually appropriate outputs.

Self-Attention Mechanism The self-attention mechanism is a crucial component of the Transformer architecture, allowing the model to dynamically weigh the importance of different words in a sentence. This mechanism captures the dependencies between words regardless of their distance in the sequence, facilitating a better understanding of context.

Given an input sequence of word embeddings $X = \{x_1, x_2, \ldots, x_n\}$, the self-attention mechanism transforms these embeddings into three distinct representations: queries Q, keys K, and values V. These representations are obtained through learned linear transformations:

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V$$

where W^Q, W^K, and W^V are weight matrices that are learned during training phase or updated while fine-tuning the model.

The attention scores are computed by taking the dot product of the query with all keys, scaled by the square root of the dimensionality of the keys (denoted as d_k):

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Here, the dot product QK^T produces a matrix of scores, which are then scaled and passed through a softmax function to obtain the attention weights. These weights are used to compute a weighted sum of the values V, producing the final output of the self-attention mechanism.

The scaling factor $\sqrt{d_k}$ helps mitigate the effect of having large dot product values, which can lead to small gradients and slow training. The softmax function ensures that the attention weights are normalized and sum to one.

Multi-Head Attention

To improve the model's capacity to focus on different aspects of the input, multi-head attention is employed. This involves applying the self-attention mechanism multiple times with different learned projections and then concatenating the results:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$

where each head_i represents an independent self-attention operation, and W^O is an output weight matrix. By using multiple heads, the model can capture various relationships and features in the data.

In summary, the self-attention mechanism computes the relevance of each word in the input sequence to every other word, enabling the model to focus on different parts of the input when generating text. The key steps involve transforming the input embeddings into queries, keys, and values, computing scaled dot-product attention, and applying multi-head attention for richer representations.

Feed-Forward Network Each layer of the Transformer architecture includes a feed-forward network (FFN) that processes the information obtained from the self-attention mechanism. The feed-forward network is applied independently to each position within the sequence, providing additional non-linearity and enabling the model to learn complex transformations of the input data.

The feed-forward network in the Transformer consists of two linear transformations with a general activation function σ in between. The FFN can be described mathematically as follows:

$$\text{FFN}(x) = \sigma(xW_1 + b_1)W_2 + b_2$$

where x is the input, W_1 and W_2 are weight matrices, b_1 and b_2 are bias terms, and σ is the activation function. Common choices for σ include ReLU (Rectified Linear Unit), GELU (Gaussian Error Linear Unit), or other non-linear activation functions.

The dimensionality of the FFN is crucial for its performance. Typically, the its input and output dimensionality match the model dimension d_{model}, while the inner layer has a higher dimensionality. This increase in dimensionality allows the FFN to capture and process richer features before projecting them back to the model dimension.

An important characteristic of the FFN in Transformers is that it is applied position-wise and independently to each token in the sequence. This means that the same FFN parameters are used for all positions, ensuring that the network can generalize across different parts of the sequence. This position-wise application can be parallelized, making the model efficient in terms of computation.

Activation Functions Activation functions are a crucial component in neural networks, playing a significant role in determining the model's training dynamics and overall performance. These functions introduce non-linearity into the model, allowing it to capture and represent complex patterns within the data. Without non-linearity, the model would be limited to learning only linear relationships, which would drastically reduce its ability to perform on more complex tasks.

One key aspect of activation functions is their ability to prevent the outputs of neurons from shrinking towards each other, ensuring that the network can maintain distinct and diverse activations across different layers. This helps in preserving the richness of the information as it propagates through the network, enabling the model to learn more nuanced features from the input data.

Several factors should be considered when selecting an activation function. One of the most important is Gradient Flow. In deep networks, it's crucial to maintain a healthy gradient flow during backpropagation to avoid issues like vanishing or exploding gradients, which can hinder the training process. Activation functions like ReLU (Rectified Linear Unit) and GELU (Gaussian Error Linear Unit) are favored in deep architectures because they help preserve the gradient, ensuring that the model continues to learn effectively as it grows deeper.

Computational Efficiency is another consideration. Some activation functions are computationally simpler and faster to compute, which is especially important when training large models. For instance, ReLU is popular not only because it supports gradient flow but also because it is computationally efficient, requiring minimal resources and time to calculate.

Finally, Performance on Task is a critical factor. The choice of activation function can have a significant impact on the empirical performance of the model for specific tasks. For example, GELU is often preferred in transformer models due to its

smoothness and empirical success in natural language processing tasks. Its ability to produce smoother transitions between activations allows for more nuanced and accurate modeling of complex language patterns.

Layer Normalization Layer normalization is a fundamental technique used in deep neural networks to stabilize and accelerate the training process, particularly in models with complex architectures such as transformers. It operates by normalizing the inputs across the features for each individual training example, ensuring that the inputs to each layer are consistent and well-behaved, regardless of the scale of the original input data. This consistency is crucial for maintaining stable training dynamics, especially in deep networks where small variations in the input can be amplified as they propagate through the layers.

Layer normalization differs from batch normalization, another popular normalization technique, in several key ways. While batch normalization normalizes the inputs across the batch dimension, meaning it calculates the mean and variance using all the data points in a mini-batch, layer normalization focuses on normalizing within a single training example, across its features. This distinction is important because it makes layer normalization particularly well-suited for scenarios where batch sizes are small or variable, such as in recurrent neural networks (RNNs) or in certain types of online learning, where the model is trained on one example at a time.

One of the primary benefits of layer normalization is its ability to mitigate the problem of internal covariate shift, a phenomenon where the distribution of inputs to a given layer changes during training as the parameters of the previous layers evolve. By normalizing the inputs, layer normalization ensures that the mean and variance of the activations remain consistent across training, which reduces the amount of compensatory adjustment the model needs to make, leading to faster convergence and improved training stability.

Mathematically, layer normalization is expressed as:

$$\text{LayerNorm}(x) = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}\gamma + \beta$$

In this equation, μ represents the mean of the input x, and σ^2 represents the variance. These are calculated across the features for each individual training example. The small constant ϵ is added to the denominator for numerical stability, preventing division by zero or extremely small numbers that could destabilize the training process. The parameters γ and β are learnable, meaning they are adjusted during training. These parameters allow the model to scale and shift the normalized values, giving it the flexibility to maintain the expressive power of the original unnormalized activations.

In practice, layer normalization is applied in several critical points within neural

network architectures. For instance, in transformer models, layer normalization is typically applied after the self-attention mechanism and again after the feedforward neural network (FFN) within each layer. This repeated application ensures that the outputs of these complex subcomponents are normalized, which helps maintain training stability and improves the overall performance of the model.

Residual Connections Residual connections, also known as skip connections, are a critical architectural feature in deep neural networks, particularly in very deep models like ResNets and transformers. These connections are designed to address the challenges associated with training deep networks, most notably the problem of vanishing gradients, which can impede effective learning as the network depth increases.

In traditional deep networks, as gradients are backpropagated through many layers, they can diminish significantly, leading to very small updates to the weights in the earlier layers. This phenomenon, known as the vanishing gradient problem, can severely limit the network's ability to learn, especially in very deep architectures. Residual connections help to mitigate this issue by allowing the gradients to flow more directly through the network, bypassing one or more layers.

The key idea behind residual connections is to add the input of a layer directly to its output, effectively creating a shortcut path for the information. This ensures that the original information can pass through the network unchanged if necessary, which is particularly useful when the current layer's output should closely resemble its input. By preserving the input information, residual connections facilitate the learning of identity mappings, where the desired transformation is minimal, allowing the network to focus on learning the residual (or difference) between the input and output rather than the entire transformation.

Mathematically, a residual connection can be expressed as:

$$\text{Output} = x + \text{SubLayer}(x)$$

where x represents the input to the layer, and $\text{SubLayer}(x)$ denotes the output of the sub-layer, which could be the result of operations such as self-attention or a feedforward neural network (FFN). The addition operation here is element-wise, meaning that each element of the input vector x is added to the corresponding element of the sub-layer's output.

This simple yet powerful modification has several important effects:

- **Improved Gradient Flow:** By providing a direct path for the gradient to flow back through the network, residual connections help to maintain strong gradients even in very deep networks. This helps prevent the gradients from

vanishing as they propagate through many layers, making it easier to train deep models.

- **Ease of Optimization:** Residual connections make it easier to optimize deep networks by simplifying the learning process. Instead of having to learn the full transformation from input to output, the network only needs to learn the residual function, which is often easier and faster to converge.
- **Flexibility in Network Depth:** Residual connections enable the construction of much deeper networks without suffering from degradation in performance, which is the phenomenon where adding more layers leads to worse training and testing accuracy. This flexibility allows for more expressive models that can capture more complex patterns in the data.

