Krishna Ayyalasomayajula

Ms. Horton

IB Extended Essay

February 28 2025

Rule-based Tensor Mutations Embedded within LLMs for Low-Cost Mathematical Computation

Introduction

Artificial Intelligence (AI) has surged in popularity and capability in recent years, leading to immense developments and investments into the technology. AI, since its modern inception, has been based on a Multi-Layer Perceptron (MLP), with the underlying structure being a Neural Network (NN) (Testolin 1). An MLP is a type of artificial neural network (ANN) that consists of a layering of neurons computed through the vector multiplication of weights and biases, resulting in propagation to a set of values that represents the states of the next layer. This structure, called a neural network is a computational model inspired by the human brain, borrowing from its chaining of neurons to create complex understanding from basic electrical impulses. The below expression details the computational process involved in evaluating progressive layers of neurons in a neural network.

$$\vec{y} = f\Big(\Big[\sum_{j=1}^{n} w_{1j}x_j + b_1,\ \sum_{j=1}^{n} w_{2j}x_j + b_2,\ \ldots,\ \sum_{j=1}^{n} w_{mj}x_j + b_m\Big]^T\Big)$$

The vector $\vec{y}$ is the output vector for any given preceding layer, resulting in an m dimensional evaluation. Each connection between layers of an MLP has two values associated: a weight and a bias. Like the resistance across a synapse in the human brain, biases modulate the activation of each neuron. Mathematically, the biases for each connection are valued for each proceeding neuron, resulting in a $m \times 1$ shape. Weights are defined for each neuron, allowing the creation of a matrix $\mathbf{W}$, which is then computed against the activated values of the preceding layer in a matrix multiplication, then summed with the corresponding bias to generate a scalar for each neuron of the next layer. This process is called forward propagation, as compared to backpropagation. This entire propagation is composited inside a vector-valued activation function $\vec{f}$, which is essential in setting the bounds of the system. However, this can be problematic in complex situations.

The end goal of any machine learning experiment is to function like a regression, except with many more parameters. Backpropogation is the process of updating these weights and biases using discrete evaluation to minimize a cost function L. Truly, MLPs are the largest optimization problem ever created. In other words, MLPs can be trained to create certain output for given input using the forward propagation to evaluation, a loss function to find error, and a backpropagation optimizer to update its variables.

In recent years, a specialized kind of Machine Learning models have hit the market — Large Language Models (LLMs). An LLM is a generative MLP that creates human-readable text given a prompt. They stem from early attempts in the 2000s to use neural networks with Recurrent Neural Networks to analyze sequences of words for sentiment, keywords, and grammar (Wang et al. 2). With the rise of tokenizers, or models that convert words to vector embeddings that help encode the meaning thereof, the 2017 paper *Attention is All You Need* changed the landscape of AI forever with the introduction of the self-attention transformer. This development allowed models to understand the relationships between words with far less training. While any task is possible — theoretically — with neural networks, optimizations such as these allow for lower error with far less training, making the while process more sustainable. The probability of reaching a minimum of the loss function is far greater with such improvements to MLP architecture (Vaswani et al. 2).

These techniques were later commercialized with the advent of GPT-2, GPT-3, and BERT from AI labs like OpenAI and Google's DeepMind (Wang et al. 3). With increased supply of Graphical Processing Units (GPUs) and Tensor Processing Units (TPUs), these models began snowballing in scale. This was especially evident starting in 2019 with an iteration of GPT-2 being released with a production size of 1.5 billion parameters. In 2020, GPT-3 scaled up to 175 billion parameters — achieving true coherence in reasoning for the first time ever for a machine. GPT-4 was released by OpenAI in 2023, with an undisclosed scale in the trillions of parameters. Development investment also climbed into the hundreds of billions of dollars, with new firms such as Anthropic, Grok, etc. Open sourced projects also gained popularity, some backed by multi-billion dollar R&D teams such as Meta's Llama series.

Functionally, there is no fundamental algorithmic difference between generative and classification models. Indeed, most LLMs are initially trained to generate new sequences of words by setting the loss function to expect the next word in the series of an existing corpus, through a process known as Casual Language Modeling (CLM). For the purposes of commercialization, they have been re-purposed to be prompted as chat-bots by users. This is done by performing back propagation based on the generation of conversational sequences, with the LLM often instructed to act as if filling out a conversation's transcript.

