Efficient Post Training Pruning of LLMs

Keming Li [1] **Raj Mohanty (lead)** [1] **Kenneth Pat** [1] **Omar Samiullah** [1]

Abstract

Large Language Model (LLMs) have been ubiquitous in our daily lives in the last few years and their usage is expected to grow. LLM inference relies on high-end GPU hardware that is very expensive, scarce, and consumes a lot of energy. Hence, there has been an increased focus on efficient inference computation by various strategies. In this project, we present many solutions for post-training pruning of LLMs, incorporating several innovative contributions from recent work and our contribution. Our approach includes layer-level pruning, demonstrating the speedup achieved through 2:4 semi-structured pruning, genetic algorithm-based pruning, and exploring different versions of the Relative Importance and Activations (RIA) metric.

Experiment results show the desired speed-up from semi-structured pruning using the RIA metric. Layer-wise and genetic algorithm-based methods show continuous improvement in performance close to the Dense model.

1. Introduction

With Large Language Models (LLMs) becoming more prevalent in applications related to artificial intelligence, it becomes necessary to optimize their sizes in order to efficiently store them while still meeting requirements in performance. As a result, various pruning methods were developed and implemented by removing unnecessary parameters to facilitate the process of size reduction on LLMs.

Since LLMs are expensive to train, during-training pruning or sparse training, becomes infeasible. Hence the focus is on Post Training Pruning (PTP) which aims to remove unnecessary weights and do it in a semi-structured manner to take advantage of sparse acceleration libraries like cuSPARSELt or cuTLASS. Our goal is to also make sure the performance of the LLM doesn't degrade compared to the dense model as we increase sparsity to take advantage of the speedup.

Our goal for this project is to design, implement, and perform experiments with different pruning metrics (like Relative Importance or RI [1], magnitude, etc.) and $N : M$ sparsity methods (like Channel Permutation [1], vanilla $N : M$ sparsity method) on LLMs (e.g. Llama 3-8B) to reduce the model size and computation requirements and at the same time maintain performance close to the dense model for zero-shot tasks without any fine-tuning.

2. Background / Prior Work

Our ideas emerged from a new conference paper titled "Plug-and-Play: An Efficient Post-training Pruning Method for Large Language Models" (Zhang et al., 2024). It was published earlier this year at the International Conference on Learning Representations (ICLR) 2024, which took place in May. Additionally, there are two more papers that contributed to the shape of our ideas namely Wanda (Sun et al., 2024) and SparseGPT (Frantar & Alistarh, 2023).

2.1. Relative Importance and Activation (RIA)

The paper introduced an important concept known as "relative importance and activation (RIA)," which is a pruning metric that incorporates the factor of "activations" into "relative importance." It states that, for each element $\mathbf{W}_{ij}$, the method would combine l_2-norm of activation vectors $||\mathbf{X}||_2$ as:

$$\mathbf{RIA}_{ij} = \mathbf{RI}_{ij} \times (||\mathbf{X}_i||_2)^\alpha = \left(\frac{|\mathbf{W}_{ij}|}{\sum|\mathbf{W}_{*j}|} + \frac{|\mathbf{W}_{ij}|}{\sum|\mathbf{W}_{i*}|}\right) \times (||\mathbf{X}_i||_2)^\alpha \quad (1)$$

Figure 1 shows the benefit of using the RIA metric instead of the overall magnitude of the weights. This is because if magnitude-based pruning is done the weight at cell (4,4) will be pruned leading to an entire channel being corrupted. Since the weight relative to its input and output channel is important, the relative importance metric is shown in equation 1. Because of the issues related to activation outliers, the L2-norm of the activation $||\mathbf{X}_i||_2$ is multiplied by the metric. To control the strength of the activation, it is raised

*Equal contribution [1]University of California, Irvine. Correspondence to: Keming Li <kemingl1@uci.edu>, Raj Mohanty <mohantyk@uci.edu>, Kenneth Pat <patk@uci.edu>, Omar Samiullah <osamiull@uci.edu>.

Proceedings of the 39th International Conference on Machine Learning, Baltimore, Maryland, USA, PMLR 162, 2022. Copyright 2022 by the author(s).

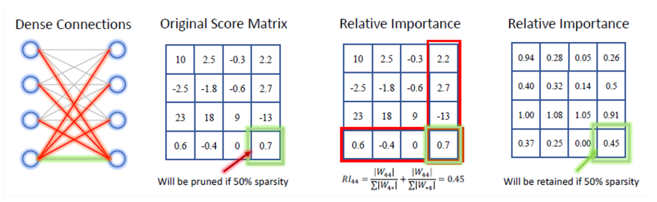

Figure 1: An example of Relative Importance. The connection at position (4,4) will be removed based on global network magnitude. However, it is retained when evaluated for its significance within relative connections.(Zhang et al., 2024)

to the power of a. Based on our experimental results the best values of a are around 0.2 to 0.6.