In transformer models, residual connections are typically used in conjunction with layer normalization, applied both after the self-attention mechanism and after the feedforward neural network (FFN) within each layer. The combination of residual connections and normalization ensures that the network maintains stable training dynamics and can effectively learn deep representations.

3.2 Entropy-based Selection Strategy (NOSE)

3.2.1 Entropy

Entropy is a fundamental concept in information theory, introduced by Claude Shannon in 1948. It quantifies the uncertainty or randomness in a probability distribution and is a measure of the amount of information required to describe a random variable. In the context of large language models (LLMs), entropy plays a crucial role in understanding the model's output, particularly in relation to the distribution of probabilities assigned to different tokens in a sequence.

Mathematically, the entropy $H(X)$ of a discrete random variable X with probability distribution $P(x)$ is defined as:

$$H(X) = -\sum_{x \in \mathcal{X}} P(x) \log P(x) \tag{3.1}$$

where $\mathcal{X}$ is the set of all possible outcomes of X, and $P(x)$ is the probability of outcome x. The logarithm is typically taken base 2, yielding entropy in units of bits, though it can also be in natural logarithm (base e) to express entropy in nats.

In large language models, entropy is particularly relevant for analyzing the model's confidence and uncertainty in generating or predicting the next token in a sequence. The output of an LLM for a given input is a probability distribution over the model's vocabulary, where each token is assigned a probability that reflects the model's estimate of how likely that token is to be the next in the sequence.

A lower entropy value in the output distribution indicates that the model is highly confident in its prediction, meaning that it assigns a high probability to one or a few tokens while assigning low probabilities to others. Conversely, a higher entropy value suggests greater uncertainty, where the model spreads its probabilities more evenly across many tokens. This uncertainty can arise from ambiguous input, lack of context, or the inherent complexity of the language.

For instance, when the model predicts the next word in a sentence, if the context clearly indicates what the next word should be, the entropy of the output distribution will be low, reflecting high confidence. However, if the context is ambiguous, the model might produce a distribution with higher entropy, indicating that multiple next words are plausible.

Entropy also influences the sampling techniques used to generate text from LLMs. In methods like *temperature sampling*, the model's output probabilities are adjusted by a temperature parameter τ, which effectively scales the entropy of the distribution:

$$P'(x) = \frac{P(x)^{1/\tau}}{\sum_{x' \in \mathcal{X}} P(x')^{1/\tau}} \tag{3.2}$$

where $P'(x)$ is the adjusted probability and τ is the temperature. When τ is low (e.g., $\tau < 1$), the model's output becomes more deterministic, reducing entropy and making high-probability tokens more likely to be selected. When τ is high (e.g., $\tau > 1$), the output distribution becomes more uniform, increasing entropy and allowing for more diverse token selection, which can result in more creative or varied text generation.

In practical applications, entropy can be used to guide the generation process, control diversity in output, or even assess model performance.

Entropy is a powerful concept that underpins much of the probabilistic reasoning in large language models. It quantifies the uncertainty in model outputs, influences sampling strategies, and is closely linked to model evaluation metrics like perplexity. By understanding and controlling entropy, practitioners can better manage the behavior of LLMs, tailoring their outputs to specific applications and desired outcomes.

Entropy is not only important for analyzing the outputs of LLMs but also plays a crucial role during their training. The entropy of a specific layer can be calculated based on the probability distribution of its features:

$$H(F) = -\int p(f) \log p(f)\, df, \quad f \in F. \tag{3.3}$$

However, it is often challenging to directly determine the probability distribution of a feature map, $p(f)$ for $f \in F$. To address this, a common approach, as discussed in

studies such as [7], is to approximate the probability distribution of the intermediate features in a layer using a Gaussian distribution.

Accordingly, the entropy of a layer can be approximated by considering the mathematical expectation of $F \sim \mathcal{N}(\mu, \sigma^2)$:

$$H(F) = -\mathbb{E}[\log \mathcal{N}(\mu, \sigma^2)] = \log(\sigma) + \frac{1}{2}\log(2\pi) + \frac{1}{2} \tag{3.4}$$

where σ represents the standard deviation of the feature set $f \in F$. In practice, a batch of data (e.g., images or text sequences) is processed through a model to extract the feature set F from a specific layer, whether it be an attention layer or an MLP layer. The entropy $H(F)$ is then proportional to $\log(\sigma)$ plus two constants, which are often omitted in practical computations for simplicity.

This entropy calculation is typically performed for each dimension within the intermediate feature set. Therefore, neglecting constant terms, the entropy of each layer is approximately proportional to the sum of the logarithms of the standard deviations of each feature dimension:

$$H_\sigma(F) \propto \sum_j \log[\phi(F_j)] \tag{3.5}$$

where $\phi(F_j)$ computes the standard deviation of the j-th dimension of the feature set F.

3.2.2 Transfer Entropy

In large language models, understanding how information is processed and transferred between layers can offer significant insights into the model's inner workings. Transfer entropy, a measure of directed information flow, is particularly useful for this purpose. It quantifies the amount of information transferred from one layer to another and can help evaluate the functional dynamics of different layers in an LLM.

Transfer entropy, a concept rooted in information theory, provides a non-parametric measure of the directed transfer of information between two stochastic processes. Introduced by Thomas Schreiber in 2000, transfer entropy has found applications across various fields, from neuroscience to climate science, offering insights into the dynamic interactions and dependencies between systems. Transfer entropy quantifies the reduction in uncertainty about the future state of a target process, given the knowledge of the past states of both the target and the source process. Mathematically, for two discrete stochastic processes X and Y, the transfer entropy from X to Y is defined as:

$$TE(X \to Y) = H\left(Y_t \mid Y_{t-1:t-L}\right) - H\left(Y_t \mid Y_{t-1:t-L}, X_{t-1:t-L}\right) \tag{3.6}$$

where $H(\cdot)$ is the entropy defined in the equation (3.1), Y_t is the current state and $Y_{t-1:t-L}$ and $X_{t-1:t-L}$ are the past state if the respective processes.

In the context of Large Language Models, transfer entropy can be a powerful tool to evaluate the influence of one layer on another, shedding light on the flow of information and the hierarchical dependencies within the model architecture. It is useful, for example, to evaluate the influence that a selected layers has on a target one. Reformulating the equation (3.6) in the llms the transfer entropy can be calculated as:

$$TE = H(F_{\text{target}}) - H(F_{\text{target}} \mid A \setminus \text{Attn}_{\text{source}}) \tag{3.7}$$

with $H(F_{\text{target}})$ representing the original entropy of the target layer and the second factor the entropy of the same layer obtained, this time, masking the $\text{Attn}_{\text{source}}$, e.g. setting the attention layer mechanism as identical mapping.

Therefore, the numerical value of TE provides insight into the extent to which information is transferred from the source layer to the target layer, effectively reflecting the significance of the source layer's influence on the target layer. This measurement captures the degree of correlation between the layers, allowing us to assess how the information from one layer contributes to or affects the processing in another layer. In practical terms, a higher value of TE indicates a stronger influence or higher significance of the source layer in shaping the activations of the target layer, while a lower value suggests a weaker influence or less correlation.

In this way it's possible to analyze and identify combinations of multiple attention layers within the model that exhibit minimal correlation with the final output layer. By doing so, we can determine which attention layers have the least impact on the final output, providing valuable insights into how information flows through the network. This can help in optimizing the model by focusing on the most relevant layers and potentially reducing the influence of layers that do not significantly contribute to the output, thus improving model efficiency and interpretability.

3.2.3 Algorithm

The Entropy-based Selection Strategy (NOSE) is an algorithm, proposed in the paper [3], designed to optimize feature selection by leveraging entropy measures to identify the most informative attributes. In many machine learning tasks, choosing the right subset of features is crucial for model performance, and NOSE provides a systematic approach to achieve this. By iteratively evaluating the contribution of each attribute to the overall entropy of the model, NOSE selects the attributes that offer the greatest potential for improving model understanding and performance. This method is particularly useful in scenarios where reducing dimensionality while retaining critical information is essential.