Several underlying technologies are involved in the life cycle of an LLM. The process of creating one usually starts with the definition of a vocabulary. Sequences of language are broken into tokens by algorithms called tokenism's. Tokenizers split text into smaller units, which are then encoded into a vector by another MLP. This is done to develop a sense of meaning via the mathematical similarity of similar words. The similarity of two vectors can be calculated using the cosine-similarity formula, which calculates the angle ϕ between two vectors.

$$\cos\phi = \frac{\vec{A}\cdot\vec{B}}{||\vec{A}||||\vec{B}||}$$

Efforts to increase the performance of LLMs tend to include provisions for an increased vocabulary of cardinal tokens, leading to more efficient generation of text since more complex words, numbers, and symbols would normally need multiple tokens with the use of techniques like Byte Pair Encoding.

Benchmarks for evaluating Large Language Models (LLMs) assess their performance across various tasks, including reasoning, comprehension, generation, and factual accuracy.

Standard benchmarks include GLUE and SuperGLUE for natural language understanding, MMLU (Massive Multitask Language Understanding) for evaluating knowledge across diverse subjects, and BIG-bench for measuring reasoning and generalization capabilities (Ivanov and Penchev 8). HELLASWAG and LAMBADA test commonsense reasoning and long-range dependency understanding, while TruthfulQA and BBQ assess biases, factual consistency, and ethical alignment (6). Additionally, human evaluations and BLEU, ROUGE, and METEOR scores help measure text generation quality. As LLMs advance, new benchmarks continuously emerge to capture nuances in performance, efficiency, and ethical behavior.

Adding to the complexity of creating increasingly more performant are the computational and capital costs of building AI-capable supercomputers, clusters, and data centers for corpora, or CLM text databases. Improvements in model architecture are sought before attempts to increase the scale of models and their parameter counts because of the prohibitive scaling laws of neural networks. Experimentally, it has been found that increased parameter size has an exponential relationship with FLOPs of computational cost (Hoffmann et al. 2). This is seen in relation to the exponentially slowing gain in CLM accuracy with increased compute (5). This is taken to mean that there is a point at which scaling a model to gain accuracy is unsustainable. The Chinchilla scaling law is an experimentally conjectured hypothesis which states that an increase in model scale for a given architecture will tend to reducing model performance as the number of parameters tends to infinity. Although some teams claim to have statistically significant results to disprove it, these results have not been reaffirmed by third parties.

Problem Statement

Table 1: Comparison of LLM Sizes and Their Computational Requirements

Model Name	**Parameters (billions)**	**Training Compute (PF-days)**	**Inference Time (ms/token)**	**Memory Usage (GB)**
GPT-2	1.5	5.6	12	3
GPT-3	175	3,640	75	350
Llama-2-7B	7	184	18	14
Llama-2-13B	13	368	32	26
Llama-2-70B	70	1,720	145	140
Claude 2	$\sim$100	N/A	82	$\sim$200
GPT-4	$\sim$1,500	$\sim$25,000	210	$\sim$3,000

Note: Training compute is measured in petaflop-days. Inference time is measured for a single A100 GPU. Memory usage refers to the VRAM required during inference. Some values for proprietary models are estimated based on public information.

Despite the aforementioned significant advancements in LLMs and their increasingly large reasoning capabilities over time as seen in Figure 1, the mathematical capabilities of models have increased in a sub-linear fashion. This also correlates with the decreasing performance as model sizes, even within the same generation, scale up. For example, Figure 1 displays the Llama-2 generation over 3 different sizes. Current LLMs approach mathematical operations through pattern recognition learned from their training data rather than through formal algorithmic processing, resulting in inconsistent performance when handling numerical calculations beyond simple arithmetic (Hendrycks et al. 3).

This research aims to investigate the potential integration of rule-based tensor mutations within an existing LLM architecture as a mechanism to enable low-cost, rule-driven (as opposed to pattern driven) mathematical computation. The intended outcome of the experiment detailed below include an increase in mathematical accuracy and a decrease in the Inference Time for prompts with necessary mathematical computation associated.

RQ: How can deterministic rule-based tensor mutations be embedded within LLM architectures to enable more accurate and efficient mathematical operations?

The significance of this line of inquiry lies in its potential to address a fundamental limitation of current generative AI systems like ChatGPT, Anthropic's Claude, etc. While specialized numeric compute systems exist (e.g. RAG with Wolfram Alpha), they operate independently of the SIMD, low-latency systems of LLMS, leading to sizable latency in communication. This is especially prevalent in workflows involving both mathematical and linguistic reasoning. The integration of computational resources required for such workflows within LLMs could substantially reduce the computational resources required for complex tasks that involve both natural and language processing and mathematical reasoning.