2.2. Channel Permutation

N:M sparsity is usually favored by post-training pruning given its practical speed-up on specific hardware that support libraries like cuSPARSELt (Blog, 2020). Figure 2 shows why channel permutation is important for N:M sparsity compared to the regular N:M sparsity method followed where N out of M contiguous weights are removed based on the chosen pruning metric. As a first step, a huristic channel allocation is followed. The sum of weight importance for each input channel is calculated and channels are sorted into K blocks. A heuristic allocation strategy is used to ensure important channels are retained within each block with N:M sparsity. Channels are alternately allocated into blocks, repeating the process until all channels are assigned. For example, with 8 input channels and 4 output channels under 2:4 sparsity, channels are distributed among blocks to maximize weight importance. This heuristic approach significantly improves the sum of retained weight importance scores compared to direct N:M pruning.

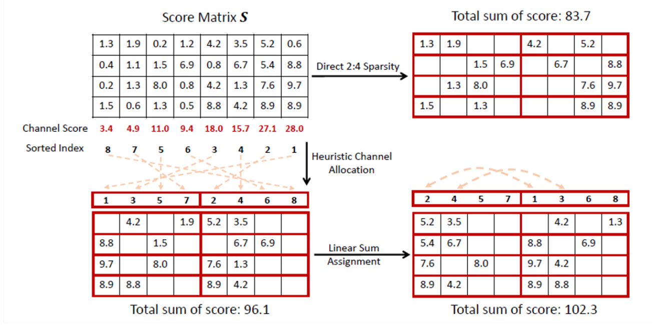

Figure 2: Illustration of Channel Permutation. Given a score matrix S assigned by various criteria, directly processing it with 2:4 sparse results in a total sum of the retained score being 83.7. However, by using channel permutation, we could get a final total sum of score 102.3.(Zhang et al., 2024)

To further improve this, a Linear Sum Assignment (LSA) is applied. The Hungarian algorithm is used to solve this problem, efficiently reassigning channels to blocks. This reassignment improves the weight importance sum, where swapping channels results in a further score sum improvement

2.3. Metric of Evaluation

$$\text{Perplexity}(P) = \left(\prod_{i=1}^{N} \frac{1}{P(w_i)} \right)^{\frac{1}{N}} \quad (2)$$

To evaluate the performance of a LLM, Perplexity (PPL) is normally used. Perplexity is a measure of how well a probabilistic model predicts a sample, defined as the exponentiation of the average negative log-likelihood of the test data, with lower values indicating better predictive performance.

PPL is the Average number of words to guess from vocabulary to predict the next word correctly. In extreme cases,a Random Model with P(wi) = 1/V has PPL = V , where V = size of vocabulary where as a Perfect model with P(wi) = 1 will have PPL = 1. So lower PPL is better.

2.4. Evolution Strategy

Since in the next few sections of this report, we will introduce using Evolution Strategy to do the post-training pruning. Hence, in this chapter, we will introduce the most popular Evolution Strategy (ES), their potential of solving post-training pruning problem in terms of strengths and weaknesses.

Evolution Strategy (ES) is a class of optimization algorithms inspired by the principles of natural evolution. ES is particularly well-suited for solving black-box optimization problems, where the objective function is complex, noisy, and without gradient information. The basic idea is to iteratively improve a population of candidate solutions through mutation, selection, and recombination processes.

Basic Idea of Evolution Strategy: ES generally follows these steps: (1) Initialization: Start with a randomly generated population of candidate solutions. (2) Evaluation: Assess the fitness of each candidate based on the objective function. (3) Selection: Select the best-performing candidates to form the next generation. (4) Mutation: Apply random changes to the selected candidates to explore new solutions. (5) Recombination: Optionally combine pairs of candidates to produce offspring. (6) Iteration: Repeat the evaluation, selection, mutation, and recombination steps until convergence or a stopping criterion is met.

(1+1)-ES: The general idea is that a single parent generates one offspring through mutation. The better solution between the parent and the offspring is selected for the next generation (Schwefel, 1981). The strengths of this method include its simplicity, ease of implementation, and suitability for small decision spaces. However, it has limited exploration ability, is prone to local optima, and is inefficient for large decision spaces. This method is effective for small decision spaces and single-objective optimization.

(μ/μ,λ)-ES: In this method, a population of μ parents generates λ offspring through recombination and mutation. The best μ offspring are selected for the next generation (Rechenberg, 1994). It balances exploration and exploitation, offers robust performance, and can handle moderate decision spaces. The weaknesses include increased computational cost and the potential need for fine-tuning parameters. This method is suitable for moderate decision spaces and can handle both single and multi-objective optimization.