Algorithm 1 Entropy-based Selection Strategy (NOSE)

Input: model, samples, N
Output: S

$S \leftarrow \emptyset$
$C \leftarrow \{\text{Attn}_1, \text{Attn}_2, \ldots, \text{Attn}_l\}$

for each iteration in N **do**
  $TE \leftarrow \infty$
  $Attn_{idx} \leftarrow -1$

  for Attn_i in S **do**
    $\text{Attn}_i \leftarrow \mathbf{I}$
  end for

  $BE \leftarrow$ average base entropy

  for Attn_i in C **do**
    $\text{Attn}_i \leftarrow \mathbf{I}$
    $E \leftarrow$ average entropy
    $TE_i \leftarrow BE - E$

    if $TE_i < TE$ **then**
      $Attn_{idx} \leftarrow i$
      $TE \leftarrow TE_i$
    end if

  end for

  $S \leftarrow S \cup Attn_{idx}$
  $C \leftarrow C \setminus \{\text{Attn}_{idx}\}$

end for

return S

The algorithm (reported in Algorithm 1) starts with an empty set and a full set of candidate attention mechanisms. It then iterates through a specified number of rounds, where each candidate is evaluated based on how much it reduces uncertainty, measured by entropy. For each round, the algorithm calculates how including each candidate changes the model's entropy compared to the base entropy. The attention mechanism that results in the greatest reduction in entropy is chosen and added to the set of selected attributes, and then removed from the candidate pool. After completing the iterations, the algorithm returns the final set of indexes of the selected layers. The objective is to identify and keep the layers that have the less influence on the final output layer.

3.3 Model Shrinking

In neural network architectures, attention mechanisms play a crucial role in directing the model's focus towards significant features within the input data. This capability enhances the model's performance by allowing it to prioritize and emphasize the most relevant aspects. However, there are scenarios where reducing or completely eliminating the influence of the attention mechanism can be advantageous. This is achieved through techniques that adjust how attention outputs are combined with the original input data.

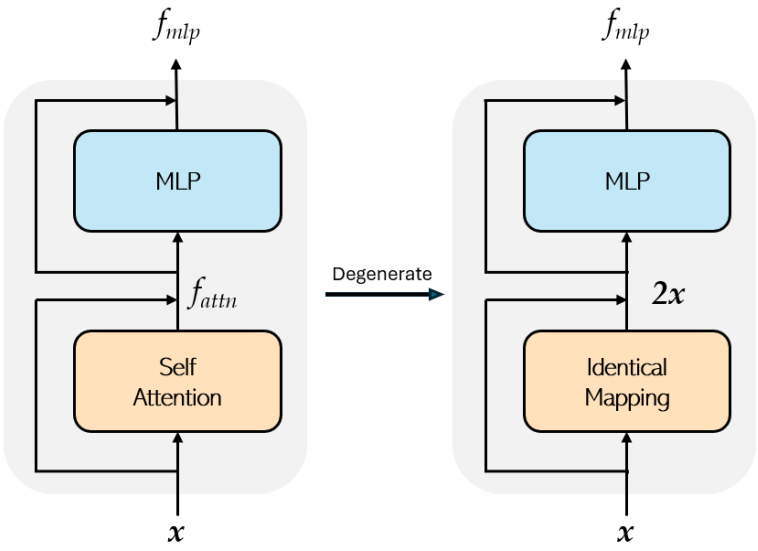

Figure 3.2. Self Attention layer degeneration

One method for controlling the impact of attention mechanisms involves blending the attention output with the original input (Fig 3.2). This process can be described by the following formula:

$$f_{\text{attn}} = M \odot \text{Attn}(\mathbf{x}) + \mathbf{x} \tag{3.8}$$

Here, M is a mask matrix of dimensions $(P+1) \times d$, and $\text{Attn}(\mathbf{x})$ denotes the output from the attention mechanism. The mask matrix M regulates the degree to which the attention output contributes to the final result compared to the original input $\mathbf{x}$. The symbol $\odot$ represents element-wise multiplication, allowing for precise control over the blending of attention features with the input data.

To refine this adjustment, the formula is further modified as follows:

$$\hat{f}_{\text{attn}} = M \odot \text{Attn}(\mathbf{x}) + (1 - M) \odot \mathbf{x} + \mathbf{x} \tag{3.9}$$

Simplifying this expression gives:

$$\hat{f}_{\text{attn}} = M \odot \text{Attn}(\mathbf{x}) + (2 - M) \odot \mathbf{x} \tag{3.10}$$

This revised formula allows for a dynamic adjustment of the attention mechanism's influence. By blending the attention output with the original input data in this manner, the technique facilitates a gradual reduction or complete elimination of the attention mechanism's effect, depending on the desired outcome.

This approach offers flexibility in how the model integrates attention outputs with the original input, making it possible to tailor the model according to specific needs and objectives. Whether aiming to diminish the attention mechanism's impact or phase it out entirely, this method provides precise control over its contribution to the model's predictions.

3.4 Metrics

3.4.1 BLEU

The BLEU (Bilingual Evaluation Understudy) metric is one of the most widely used methods for evaluating the quality of machine-translated text. Introduced by Kishore Papineni and colleagues in 2002 [5], BLEU has become a cornerstone in the field of Natural Language Processing (NLP), particularly for tasks involving language generation and translation. It is an automatic evaluation metric that compares machine-generated text to one or more reference texts usually created by humans. It measures the degree of overlap between the candidate translation and the reference translations using a precision-based approach. The core idea is that the more n-grams (contiguous sequences of n words) in the candidate translation that match the reference translations, the higher the quality of the translation. It is calculated using the following formula:

$$\text{BLEU} = \text{BP} \times \exp\left(\sum_{n=1}^{N} w_n \log p_n\right)$$

where:

- BP is the brevity penalty;
- p_n is the precision for n-grams of size n;
- w_n is the weight for the n-gram level;
- N is the maximum n-gram level.

The **Brevity Penalty** (BP) is calculated as:

$$\text{BP} = \begin{cases} 1 & \text{if } c > r \\ e^{(1-\frac{r}{c})} & \text{if } c \leq r \end{cases}$$

where:

- c is the length of the candidate translation;
- r is the effective reference length (usually the closest reference length to c).

This factor is designed to penalize translations that are shorter than the reference translations, as shorter translations may miss important content. It ensures that translations which are too concise are penalized, with an exponential decay based on the ratio of r to c, preventing them from artificially inflating the BLEU score.

Precision for n-grams (p_n) measures the proportion of n-grams (contiguous sequences of n words) in the candidate translation that match n-grams in the reference translations.

For each n-gram size n (usually goes from 1 up to 4):

$$p_n = \frac{\text{Number of matching } n\text{-grams}}{\text{Total number of } n\text{-grams in candidate}}$$

- precision for 1-grams (p_1) counts the number of matching individual words;
- precision for 2-grams (p_2) counts matching pairs of words, and so on up to N-grams;
- higher precision values indicate that a greater proportion of the n-grams in the candidate translation are present in the reference translations.

Weight for the n-gram level (w_n) are used to average the precision scores for different n-gram levels. Typically, the weights are uniform, meaning:

$$w_n = \frac{1}{N}$$

where N is the maximum N-gram level considered (usually 4). This ensures that each n-gram level contributes equally to the final score.

In the end logarithm and exponentiation are used to compute the geometric mean of the precisions. This method helps to average the precision across different n-gram sizes in a way that penalizes the presence of higher n-grams in the candidate translation if they are not present in the references.

3.4.2 CodeBLEU

CodeBLEU is an evaluation metric specifically designed to assess the quality of code generated by large language models [6]. While traditional BLEU is widely used for evaluating natural language translations, it does not fully capture the shades and requirements of programming languages. CodeBLEU extends the BLEU metric by incorporating additional components tailored for code, such as syntax, data flow, and logic consistency, to provide a more comprehensive evaluation of code quality.

The CodeBLEU metric maintains the core principle of n-gram matching from BLEU but adds language-specific adaptations. These adaptations include assessing the structural similarity between generated and reference code, ensuring syntactical correctness, and analyzing data flow to verify that the variable usages in the generated code align with those in the reference code. Additionally, CodeBLEU evaluates the logical correctness of the generated code by considering code-specific patterns and practices.

It's computed as follow:

$$\begin{aligned}\text{CodeBLEU} = \alpha \cdot \text{BLEU} + \beta \cdot \text{Weighted n-gram match} + \\ \gamma \cdot \text{Syntax match} + \delta \cdot \text{Data flow match}\end{aligned} \tag{3.11}$$

where:

- BLEU is the traditional BLEU score;
- Weighted n-gram match enhances the n-gram matching by giving different weights to different types of tokens;
- Syntax match evaluates the structural similarity of the generated code compared to the reference code;
- Data flow match checks the consistency of data usage, ensuring that variables are used in similar ways in both the generated and reference code;

- $\alpha, \beta, \gamma, \delta$ are weights assigned to each component to balance their contributions, typically chosen based on empirical validation.

CodeBLEU's multi-faceted approach addresses the limitations of BLEU when applied to programming languages. By evaluating not just the surface-level text similarity but also the deeper structural and logical aspects of code, it offers a more robust and reliable measure of the functional accuracy and quality of machine-generated code. This makes it an essential tool for researchers and developers working on code generation models, ensuring that the generated code is not only syntactically similar but also functionally correct and logically coherent.