This investigation focuses specifically on the following mathematical operations:

- Basic arithmetic (addition, subtraction, multiplication, division)
- Matrix Operations (multiplication, inversion, determinant)
- Binary Operations (XOR, AND, NAND, left shift, right shift, OR, complement)
- Array Operations (array sum, as well as the mean, median, mode, standard deviation, variance, and other single variable metrics of a data set)

Furthermore, as previously mentioned, the scope of the experiment is limited to implementing these operations within existing open source LLM architectures of moderate scale (1-7 Billion Parameters) as opposed to developing entirely new architectures. This is both because it is desirable to eliminate all sources of subject variability to help ascertain statistical significance, and because of the readily available weights. Namely, the target model for this paper is the Llama-3-3B model, due to its lightweight and fully open source nature.

Related Works

Prior research has explored various approaches to improving mathematical reasoning capabilities in LLMs, including specialized training on mathematical corpora (Ahn et al. 4). Additionally, other work has been done to fine-tune responses to mathematically demanding prompts using reinforcement learning (Cobbe et al. 1). Others still have tried to add secondary inferences or *Verifiers* to determine accuracy of model outputs when containing computations (2). The immediately evident disadvantage to these approaches is the need for extended training cycles and copious amounts of new corpora. Furthermore, training corpora are required to be similar to testing samples since the strategies outlined above fail to grant models a mechanical performance edge.

Methodology

Define a rule-based function $\mathcal{R} : R^{n\times d} \to R^{n\times d}$ such that:

$$\mathcal{R}(\mathbf{X})_i = \begin{cases} \mathbf{X}_i + \mathbf{X}_{i+1}, & \text{if rule is "sum with right neighbor"} \\ \det(\mathbf{X}_{i:i+2,j:j+2}), & \text{if rule is "3×3 determinant over submatrix"} \\ \mathbf{X}_i, & \text{otherwise} \end{cases}$$

Then pass $\mathcal{R}(\mathbf{X})$ into the modified attention layer: $\mathbf{Z} = \text{Attention}(\mathcal{R}(\mathbf{X}))$

In the above example formulation of a fixed-index mutator function coupled with a self-attention layer for filtering noise from higher confidence inputs, specific transformations are applied at specific indices of the input matrix. Here, $\mathbf{X} \in R^{n\times d}$ represents the input tensor for a given layer. d is the embedding dimensionality, where the function $\mathcal{R}$ applies specific, discontinuous, logic based on identified patterns of the input located at specific indices. The mutated embedding is then processed through the standard attention mechanism to produce the output representation $\mathbf{Z}$. This output is then concatenated with the parallel layer(s) of the original neural network, allowing for an expansion of the network's capacity for operations without need for compromising on the token-based original throughput.

Location-Based Rule Selection

A critical aspect of the methodology is the mechanism responsible for determining which rule is applicable at each position within the input matrix $\mathbf{X}$. Rather than relying on stochastic selection, this approach implements a deterministic, location-based rule selection strategy that leverages the contextual information encoded within the model's representations. A major advantage of this fixed-index approach is the minimization of dynamic surfaces in the model's cost function, thereby reducing the amount of noise in output, as well as reducing the required amount of training as over fitting is a non-issue without randomness.

Fixed-Index Architecture

The implementation architecture utilizes predetermined index relationships within the tensor space. For example, indices $(0, 1)$ and $(1, 2)$ might have a fixed relationship where their values are added together and output at index $(0, 4)$. This fixed-index approach creates explicit computational pathways within the neural network architecture, allowing for deterministic mathematical operations without disturbing the stochastic nature of the remaining network. The fundamental advantage of this approach lies in its compatibility with modern hardware acceleration techniques, particularly Single Instruction Multiple Data (SIMD) operations that commonly take place on GPUs. These GPUs are already leveraged for matrix multiplication in the vast majority of existing AI/ML runtimes.

Each mathematical operation type is assigned specific input and output indices within the tensor, creating a predictable computational graph that can be optimized during compilation using the CUDA compiler, `gcc`, and manual assembler optimization like with DeepSeekV3 (DeepSeek-AI et al. 16). Addition operations, for instance, use indices (i, j) and $(i + 1, j)$ as inputs, with results stored at $(i, j + d/2)$, effectively partitioning the embedding space into operand and result regions. Multiplication operations utilize indices (i, j) and $(i, j + 1)$ as inputs, with results projected to $(i + 1, j + d/2)$, maintaining a consistent pattern of spatial relationships within the tensor. More complex operations like matrix determinant calculations employ a 3×3 submatrix starting at index (i, j) with results consolidated at $(i + 3, j)$. This systematic approach to index mapping enables highly efficient computation on GPU architectures, as the fixed patterns allow for optimized memory access patterns due to hard-coded indexing at compile time, and reduced cache thrashing during tensor operations. Modern GPUs excel at these fixed-pattern operations, particularly when they can be expressed as fused operations within CUDA kernels or optimized through tensor cores designed specifically for matrix multiplication(NVIDIA).