Covariance Matrix Adaptation Evolution Strategy (CMA-ES): This method uses a multivariate normal distribution to sample offspring, adapting the covariance matrix to capture the dependencies between variables (Hansen & Ostermeier, 2003). Its strengths are excellent exploration capability, good adaptation to the landscape of the objective function, and robust performance on large decision spaces. The main weakness is its high computational cost, and it may struggle with highly noisy functions. CMA-ES is highly effective for large decision spaces, well-suited for single-objective optimization, and can be extended to multi-objective problems.

Multi-Objective Evolution Strategy (MOES): This method extends ES to handle multiple objectives by maintaining a diverse set of solutions and using Pareto dominance to guide selection (Deb et al., 2002). The strengths include the ability to find a diverse set of trade-off solutions and suitability for complex multi-objective problems. The weaknesses are increased computational complexity and the challenge of managing diversity. MOES is effective for multi-objective optimization and scalable to moderate decision spaces.

Using ES for LLM post-training pruning presents several challenges. The large decision space involves billions of parameters, which is significantly larger than typical ES applications. Designing an ES algorithm that can efficiently explore such a vast decision space is challenging. Additionally, pruning large models must be done within the constraints of available memory, requiring memory-efficient strategies for candidate solution generation and evaluation. The pruned model must meet specific sparsity requirements, maintaining performance while reducing parameters. Ensuring that the ES algorithm can enforce sparsity constraints during the optimization process is critical. Evaluating the fitness of pruned models can be computationally expensive, as it involves assessing the performance of large neural networks. Implementing efficient fitness evaluation techniques to reduce computational overhead is necessary.

Potential solutions and strategies include hybrid approaches that combine ES with gradient-based methods to leverage the strengths of both approaches. For example, using ES to explore the decision space and fine-tuning promising candidates with gradient descent. Dimensionality reduction techniques can also be applied to reduce the effective dimensionality of the decision space, such as using principal component analysis (PCA) or other methods to focus on the most impact parameters. Efficient sampling techniques, like importance sampling or adaptive sampling, can explore the decision space more effectively. Leveraging parallel and distributed computing can handle the computational demands of ES, such as implementing ES on a distributed computing platform to speed up fitness evaluations and population management.

3. Approach

We tried various approaches apart from the ones used in (Zhang et al., 2024).

3.1. Tweaking the Activation strength parameter a

In equation 1, the parameter a is used to change the strength of the activation $||X_i||2$. We tried unstructured pruning at 50% sparsity with $a = 0.5$ and $a = 0.3$ to see how sensitive is PPL to it.

3.2. Pruning more on MLP layers vs Self Attention Layers

In transformer models, self attention layers are quite important compared to the regular multi layer perceptron or MLP layers. We tried to prune MLP layers at 50% sparsity and the self attention layers at 25% sparsity leading to a overall sparsity of 41.71%. For comparison we pruned all layers at 41.71%.

3.3. Magnitude Monte-Carlo Pruning

This is one of the simplest pruning methods, where weights with the smallest magnitudes are pruned. It is an unstructured pruning approach that does not consider the structure of the neural network. It is accomplished in three steps: **1.** Try pruning layer by layer to find top 10-20% of the lowest weights. **2.** Try it again on a row-by-row basis, and **3.** Try it again on a column-by-column basis.

3.4. Gradient Pruning

In this approach, weights are pruned based on their gradients during training. Weights with small gradients are pruned as they are considered less important for the model's learning. **1.** If the gradient is small, prune the corresponding weight. **2.** If the gradient is large, prune the corresponding weight. The idea is that weights with smaller gradients contribute less to the overall loss and can be pruned without significantly affecting the model's performance. In the provided code, the gradients of all trainable parameters are retrieved using the following line: W_grads = name: param.grad.data.clone() for name, param in model.named_parameters() if param.requires_grad

For each layer, the code calculates a metric W_metric based on the absolute value of the gradients (W_metric = torch.abs(W_grads[name])). The weights with the smallest W_metric values are then pruned by setting them to zero using a pruning mask.

3.5. Entropy-Based Pruning

Entropy-based pruning is a method that prunes weights based on the entropy of their distribution. The entropy of a weight distribution is a measure of its uncertainty or randomness. Weights with low entropy (more uniform distribution) are considered less informative and are pruned. The code calculates the entropy of the weight distribution along the output dimension using the following line: python W_metric = -(W * torch.log(W + 1e-8)).sum(dim=0) # Entropy along output dimension

The 1e-8 term is added to avoid numerical issues with logarithms of zero values. The weights with the lowest entropy values (more uniform distribution) are then pruned by setting them to zero using a pruning mask.