3.4.3 Computational Accuracy

Ensuring computational accuracy is crucial when using large language models for code generation. LLMs have demonstrated significant potential in assisting with code development, offering solutions ranging from simple functions to complex algorithms. However, the inherent nature of these models means that while they can produce syntactically correct and seemingly logical code, the accuracy and correctness of this code must be rigorously validated to meet the required standards. Computational accuracy refers to the degree to which the output of a computational process conforms to the expected results. In the context of code generation, this means that the code must not only compile and run without errors but also perform the intended operations correctly and efficiently. To check it's correctness unit test are used. Unit testing involves testing individual units or components of a software program to determine whether they function correctly. Each unit test verifies a specific aspect of the code, ensuring that the output matches the expected results for a given set of inputs. So the pass rate of unit tests can serve as a valuable metric to evaluate the effectiveness and reliability of an LLM in code generation.

3.4.4 Model Efficiency Metrics

Number of Parameters

The number of parameters in a large language model (LLM) serves as a fundamental metric for assessing various versions of the same model. Parameters, which include the weights and biases, are learned during the training process and define the model's ability to learn from data and generalize to new inputs. Typically, a higher parameter count allows the model to capture more intricate patterns, potentially enhancing performance. However, this also increases the model's size and the computational resources required for both training and inference.

Evaluating different iterations of the same LLM based on the parameter count is crucial for understanding the trade-offs between model complexity and resource efficiency. For instance, a model with fewer parameters will occupy less memory when loaded, making it advantageous for deployment on devices with limited GPU

memory.

In summary, the number of parameters is a pivotal metric influencing the overall efficiency and effectiveness of a large language model. It directly impacts memory consumption, computational cost, and the feasibility of deploying the model across various environments. Comparing different versions of the same model based on this metric provides valuable insights into achieving optimal performance with efficient resource utilization.

FLOPs - Floating-Point Operations

FLOPs, or Floating-Point Operations, is a key metric for gauging the computational complexity and efficiency of various versions of the same large language model (LLM). FLOPs denote the number of arithmetic operations the model executes to process a single input, offering insights into the model's computational demands and performance characteristics.

A model with a higher FLOP count typically necessitates more computational power and time to process inputs, which can lead to increased energy consumption and longer inference times. In contrast, a model with fewer FLOPs generally operates more efficiently, requiring less computational effort and enabling quicker processing.

Assessing different versions of the same LLM based on this metric helps identify models that strike a balance between performance and computational efficiency. By comparing the operations required by various models, researchers can pinpoint which versions deliver optimal performance with minimal computational cost. This comparison is vital for optimizing the deployment of LLMs in resource-constrained environments or where energy efficiency is paramount.

GPU Utilization Requirement

GPU utilization is a significant metric for evaluating the memory resources necessary to load different versions of the same large language model (LLM). This metric centers on the memory capacity needed to load the model into the memory, rather than the computational resources required for inference.

The memory required to load a model is directly correlated with the number of parameters it contains. A model with a higher parameter count will typically need more GPU memory to be fully loaded, affecting the feasibility of deploying the model on devices with limited capacity, such as edge devices or consumer-grade GPUs. Conversely, models with fewer parameters will occupy less memory, making them more suitable for deployment in memory-constrained environments.

These metrics are closely interrelated. A lower number of parameters not only reduces the memory needed to load the model but also decreases the number of FLOPs,

leading to less computational effort during operations. Furthermore, this reduction minimizes the GPU resources required to effectively run the model, enhancing overall efficiency.

Chapter 4

Methods and Results

4.1 Model

For this project, the OpenCodeInterpreter-DS-6.7B model has been selected[1] [10]. This model is part of a broader series that exemplifies the progressive advancements in coding model performance, particularly through the integration of execution feedback mechanisms. It is specifically trained on code, which significantly enhances its ability to understand and generate accurate programming solutions. Despite its relatively modest size—comprising 6.7 billion parameters—the model demonstrates exceptional performance on the EvalPlus benchmark tests [4], standing out even when compared to competitors that have over 33 billion parameters. The OpenCodeInterpreter-DS-6.7B is built upon the LLaMA architecture [8], a foundation renowned for its efficiency and robustness in natural language processing tasks. The model architecture consists of 32 decoder layers, a design that contributes to its capability to process and generate code effectively. This structure is illustrated in Figure 4.1, which provides a detailed overview of its components. The selection of this model underscores the importance of both parameter efficiency and innovative strategies, such as execution feedback, in modern AI models. Its specialized training on code allows it to offer a compelling balance of performance and efficiency, making it particularly well-suited for the specific requirements of this project.

[1] https://huggingface.co/m-a-p/OpenCodeInterpreter-DS-6.7B

```
LlamaForCausalLM(
  (model): LlamaModel(
    (embed_tokens): Embedding(32256, 4096)
    (layers): ModuleList(
      (0-31): 32 x LlamaDecoderLayer(
        (self_attn): LlamaSdpaAttention(
          (q_proj): Linear(in_features=4096, out_features=4096, bias=False)
          (k_proj): Linear(in_features=4096, out_features=4096, bias=False)
          (v_proj): Linear(in_features=4096, out_features=4096, bias=False)
          (o_proj): Linear(in_features=4096, out_features=4096, bias=False)
          (rotary_emb): LlamaLinearScalingRotaryEmbedding()
        )
        (mlp): LlamaMLP(
          (gate_proj): Linear(in_features=4096, out_features=11008, bias=False)
          (up_proj): Linear(in_features=4096, out_features=11008, bias=False)
          (down_proj): Linear(in_features=11008, out_features=4096, bias=False)
          (act_fn): SiLU()
        )
        (input_layernorm): LlamaRMSNorm()
        (post_attention_layernorm): LlamaRMSNorm()
      )
    )
    (norm): LlamaRMSNorm()
  )
  (lm_head): Linear(in_features=4096, out_features=32256, bias=False)
)
```

Figure 4.1. Model Structure

4.2 Datasets

Training Dataset

To retrain the final layers of the model, the same dataset used for its initial comprehensive training, Code-Feedback, was employed[2]. This dataset consists of 66,400 rows, each containing multi-turn interactions that span various programming languages.

```
[{ "role": "user", "content": instruction...},
 { "role": "assistant", "content": answer...},
 { "role": "user", "content": instruction...},
 ... ]
```

Figure 4.2. Sample Structure

These rows are composed of diverse programming-related questions and tasks,

[2] https://huggingface.co/datasets/m-a-p/Code-Feedback

structured in a way that aligns with the example shown in Figure 4.2. Code-Feedback supports OpenCodeInterpreter by integrating execution and human feedback into its training process, enabling dynamic code refinement. This dataset's inclusion of feedback mechanisms plays a crucial role in enhancing the model's ability to generate accurate and efficient code.

Benchmark

The evaluation of the models has been carried out using the G-TransEval[3] dataset, which is detailed in the research paper [2]. This dataset, available at the repository G-TransEval, comprises 400 code samples that each demonstrate code translation across five different programming languages: Python, C++, Java, C#, and JavaScript. Each sample is accompanied by corresponding unit tests designed to validate the accuracy and correctness of the translations.

In this project, the focus is specifically on translating code snippets from Java to Python. The unit tests provided with the G-TransEval dataset are utilized to rigorously assess the correctness and reliability of the translations. These tests are instrumental in verifying that the translated Python code behaves as expected and maintains the functionality defined by the original Java code.

The primary objective of this evaluation is to determine how different versions of the model perform in translating Java code to Python, using unit tests to measure accuracy and robustness. This focused approach facilitates a comprehensive examination of the model's translation capabilities, offering insights into its strengths and identifying potential areas for improvement. The findings from this evaluation contribute to a deeper understanding of the model's effectiveness after its layer's update.

4.3 NOSE

In Algorithm 1, the NOSE method was applied to identify the layers at each step where the transfer entropy with the final layer was minimal. Figure 4.3 illustrates the progression of these steps, revealing a general upward trend in transfer entropy values. This trend indicates that the later layers exert a stronger influence on the entropy of the output. During each step, the layer with the lowest transfer entropy value is selected, and after iterating this process across 10 steps, the final results are presented in Figure 4.4.