The architecture maintains parallel processing paths that preserve the dual nature of the system's capabilities. The standard language processing path continues to leverage the probabilistic, statistical nature of the transformer architecture, preserving the original LLM capabilities that have proven effective for natural language understanding and generation. Simultaneously, the mathematical computation path applies fixed-index transformations for specific operations, creating a deterministic subsystem within the larger stochastically variant network. These parallel streams capitalize on the inherent parallelism of GPU architectures, allowing different CUDA cores and cache regions to process distinct streams simultaneously. The fixed-index nature of the mathematical operations enables compiler optimizations that can allocate dedicated tensor cores for these operations, maximizing throughput and minimizing latency. Existing models, as shown in Figure 1 tend to use far more VRAM than cores, leading to an allocation inefficient in terms of performance per millisecond of inference. The paths are later merged through concatenation and a projection layer, a process that similarly benefits from the warp-level primitives available in modern GPU architectures for efficient tensor manipulation.

The attention mechanism serves as a noise filter and integration component, allowing the model to selectively focus on either standard language representations or mathematically transformed representations based on input context. This selective focusing behavior effectively routes information through the appropriate pathway based on the input's semantic requirements. From a hardware acceleration perspective, this mechanism benefits from the recent advancements in GPU architecture specifically designed for transformer models. The attention operations leverage dedicated tensor cores in NVIDIA's Ampere and Hopper architectures, which provide specialized hardware acceleration for matrix multiplication and accumulation operations at various precisions. The fixed-index nature of the approach enables further optimization of these operations through persistent CUDA kernels that maintain tensor data in high-bandwidth on-chip memory (L3-L4 cache), reducing expensive global memory access operations during the attention computation phase.

Implementation Hardware Software

Rust was selected for its memory safety guarantees, zero-cost abstractions, and deterministic concurrency model. The neural network is implemented using the `burn` crate, a modular, backend-agnostic deep learning framework designed for Rust. Burn enables explicit architectural definition via trait-based modules and supports GPU acceleration using backends such as `burn-wgpu` and `burn-candle`. This design aligns with IB Computer Science principles of modularity, abstraction, and system performance.

```
[dependencies]
burn = "0.12"
burn-wgpu = "0.12" log = "0.4"
env_logger = "0.10"
```

The system targets an MSI RTX 4090 (24GB VRAM, 900W), utilizing `burn-wgpu` to leverage WebGPU for training on tensor cores. This setup maximizes throughput for floating-point operations critical in gradient descent and backpropagation.

```
fn main() {
  env_logger::init(); info!("Training initialized");
}
```

The `log` crate provides structured runtime logging, while `env_logger` parses environment variables to configure log levels. Logging supports traceability, a key aspect of IB standards emphasizing system reliability and maintainability. Modular logging also illustrates core software engineering practices, such as separation of concerns and system observability, during neural network training and mutation processes.

Works Cited

Ahn, Janice, et al. Large Language Models for Mathematical Reasoning: Progresses and Challenges. 2024. *arXiv*, arxiv.org/abs/2402.00157.

Cobbe, Karl, et al. Training Verifiers to Solve Math Word Problems. 2021. *arXiv*, arxiv.org/abs/2110.14168.

DeepSeek-AI, et al. DeepSeek-V3 Technical Report. 2025. *arXiv*, arxiv.org/abs/2412.19437.

Hendrycks, Dan, et al. Measuring Mathematical Problem Solving With the MATH Dataset. 2021. *arXiv*, arxiv.org/abs/2103.03874.

Hoffmann, Jordan, et al. Training Compute-Optimal Large Language Models. 2022. *arXiv*, arxiv.org/abs/2203.15556.

Ivanov, Todor, and Valeri Penchev. AI Benchmarks and Datasets for LLM Evaluation. 2024. *arXiv*, arxiv.org/abs/2412.01020.

NVIDIA. 1. Introduction - CUDA C++ Programming Guide. *1. Introduction - CUDA C++ Programming Guide*, Feb. 2025. docs.nvidia.com/cuda/cuda-c-programming-guide/.

Testolin, Alberto. “Can Neural Networks Do Arithmetic? A Survey on the Elementary Numerical Skills of State-of-the-Art Deep Learning Models”. *Applied Sciences*, vol. 14, no. 2, 2024. https://doi.org/10.3390/app14020744.

Vaswani, Ashish, et al. Attention Is All You Need. 2023. *arXiv*, arxiv.org/abs/1706.03762.

Wang, Zichong, et al. “History, development, and principles of large language models: An introductory survey”. *AI and Ethics*, Oct. 2024. https://doi.org/10.1007/s43681-024-00583-7.