3.6. Structured Magnitude Layer Pruning from Matrices

In this section, we implemented structured pruning on multiple layers. Recall that layer has 6 matrices (also know as "projections"): `Q`, `K`, `V`, `Gate`, `Up`, and `Down`. They are represented in the code as `args.matrix = ["q_proj", "k_proj", "v_proj", "o_proj", "gate_proj", "up_proj", "down_proj"]`. Following this pattern, we execute our pruning algorithm based on the matrix chosen by the user. When running the program from the command prompt, users can select the desired matrix by using the `--matrix` follow by one of the aforementioned matrix type.

3.7. Pruning different methods across all layers

The code demonstrates the ability to apply different pruning methods across all layers of the model. It iterates over the layers of the model and applies the chosen pruning method (magnitude, gradient, or entropy) to specific subsets of layers, such as q_proj, k_proj, v_proj, o_proj, gate_proj, up_proj, and down_proj. This allows for selective pruning of different components of the model using different pruning strategies.

3.8. Evolution Strategy for LLM Post-Training Pruning

To address the challenges of pruning large language models (LLMs) with billions of parameters, we employ a Genetic Pruning algorithm based on Evolution Strategy (ES). This method iteratively improves a pruning mask that determines which weights in the neural network should be zeroed out. The algorithm leverages evolutionary principles such as mutation and selection to explore the vast decision space efficiently, handle memory limitations, and maintain specific sparsity levels.

Challenges Addressed:

- **Large Decision Space:** Efficient exploration of a vast number of possible pruning configurations.
- **Memory Limitations:** Managing high memory demands and computational cost of evaluating pruned models.
- **Maintaining Sparsity Levels:** Achieving a specific sparsity level while maintaining acceptable performance.

Solutions:

- **(1+1)-ES based algorithm:** Maintaining a population of at most 2 (parent and its single child) to minimize the GPU VRAM usage.
- **Probabilistic Flipping:** Using a function to probabilistically flip elements in the pruning mask based on the weight magnitudes, guided by mutation rates that adapt over generations. As shown in Algorithm 1, at line 3 and 19, we keep two mutation rate number m_1 and $m2$ such that $m1 \times S = m2 \times (1 - S)$, where $m1$ is the mutation rate for elements that is 1 flipping to 0, $m2$ is the mutation rate for elements that is 0 flipping to 1, and S is the sparsity level in range $[0, 1]$.
- **Efficient Fitness Evaluation:** Evaluating the fitness of each candidate based on the model's performance, allowing iterative improvement of the pruning mask while adhering to memory constraints.

The algorithm begins with the initialization of parameters such as population size and the number of generations. Mutation rates m_1 and m_2 are set based on the desired sparsity level. The *ObjFunction* evaluates the performance of a

Algorithm 1 Genetic Pruning Evolution Strategy

```
 1: Input: model, tokenizer, dev, args
 2: Output: best_individual, best_fitness
 3: Initialize population_size, num_generations, m1, m2
 4: best_individual = GETFIRSTINDIVIDUAL(model)
 5: best_fitness = OBJFUNCTION(model, best_individual)
 6: for generation in 0 to num_generations-1 do
 7:     for candidate_num in population do
 8:         child = copy of best_individual
 9:         for each mask in child do
10:             Apply flip_matrix to mutate mask
11:         end for
12:         child_fitness = OBJFUNCTION(model, child)
13:         if child_fitness ≤ best_fitness then
14:             best_individual = child
15:             best_fitness = child_fitness
16:         end if
17:     end for
18:     Adjust mutation rates m1 and m2
19: end for
20: Output: best_individual and best_fitness
```

Algorithm 2 Objective Function

```
1: function OBJFUNCTION(model, mask)
2:     Copy model to local_model
3:     Apply mask to local_model
4:     Evaluate local_model performance
5:     return performance metric as fitness
6: end function
```

Algorithm 3 Get First Individual

```
1: function GETFIRSTINDIVIDUAL(model)
2:     Copy model to local_model
3:     Prune local_model to initial sparsity level # can be
   RIA or any pruning method
4:     Generate and return initial_mask from local_model
5: end function
```

pruned model by creating a deep copy of the model, applying the pruning mask, and evaluating the model using a specified dataset.

The *GetFirstIndividual* function generates an initial pruning mask by pruning the model to the initial sparsity level and creating a mask from the pruned weights. The evolutionary loop iterates over the population, creating new child (candidate) pruning masks by probabilistically flipping elements in the current best mask. The *flip_matrix* function is used to mutate the masks based on the weight magnitudes, ensuring guided mutations. Each candidate mask is evaluated, and if a candidate yields better performance, it replaces the current best mask. Mutation rates are adjusted (getting less and less, but not less than a threshold) after each generation to refine the search process.