[3] https://github.com/PolyEval/G-TransEval

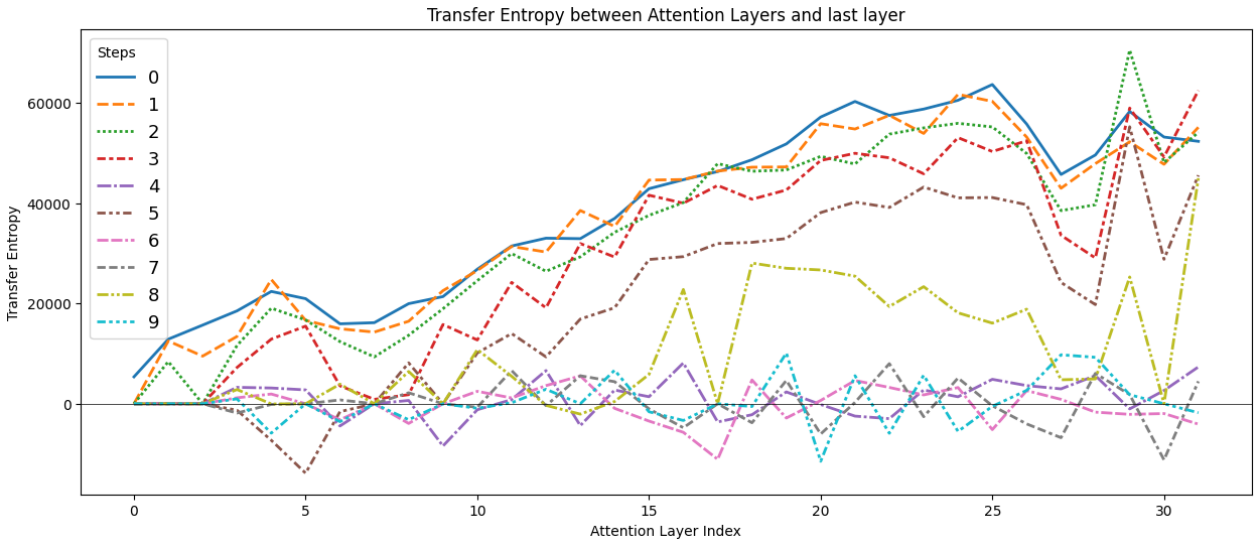

Figure 4.3. Transfer Entropy during NOSE's steps

A notable observation can be drawn from these plots: while the transfer entropy values across the first few steps display a similar pattern, a significant shift occurs from step 6 onwards. Beyond this point, the values fluctuate around zero, making the selection of the optimal layer index increasingly evident. After this point, the layer's index selected is not included in the first layers as the beginning ones but the algorithm selects indexes in the middle and last part of the model structure. The shift may be attributed to the fact that the layer extracted in a previous step has a substantial impact on the remaining layers, reducing the information available from the other layers as they can no longer rely on the critical data that has been removed. This phenomenon underscores the intricate dependencies between layers and highlights the challenges in extracting their contributions to the overall entropy.

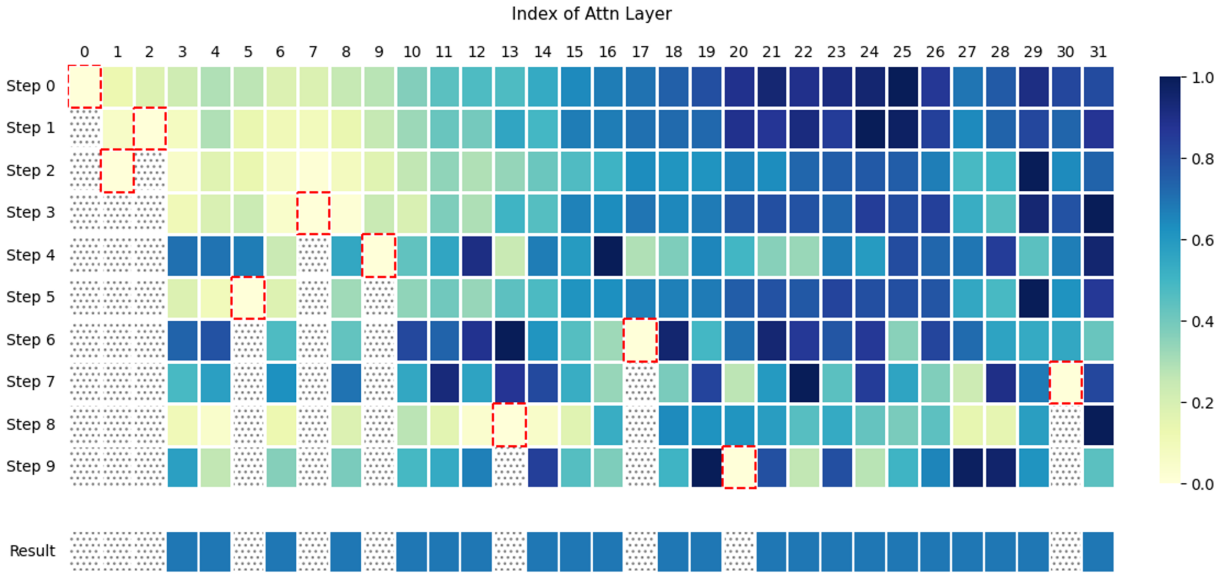

Figure 4.4. Visualization of NOSE

4.4 Model Shrinking

In the process of fine-tuning the decoder layers using the NOSE algorithm, we modify the outputs of the self-attention layers according to the expression provided in Equation 3.10. Furthermore, during each epoch of the fine-tuning process, we dynamically update the elements within the matrix M. This matrix M is structured based on a specific criterion:

$$M = M_0 \cdot \left(1 - \frac{\text{current epoch number}}{\text{total epochs number}}\right) \tag{4.1}$$

Here, M_0 represents the initial matrix, which is populated with all elements set to 1 and has dimensions equivalent to the output tensor generated by the self-attention mechanism. By adhering to this formulation, the impact of the self-attention mechanism is progressively diminished over the course of the fine-tuning epochs. Eventually, this reduction ends up in the complete elimination of the self-attention effect, which is then supplanted by twice the input values. This strategic adjustment ensures a smooth transition and integration of the new processing approach, enhancing the model's adaptability and performance during the fine-tuning phase.

4.5 Training

Before tokenizing and passing the samples to the model for the training a chat template is applied: in this way, the information about the user prompt and the assistant reply are made readable for the model using special tokens.

A few tests have been run to find the best choice of hyperparameters to select during the training phase and the following results have been obtained:

- **learning rate**: a suitable value for this hyperparameter is found to be $3.5 \cdot 10^{-7}$ since a higher value brings the training to stops for NaN during training, especially for multiple attention layer reduction;
- **batch size**: batch size equal to 1 has been used since the memory required to allocate more samples exceeded the available one;
- **gradient accumulation**: since the batch size couldn't be increased, we try using an accessory technique that allows making the gradient flow backwards not at each step. Using this technique the loss at each step was averaged among the number of accumulation steps selected and the optimization step ruuned after each gradient accumulation step. Even using this tool the model couldn't train properly and the time required did not allow the model to adjust the weights enough so even this value was settled to 1;
- **gradient clipping**: during some steps, the gradient reaches too high values that make the model's weights update too big to generate numerical instability during this phase. To solve this problem the clipping gradient has been used, up-bounding the value to 1.

The different versions of the model were trained for 10 epochs using A100 single GPU that required a total training time of around 70 hours for each training.

4.6 Results

The first fine-tuning experiment focused on modifying the attention mechanism in the first layer selected using the NOSE algorithm. The results of this experiment are presented in Table 4.1. While the outcomes are promising, they deviate somewhat from our initial expectations.

The table shows a slight reduction in the model's parameters (-1.04%) and computational requirements (-1.01%) was achieved through this procedure. Despite these reductions, performance metrics such as BLEU and CodeBLEU (CB) remain relatively stable, with minimal change. BLEU remains constant, while CodeBLEU shows a minor decrease of -1.30%. However, the most significant observation is the decline in Computation Accuracy (CA), which dropped by -7.53%. This suggests that while parameter and memory optimizations have been effective, the drop in

Metric	Base	Nose Step 0
Flops (B)	6766	6698 (-1.01%)
Parameters (B)	6.74	6.67 (-1.04%)
Memory (GB)	12.56	12.43 (-1.04%)
BLUE	0.70	0.70 (+0.00%)
CB	0.77	0.76 (-1.30%)
CA	0.93	0.86 (-7.53%)

Table 4.1. Comparison of Model Performance with the Base Model

CA highlights a trade-off that may impact overall model quality. This reduction indicates a potential area for further improvement, as maintaining high accuracy remains a priority for this task. We did not anticipate this behaviour, as it contrasts with our expectations based on the validation loss observed during training. As shown in Figure 4.5, the validation loss remained nearly constant when fine-tuning the model with modifications to only the first self-attention (SA) mechanism selected. Typically, this would indicate that the model's performance is relatively stable, with no significant decline during the fine-tuning process. Given the stable loss curve, we initially assumed that performance metrics, such as BLEU, CodeBLEU, and Computation Accuracy (CA), would remain unaffected or only exhibit minimal degradation.