Finally, the best pruning mask is applied to the model, zeroing out the weights according to the mask, and the best solution is output.

4. Analysis

4.1. Activation strength and Layer specific pruning results

Method	PPL	CUDA Time Avg	Speedup
Dense	5.47	1.962	1x
RIA (a=0.5) Unstructured 50% Sparsity	6.8046	1.961	1x
RIA (a=0.3) Unstructured 50% Sparsity	6.7977	1.962	1x
2:4 Semi Structured	11.5	1.059	1.85x
RIA (a=0.5) Unstructured 50% MLP 25% ATTN (41.71% Overall)	6.4221	1.962	1x
RIA (a=0.5) Unstructured 41.71%	6.0618	1.961	1x

Table 1: Performance comparison changing a and trying different sparsity by layer type

Table 1 shows the performance in terms of PPL values various pruning methods we tried on a Nvidia A10G GPU with 24GB VRAM. Since we are using the input activation in our pruning metric, we needed a calibration dataset (C4). To get the PPL results we used wikitext2 as our evaluation dataset. Llama-2-7b was used a the LLM for this experiment.

From the results we can see that $a = 0.3$ gets a slightly better PPL value as compared to $a = 0.5$. The result for the pruning where we pruned MLP layers at 50% and self attention layers at 25%, reaching an overall sparsity of 41.71%, we see a higher PPL value compared to pruning all layers equally at 41.71%. This results was suprising to us since we assumed that the self attention layer are more important than the MLP layers. Hence preserving more weights in the self attention layer could lead to better PPL results.

In terms of average speedup, the average time of `aten::_cslt_sparse_mm` vs `aten::mm` were compared. `aten::_cslt_sparse_mm` is sparse matrix multiplication using the accelerated cuSPARSELt library of Nvidia (Blog, 2020) vs the stand matrix multiplication aten::mm. We observe a 1.85x speed up which show the power of semi-structured N:M pruning.

Figure 3 shows the VRAM usage over time as we prune using the C4 calibration dataset and then calculate PPL using the wikitext2 evaluation dataset on Llama-2-7b LLM. Since our VRAM was 24GB, we did not face any out of memory issues. We also notice that during pruning the VRAM fluctuates a lot vs when we just run inferences.

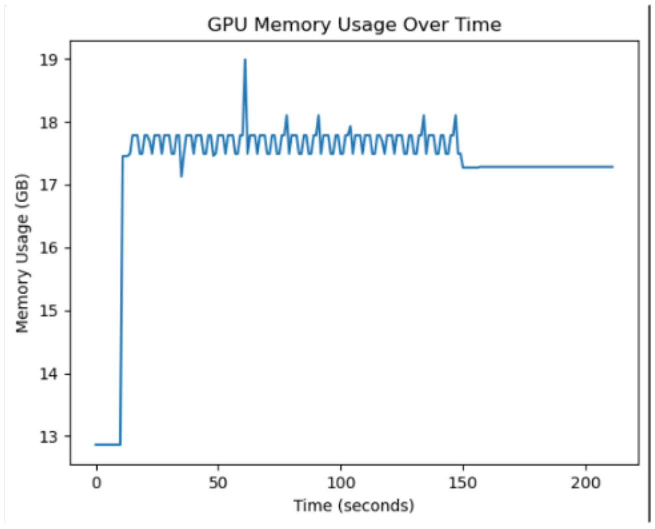

Figure 3: VRAM usage during pruning and evaluation

4.2. Weight Distribution Analysis

To understand the weight distribution across different types of matrices in the Llama-3-8B model, we conducted a statistical analysis as illustrated in Figure 4. The pie chart reveals the percentage of weights occupied by each type of matrix. Notably, up-sampling matrices constitute 26.9% of the model's weights, which is the same proportion as down-sampling matrices and gate matrices, each also comprising 26.9%. This significant proportion suggests that these matrices play a crucial role in the model's overall structure and functionality. Other matrices, such as Q and K matrices, each account for 7.7% of the weights, while V and O matrices are relatively minor, comprising 1.9% each.

4.3. Pruning Impact Analysis

Figure 5 presents the results of experiments where each line represents a scenario in which only one specific type of matrix was pruned. The x-axis denotes the matrix sparsity level, indicating the percentage of weights pruned from the specified matrix, while the y-axis measures the model's perplexity post-pruning, using C4 as the calibration dataset and Wikitext-2 as the evaluation dataset. The pruning method employed is unstructured pruning.