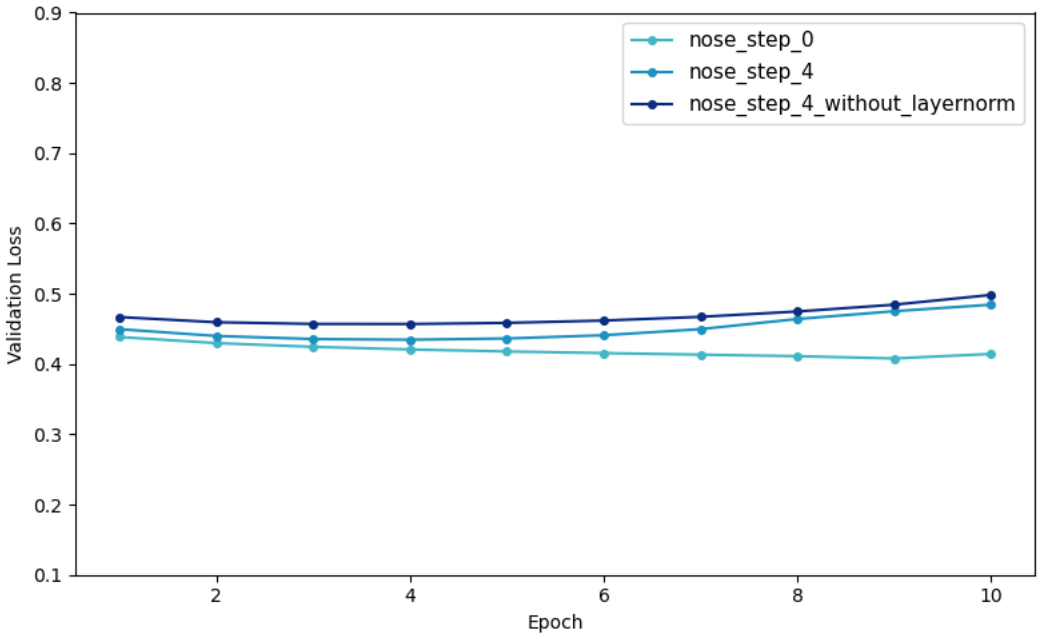

Figure 4.5. Fine-tuning Validation Losses

However, the results revealed a different trend. This unexpected outcome prompted further investigation into the behaviour of the model. To explore this anomaly in greater depth, we extended the fine-tuning process by making additional adjustments. Specifically, we expanded the scope of the NOSE algorithm's application beyond the first SA layer to include the first five SA layers, reducing them to identity mappings. This approach aimed to determine whether more extensive modifications to the attention mechanism could yield different results or shed light on why the initial fine-tuning led to such a marked decline in CA.

In the first approach, we retrained both the self-attention mechanism and the post-attention layer normalization for these layers. In the second approach, we kept the layer normalization frozen while retraining only the self-attention mechanisms. The reason behind conducting these two cases stems from the critical role that layer normalization plays in maintaining stability during training. By allowing the post-attention layer normalization to be retrained in the first experiment, we aimed to give the model maximum flexibility to adapt to the changes introduced by the removal of the initial SA layers. This retraining helps the model recalibrate how it normalizes the outputs from the attention mechanism, which can be crucial when adjusting to new patterns or fine-tuning a task that may differ from pretraining data. On the other hand, we also wanted to assess the model's behaviour when the normalization layers were left unchanged, as they were during pretraining. In this second case, keeping the layer normalization frozen allowed us to isolate the impact of fine-tuning the self-attention mechanism alone. This approach would help us determine whether the drop in performance was tied more directly to the self-attention mechanism or if the layer normalization itself contributed to the problem. By controlling these variables separately, we could better understand the interactions between the self-attention layers and the normalization process, and how they affect the model's overall performance.

Unfortunately, in both cases, we encountered a critical issue where the model ultimately produced a tensor of *NaN* values during inference after the attention mechanisms were replaced with an identity mapping following the fine-tuning process. This outcome was observed regardless of whether the post-attention layer normalization was retrained or kept frozen. The emergence of *NaN* values suggests that the model's internal stability was compromised, likely due to the drastic modification of the self-attention layers. By reducing these layers to identity mappings, we effectively removed the ability of the attention mechanism to dynamically capture dependencies within the input data, which may have led to improper gradient propagation or numerical instability during inference. As a result, the model's ability to produce meaningful outputs was severely impaired, ending in this failure mode across both experimental conditions.

Moreover, in both cases, the validation loss exhibited a noticeable upward trend during training, rather than remaining stable (Figure 4.5). This increase in loss was

an early indication of potential issues, suggesting that the model was struggling to adapt to the modifications applied to the self-attention layers. The rising validation loss hinted at a degradation in the model's generalization ability, signalling instability in the learning process. This behaviour could potentially be attributed to the choice of hyperparameters or issues during the training phase. However, the behaviour observed remains unexplained, even considering these factors, until this point.

In an effort to understand the underlying cause of the model's erroneous behaviour after fine-tuning, we explored various strategies aimed at mitigating the instability. Specifically, we focused on adjusting the scaling of the matrix M by applying a multiplicative factor to the input tensor across different layers. We experimented with values for this factor within the range of 0.1 to 1, while also reducing the number of self-attention layers that were retrained. The adjustments were guided by the insights derived from Equation 3.10, which helped inform our approach. We also worked on the input tensor multiplier trying to reduce this factor from 2 up to 1.

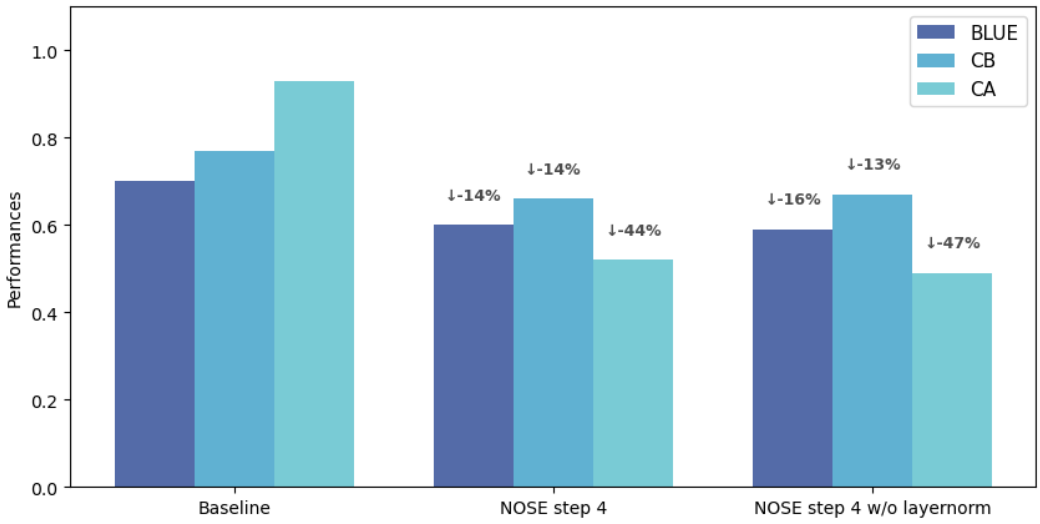

Figure 4.6. Inference after fine-tuning of 5 layers with and without layernorm with custom configuration

Through a series of trials with different combinations of layer modifications and scaling factors, we discovered that the model's performance began to stabilize and improve under specific conditions. Notably, the model achieved good performance when we applied identity mappings to the final retrained layers selected by the NOSE algorithm and used a scaling factor of 0.1 for the matrix M in the other layers. More specifically, using the fine-tuned model with the SA of the layers with indexes 0, 2, 1, 7 and 9 (see Figure 4.4) we substitute the SA mechanism of layer 9 with the identical mapping and multiplied the matrix M of all the others for 0.1 and using a factor additional contribute of the input tensor of 1.7. This combination allowed the model to retain the benefits of the self-attention mechanism in key areas while introducing controlled changes in other parts of the architecture to maintain

stability. Despite that, the performances were still lower than expected (Figure 4.6).

These results suggest that the initial layers of the model may not be the most optimal choice for fine-tuning, challenging the approach of starting with the earlier layers in the architecture. Following this observation, we decided to adjust the strategy to focus on the latter layers of the model, as their modifications seemed to have a more significant impact on maintaining performance. To test this hypothesis, we conducted an inference on the testing dataset, this time replacing the self-attention mechanisms of the layers selected by the NOSE algorithm, but in reverse order, starting from the final layers of the model (layer 19 as shown in Figure 4.4).

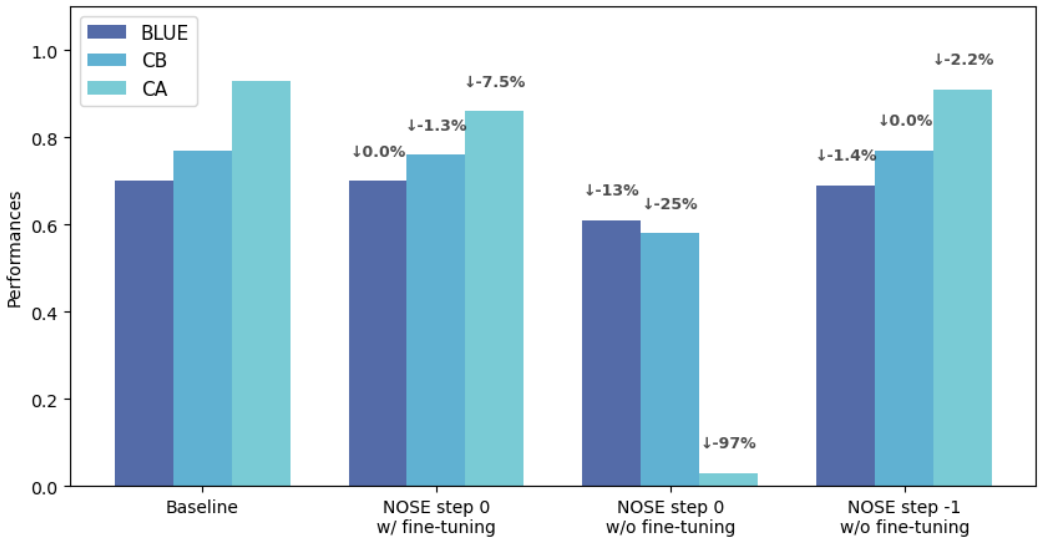

Figure 4.7. Inference updating a single layer

Surprisingly, even without retraining these selected layers, the model's performance remained high, in stark contrast to the previous experiments (Fig 4.7). In the plot, we can observe that the performance of the model, when replacing only the last layer selected by the NOSE algorithm without any fine-tuning, surpasses the performance of the fine-tuned model where only the first selected layer was updated. This result is particularly striking, as it demonstrates that even without retraining, the modification of the later layers can lead to better outcomes compared to fine-tuning earlier layers. This suggests that the layers closer to the output, such as the final self-attention layers, play a more critical role in influencing the model's task-specific performance. In contrast, fine-tuning earlier layers seems to introduce instability or degrade performance, potentially because those layers are responsible for capturing more general or low-level features. In fact, when comparing the models without fine-tuning, it becomes evident that updating the earlier self-attention mechanisms results in a significant loss of information. These early layers are responsible for capturing foundational and altering them disrupts the model's ability to process crucial input information effectively. On the other hand, the

self-attention mechanisms in the later layers appear to be less critical for capturing important features. As a result, modifying or even replacing these layers with identity mappings has a much smaller impact on performance, suggesting that the later layers primarily refine task-specific outputs rather than extract essential input features. This distinction further highlights the importance of carefully selecting which layers to modify during fine-tuning.

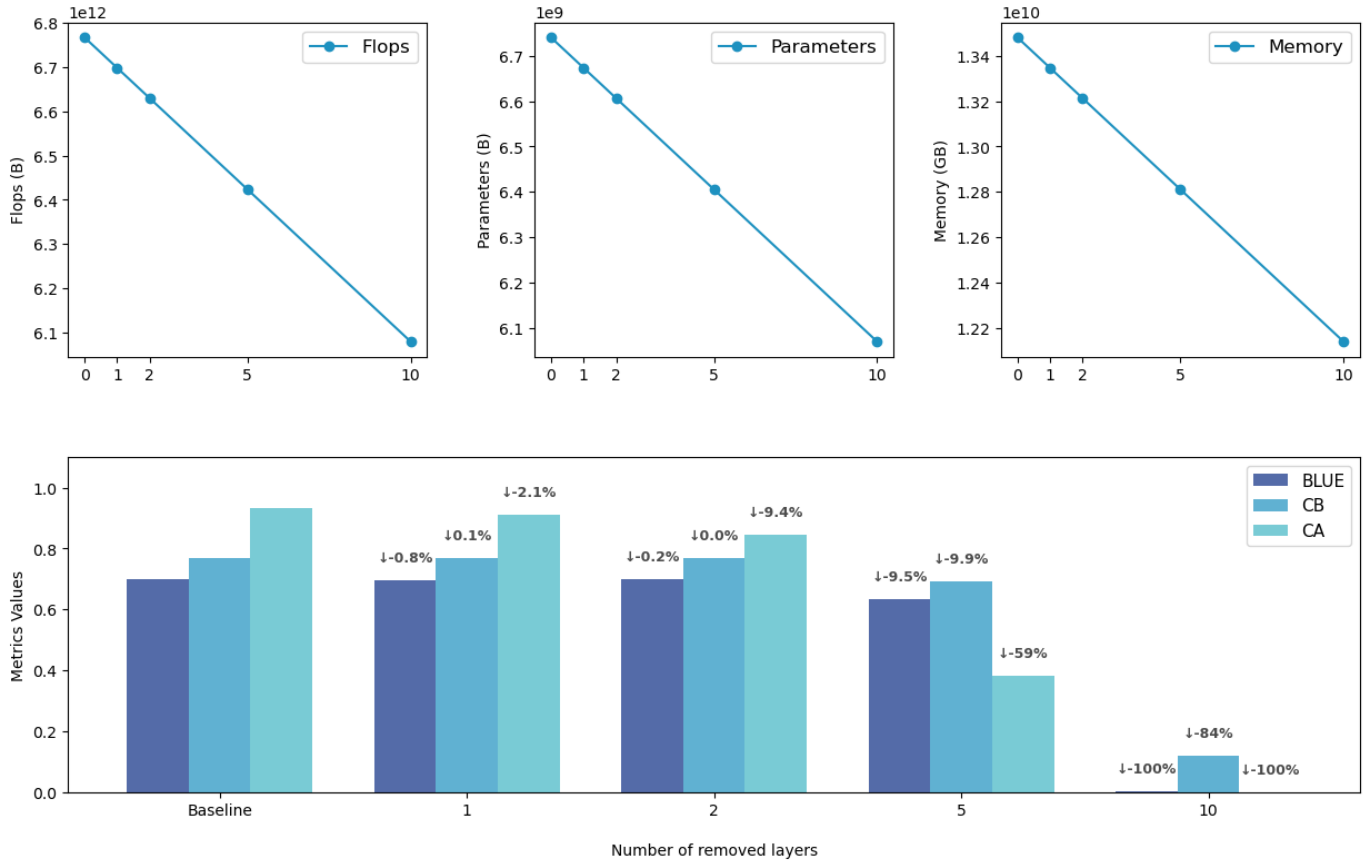

Figure 4.8. No fine tuning removal comparison (reverse NOSE)

To further investigate this discovery, we conducted an inference on the test set by replacing the selected self-attention mechanisms with identity mappings, without retraining, using varying numbers of layers. Specifically, we tested with 1, 2, 5, and 10 layers, starting from the last layer identified by the NOSE algorithm and going up. The results of this experiment are presented in the plot shown in Figure 4.8. We can see how the model efficiency metrics (flops, parameters and memory) decrease linearly with the number of updated layers and how the performances of the model slowly decrease since without retraining the model we are just changing its weights.