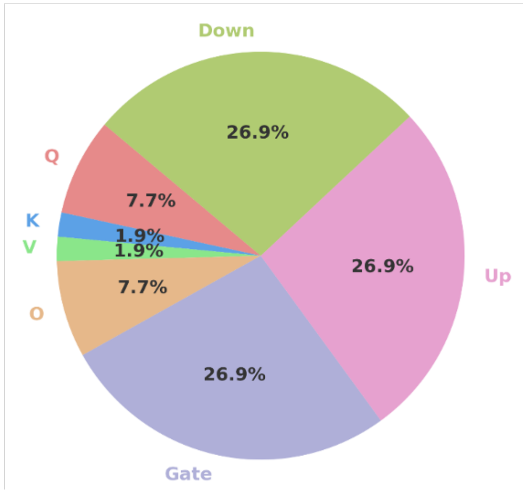

Figure 4: Weight distribution of Llama-3-8B.

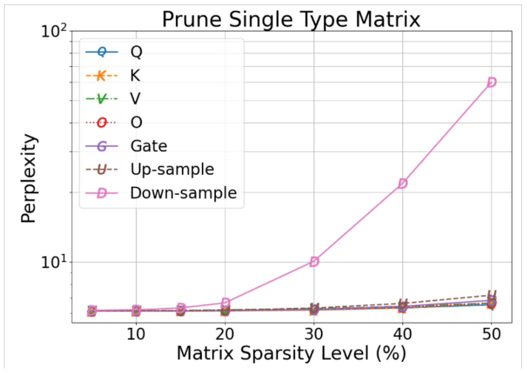

Figure 5: Prune specific matrix experiment result.

From the experiment, it is evident that pruning the down-sampling matrices (represented by the pink line) has the most significant impact on the model's performance. For instance, at a 10% sparsity level, the perplexity increases substantially, indicating a deterioration in model performance. This suggests that down-sampling matrices are critical to maintaining the model's functionality.

Further Observations Analyzing both images, we can draw several interesting conclusions. Firstly, despite the equal weight distribution of up-sampling, down-sampling, and gate matrices, the down-sampling matrices are evidently more sensitive to pruning. This indicates a higher dependency of the model's performance on the integrity of down-sampling matrices compared to up-sampling or gate matrices. Additionally, matrices such as Q, K, V, and O, which occupy a smaller proportion of the total weights, exhibit relatively minor changes in perplexity when pruned, suggesting their lesser impact on the overall performance. This observation can be reasoned by their specialized roles in the model, where redundancy might be inherently higher, allowing for more aggressive pruning without significantly affecting performance.

4.4. Evolution Strategy Experiment Results

The experimental results for the Evolution Strategy (ES) algorithm applied to LLM post-training pruning are illustrated in Figures 6 and 7. The x-axis in each plot denotes the number of individuals generated and evaluated for fitness, while the y-axis represents the model's perplexity after pruning, using C4 as the calibration dataset and Wikitext-2 as the evaluation dataset. The point at individual generated at 0 indicates the initial individual generated by some other algorithm. The first figure uses an unstructured pruning model as the initial individual in ES, while the second figure employs an initial individual generated by the RIA method(Zhang et al., 2024).

Results with Unstructured Initialization. Figure 6 shows the results of the ES algorithm using an unstructured pruning model as the initial individual. The three sub-images correspond to experiments with sparsity levels (SL) of 0.5, 0.7, and 0.9. For SL=0.5, the perplexity rapidly decreases and stabilizes as more individuals are generated and evaluated. This trend is consistent across all sparsity levels, though higher sparsity levels result in higher final perplexity values. This indicates that while the model can adapt to pruning through ES, the degree of pruning directly impacts the model's performance, with more aggressive pruning leading to higher perplexity.

Results with RIA Initialization. Figure 7 presents the results using the RIA method as the initial individual. The three sub-images also correspond to sparsity levels of 0.5, 0.7, and 0.9. Similar to the unstructured initialization, the perplexity decreases rapidly as individuals are generated and evaluated. However, it is notable that the final perplexity values are lower compared to the unstructured initialization, particularly at higher sparsity levels. This suggests that the RIA initialization provides a better starting point for the ES algorithm, leading to more efficient pruning and better overall model performance.

Motivation and Analysis. The motivation for using the ES algorithm for LLM post-training pruning is twofold:

1. ES can always use heuristic solutions as seeds to explore better-pruned models.
2. In terms of computation, ES is much cheaper than retraining or fine-tuning. The major cost is in model evaluation, which is inference. Thus, the better the ES performs, the more efficient the inferences, and vice versa, creating mutual promotion.