To gain a deeper understanding of the behavior of the models we can make the following considerations:

- **Early Layers and Output Distribution:** The early layers of a transformer model, particularly those in the initial stages of the network, do not significantly influence the final output distribution. This is because these layers primarily focus on capturing low-level features and performing initial transformations on the input data. As a result, the impact of these early layers on the overall model output is relatively minimal and this is probably the reason why they are extracted in the earliest stages of the NOSE algorithm.
- **Last Layers and Probability Distribution:** In contrast, the last layers of the transformer model play a crucial role in shaping the final probability distribution of the predictions. These layers are responsible for refining the high-level features and making the final decisions based on the information gathered throughout the network. Despite their importance, these layers may not fully exploit the potential of the attention mechanism. This is evidenced by the observation that replacing a few of these layers with an identical mapping does not drastically reduce the model's, differently from the first ones. This suggests that while the last layers are essential for the final output, they may not be utilizing the full power of the attention mechanism to its fullest extent.
- **Importance of Early SA Layers:** The initial self-attention layers, particularly those at the beginning of the network, are highly relevant for the compression and transformation of low-level features. These layers are crucial for encoding the input data into a format that can be effectively processed by subsequent layers. Given their importance, it is not feasible to replace these layers with their corresponding MLPs, as the MLPs alone cannot capture the same level of information as the attention mechanism. The attention mechanism allows for a better and context-aware representation of the input data, which is essential for the overall performance of the model.
- **Interplay Between Attention and MLPs:** The interplay between the attention mechanism and the MLPs throughout the network is a key factor in the model's performance. While the attention layers are adept at capturing complex relationships and dependencies in the data, the MLPs are essential for performing non-linear transformations and refining the features extracted by the attention layers. This collaborative effort ensures that the model can effectively learn and generalize from the input data, leading to robust and accurate predictions. But another consideration has to be done. From our experiments, we observed that the position of the decoder layer, where these components are integrated, significantly influences their importance. Specifically, the Self-Attention (SA) mechanism appears to be crucial for the model's performance in the earlier stages. However, as the layers progress

towards the end, the relevance of SA diminishes. This insight allowed us to strategically remove the SA mechanism from the final layers during model adaptation. By doing so, we were able to enhance the model's efficiency and reduce its computational consumption.

- **Optimization and Efficiency:** Understanding the role of each layer in the transformer model is not only crucial for interpreting the model's behavior but also for optimizing its efficiency. By identifying the layers that contribute most significantly to the model's output, we can focus on optimizing these layers to improve the model's performance without compromising its efficiency. This could involve fine-tuning the attention mechanism, adjusting the architecture of the MLPs, or even pruning redundant layers to reduce computational overhead.

Based on the insights we have gathered from this project, a more optimal approach to ensure its success could be structured as follows: Initially, we executed the NOSE algorithm for only 10 steps. However, given the discoveries made throughout our analysis, it may be beneficial to extend the application of the algorithm to all layers of the model. Following this, a more refined strategy would involve selecting the layers in reverse order, starting from the last layers, and progressively fine-tuning the model by incrementally increasing the number of decoder layers. This approach would allow us to assess whether the model can maintain its performance levels despite the reduction in its size. As illustrated in Figure 4.4, we observe that, during the initial steps of the NOSE algorithm, the earlier layers of the model are predominantly selected. This suggests that these early layers have less influence on the final output distribution, as detected by the NOSE algorithm. The selection of these layers in the earlier steps indicates that their contribution to the overall feature extraction process is minimal. In contrast, the layers selected during the latter steps of the algorithm are likely to reside in the second half of the model, where the contribution to the output becomes more significant. Interestingly, the attention mechanism applied to these latter layers seems to have less impact on the overall feature extraction. Instead, the most critical feature extraction work is performed by the subsequent MLP layers. The operations in these MLP layers, particularly the matrix multiplications, appear to be key to achieving the expected outcomes. By fine-tuning the model in this manner, we can observe whether the model can continue to deliver robust results even as its structure becomes more streamlined. This approach would not only enhance our understanding of the importance of specific layers but also offer insights into how the model's architecture can be optimized for both performance and efficiency.

Chapter 5

Conclusions and Future Works

This thesis has provided a comprehensive exploration of the OpenCodeInterpreter-DS-6.7B model, focusing on its fine-tuning and optimization for code generation tasks. Through a series of experiments and analyses, we have gained valuable insights into the model's architecture, the impact of different layers on its performance, and the effectiveness of various fine-tuning strategies. The aim of this project was to find a way to reduce the size and computation time during inference with a large language model (LLM), following the guidelines from the work presented in the paper by Lin et al. (2024) [3]. While the goal has not been fully achieved, a promising path for achieving this has emerged, and the results indicate that this procedure could potentially lead to significant outcomes in the future.

The NOSE algorithm was instrumental in identifying layers with minimal transfer entropy, which suggested they had a lesser impact on the final output. However, our experiments revealed that the early layers, despite having minimal transfer entropy, play a crucial role in capturing low-level features and transforming input data. When these layers were replaced with identity mappings, it led to significant performance degradation, highlighting their importance in the model's architecture. Thus, the importance of both the self-attention mechanism and the multi-layer perceptron (MLP) components within each decoder layer requires further investigation. The influence of these components at different stages of the model's processing could uncover more refined strategies for layer selection and tuning, which may ultimately lead to the desired reduction in model size and computation time.

In contrast to initial expectations, the later layers of the model, particularly those closer to the output, exhibited greater stability and less impact on overall performance when modified. For example, replacing the self-attention mechanisms of these layers with identity mappings—without retraining—resulted in minimal performance loss. This suggests that these later layers serve primarily to refine task-specific outputs, rather than performing critical feature extraction. This finding opens up a potential avenue for reducing the complexity of the model while maintaining its effectiveness.

Performance metrics also revealed interesting trade-offs. While the fine-tuning process resulted in a slight reduction in model parameters and computational requirements, with minimal changes to performance metrics such as BLEU and CodeBLEU scores, there was a notable decrease in Computation Accuracy (CA). This indicates a trade-off between efficiency and accuracy, which must be carefully considered when modifying critical layers. The slight reduction in BLEU scores suggests that, while the modifications did not drastically affect the model's ability to generate syntactically correct code, the accuracy of computation-based tasks suffered slightly, highlighting the need for careful fine-tuning to preserve performance.

Furthermore, conducting inference on the test set with varying numbers of replaced layers without retraining showed that starting from the last layers identified by the NOSE algorithm led to better performance. This approach revealed that the later layers, while important for refining outputs, are less critical for capturing essential features. Thus, more aggressive modifications to these layers can be made without significantly compromising the model's performance.

This thesis has laid a solid foundation for future work in optimizing LLMs for code generation tasks. Several key directions for further research and development arise from these findings. For instance, more sophisticated fine-tuning strategies could focus on specific layers based on their performance impact, potentially incorporating adaptive learning rates and layer-specific regularization techniques. This could help maintain stability during the fine-tuning process and lead to further model optimization. Additionally, further investigation into the role of self-attention and MLP components in each decoder layer could reveal new techniques for selecting and tuning these layers, ultimately leading to more efficient models.

Moreover, exploring hybrid models that combine elements from architectures with larger parameter sizes—while maintaining the efficiency of smaller models—could offer a new avenue for achieving both size reduction and high performance. Dataset augmentation, with more diverse and complex code samples, could also help the model generalize better to a wider range of programming tasks. This would enable the model to continue generating accurate and efficient code while further reducing its computational footprint.

Advanced optimization techniques such as pruning and quantization could further reduce computational overhead without sacrificing performance. Continuous learning approaches could also be implemented, allowing the model to adapt to new data over time, ensuring that it remains relevant as programming languages and paradigms evolve. Additionally, enhancing human-AI collaboration in code generation tasks could provide a more seamless integration of human feedback into the model's training process, potentially improving both the efficiency and effectiveness of generated code.

Finally, as AI-driven models for code generation continue to evolve, developing

scalable architectures that can handle larger datasets and more complex tasks will be crucial for deploying these models in real-world applications. While the ultimate goal of this project—reducing the size and computation time for inference—has not been fully realized, the promising path opened by this research holds significant potential for future advancements. By carefully selecting layers for modification and employing strategic fine-tuning techniques, we have demonstrated the potential for optimizing model efficiency without compromising accuracy. These findings lay the groundwork for further research and development in the field of AI-driven code generation and optimization.

Appendix A

Source Code

The source code used in this project is available on GitHub at the following link: https://github.com/Matteo-Candi/Master-Thesis.

The repository contains the complete code, data, and instructions to replicate the experiments described in this thesis.

Bibliography

[1] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale, 2021.

[2] Mingsheng Jiao, Tingrui Yu, Xuan Li, Guanjie Qiu, Xiaodong Gu, and Beijun Shen. On the evaluation of neural code translation: Taxonomy and benchmark, 2023.

[3] Sihao Lin, Pumeng Lyu, Dongrui Liu, Tao Tang, Xiaodan Liang, Andy Song, and Xiaojun Chang. Mlp can be a good transformer learner, 2024.

[4] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and LINGMING ZHANG. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[5] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In Pierre Isabelle, Eugene Charniak, and Dekang Lin, editors, *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA, July 2002. Association for Computational Linguistics.

[6] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis, 2020.

[7] Zhenhong Sun, Ce Ge, Junyan Wang, Ming Lin, Hesen Chen, Hao Li, and Xiuyu Sun. Entropy-driven mixed-precision quantization for deep network design. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 21508–21520. Curran Associates, Inc., 2022.

[8] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro,

Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023.

[9] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

[10] Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. Opencodeinterpreter: Integrating code generation with execution and refinement, 2024.