The experimental results align well with this motivation. The ES algorithm effectively reduces perplexity, indicating successful pruning. Additionally, the RIA initialization outperforms the unstructured initialization, demonstrating that starting with a more structured initial individual can lead to better outcomes.

Some opportunities to be improved is that we could see the proposed ES algorithm stuck at some local optimal when we compare Figure 6 and Figure 7. This is due to the GPU VRAM is small compared to the size of LLMs, leading to the cons that we are only able to design the population number as two, which means we almost do not have the diversity of population to increase our ability to explore the search space. In the future, we might need to solve this problem.

4.5. Impact from the Choice of Layers

Matrix	**0.05**	**0.10**	**0.15**	**0.20**	**0.30**	**0.40**	**0.50**
gate	6.1364	6.1390	6.1474	6.1631	6.2335	6.4206	6.8217
up	6.1361	6.1422	6.1563	6.1858	6.3157	6.6117	7.1816
down	6.1749	6.2084	6.3200	6.6528	10.0610	21.8864	59.9448

Table 2: Perplexity values based on matrices and sparsity levels

The table above shows all the perplexity values we got by applying structured pruning across different matrices. We can see that the choice of matrix plays a significant role in affecting the results (i.e. perplexity values). For the chosen large language model, the most optimal matrix for selection in structured pruning is the gate matrix, as it only produced a small increase with increasing the sparsity level from 0.05 to 0.5, while keeping the perplexity values under 7. The up matrix is the second most optimal, when applying structured pruning at 0.50 sparsity level, it has a perplexity value slightly above 7, which is considered a mild increase. The worst choice for structured pruning would be the down matrix since its perplexity level skyrocketed as the sparsity level increased.

5. Methodology

For our project, we used the Meta Llama 3-8B as our large language model and Google's C4 dataset as our calibration dataset. Both repositories (repo) are hosted on HuggingFace and available for non-commercial use. We used the code from the repos provided in the paper and then developed our own code to let it execute on the aforementioned LLM.

During our development process on HPC3, we ran into several major problems. First, when we tried to download the entire C4 dataset, which has a size of 305 GB, we found it exceeded the given disk quota. Luckily, we found out later that we can save our files under `/pub/UCINetID`, which we are allowed to have 1 TB of storage for each user. After we finished downloading it, we used the `gunzip` command

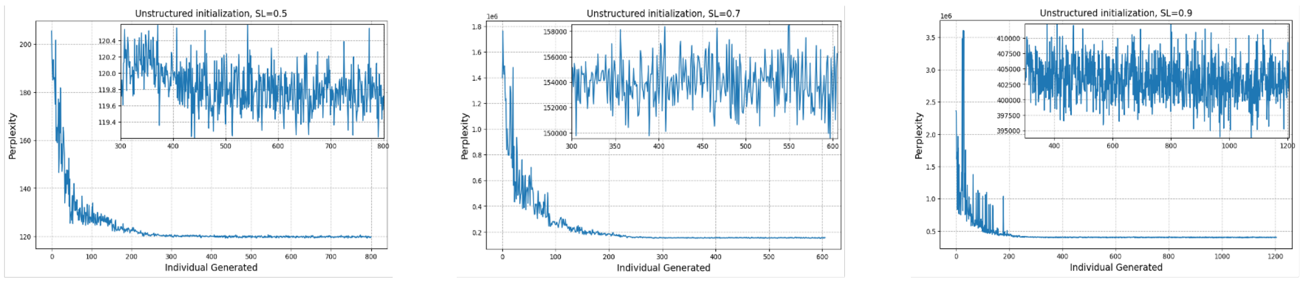

Figure 6: SL=0.5, 0.7, and 0.9 for initialization as unstructured pruning

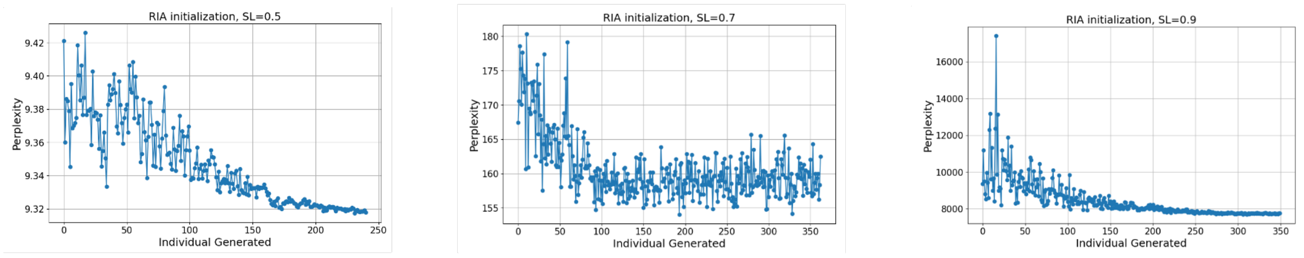

Figure 7: SL=0.5, 0.7, and 0.9 for initialization as RIA

to unzip the first training file and the first validation file. Additionally, we also needed to download the full set of LFS files for Meta Llama 3-8B. After we successfully cloned these two repos, we were also required to use the Git Large File Storage (`git-lfs`) to install their dependencies.

Next, we needed to prepare our Linux environment with the pruning code. To achieve this, we cloned the two repos mentioned in the paper, created a Python environment using Conda (we used `miniconda3/4.12.0`), and then used the `pip` command to install all the required packages, including PyTorch and SentencePiece.

When we were testing our code, one of the most prevailing problems was the lack of GPU memory. Since executing pruning methods on LLMs requires a substantial amount of memory, we tried to request GPU nodes with larger memory sizes – the Nvidia A30 (with 24 GB of VRAM) and A100 (with 80 GB of VRAM), but with the extremely competitive situation of everyone requesting computation resources on HPC3, we sometimes have to wait to get our desirous GPU nodes.

For the results in section 4.1, we used an Nvidia A10G GPU with 24GB VRAM. We modified the code from the authors of (Zhang et al., 2024) to run all experiments and then compiled the results in section 4.1. Setting up the environment was challenging since it required to make sure all pytorch libraries are compatible with the environment we were using. Understand the code and then changing the part of the code to accomplish our tasks were relatively challengining.

6. Ownership

Raj identified the paper for the project, read and understood the paper, experimented with layer-wise sparsity and 2:4 semi-structured for speedup, compiled results, and presented the prior work and methodology. He also contributed significantly on the final project report.

Omar experimented with layer-wise sparsity and 2:4 semi-structured and unstructured sparsity for different pruning methods. He compiled results and presented findings of the various pruning strategies.

Keming read the source code, helped all others solve environment problems, and wrote/debugged what they needed. He implemented and experimented on pruning profiles for different matrices and designed, implemented, and experimented on an evolution strategy on pruning based on both RIA and magnitude.

Kenneth experimented with structured magnitude layer pruning on different sparsity levels. He created the Github repository, imported the ICML 2022 template onto Overleaf for the final report, and wrote the introduction, background/prior work, and methodology sections of the final report.

7. Conclusion

Post training pruning (PTP) along with quantization and parameter efficient fine tuning are the future of LLMs for domain specific tasks. Effective use of these methods would lead to democratization of LLMs where a much smaller

7b parameter model can reach the performance of a much larger GPT-4 general LLM that is estimated to have trillions of parameters and requires a lot of resources and power for hosting. We see the incorporation of PTP in various areas.

N:M semi-structured pruning leads to speedup if the hardware supports cusparselt or cutlass libraries. Careful design of pruning metric and channel permutation method leads to much better results compared to just standard magnitude based pruning or standard 2:4 pruning. We profiled only pruning specific matrix and show down-sampling is important.ES methods developed show better perplexity compared to all SOTA methods.

In terms of future work, layer specific pruning needs to be explored further. Channel permutation utilizes the Hungarian algorithm for Linear sum assignment. Other linear sum assignment methods or Reinforcement Learning methods could be applied to see if they achieve better results customized to LLMs and domain specific datasets. Different approaches of pruning metric related to RIA could lead to better results. It would be worthwhile to design more elegant ES framework so it does not get trapped in local optimal. More model-aware ES could be tried to capture the importance of different matrices.

8. Acknowledgements

We would like to use this section to thank Professor Yeh and his teaching assistant Seyed Amirhosein Mohaddesi for their assistance and guidance in building our project.

References

Blog, N. D. Exploiting ampere structured sparsity with cusparselt, 2020. URL https://developer.nvidia.com/blog/exploiting-ampere-structured-sparsity-with-cusparselt/. Accessed: 2024-06-12.

Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2): 182–197, 2002.

Frantar, E. and Alistarh, D. Sparsegpt: Massive language models can be accurately pruned in one-shot, 2023.

Hansen, N. and Ostermeier, A. Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (cma-es). *Evolutionary Computation*, 11(1):1–18, 2003.

Rechenberg, I. Evolutionsstrategie ’94. Frommann-Holzboog, 1994.

Schwefel, H.-P. Numerical optimization of computer models. John Wiley & Sons, Inc., 1981.

Sun, M., Liu, Z., Bair, A., and Kolter, J. Z. A simple and effective pruning approach for large language models, 2024.

Zhang, Y., Bai, H., Lin, H., Zhao, J., Hou, L., and Cannistraci, C. V. Plug-and-play: An efficient post-training pruning method for large language models. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=Tr0lPx9woF.