# DISP-LLM: Dimension-Independent Structural Pruning for Large Language Models

Samsung Research America

Date: 1st October, 2025
Presented by: Yash Bhisikar

# Agenda

1. Preliminaries
2. Why the Naive Approach Fails
3. SliceGPT
4. DISP-LLM: The Core Details
5. Results
6. Future Directions

# Preliminaries

# Why to Prune LLMs ?

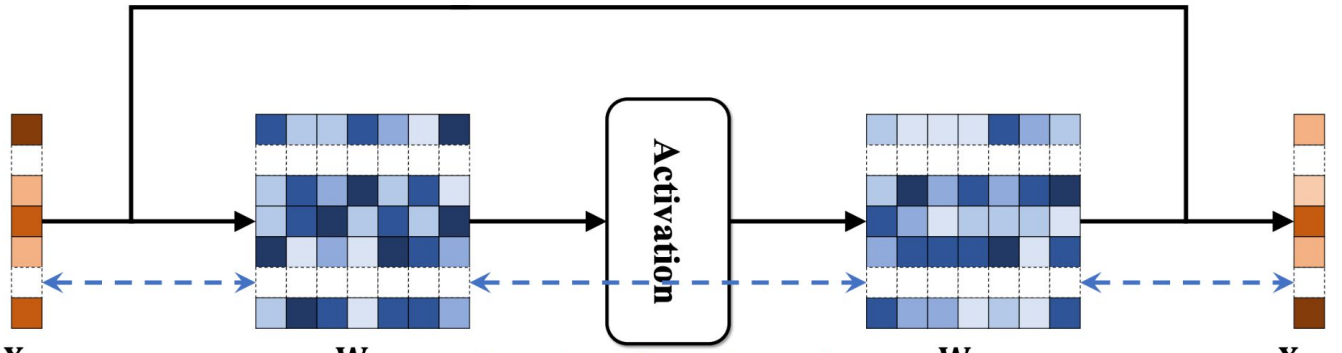
- **Reduces Inference Costs**: Pruning removes unnecessary weights, making models smaller and requiring less computational power and memory (VRAM) to run.
- **Speeds up Inference**: With fewer parameters to process, the model can generate outputs faster. This **reduces latency**
- **Deployment on Resource-Constrained Settings**: Smaller models have a reduced memory footprint, allowing them to be deployed on devices with limited resources (edge, IOT, smartphones, etc.)
- **Improves Energy Efficiency**: Less computation means lower power consumption.

Drawing on the **Lottery Ticket Hypothesis**, pruning removes redundant connections, akin to finding a smaller "winning ticket" subnetwork within the larger, unpruned model. This subnetwork can achieve comparable or even slightly better performance by eliminating less critical, "noisy" connections, leading to a significant reduction in model size with minimal to no loss.

| Feature | Structured Pruning | Unstructured Pruning |
|---|---|---|
| **What It Prunes** | Entire groups of parameters (e.g., whole neurons, channels, or attention heads). | Individual weights. |
| **Sparsity Pattern** | Regular and dense-like. | Irregular, highly sparse. |
| **Inference Speed** | Excellent speedups on any standard hardware, as the pruned model is smaller and still dense. | Poor speedups on standard hardware. Requires specialized software or hardware. |
| **Accuracy** | Can have a more significant drop in accuracy at high sparsity levels because it removes whole components, which may contain important weights. | Highest for a given sparsity level, as it can remove the most insignificant weights. |
| **Examples** | **LLM-Pruner**: A framework for structured pruning that removes attention heads and FFN layers.<br><br>**DISP-LLM**: A method that increases the flexibility of structured pruning. | **SparseGPT**: A one-shot, unstructured method that prunes by minimizing a layer-wise approximation error.<br><br>**Wanda**: Prunes weights by a criterion combining weight magnitude and input activations. |

# Why the Naive Approach Fails

# Residual connections create structural dependency along the embedding dimension




**Regular Structural Pruning Methods with Structural Dependency**

# The Problem in Action - I

**Goal**: Allow each layer to have its own selection matrix $S_L$ along embedding dimension, allowing each layer to prune differently

**Problem**: Residual connections require alignment between consecutive layers

**Setup:**

1. The model has a small embedding dimension of **d = 8**. A feature map X passing through the network would be a vector of 8 values: [x1, x2, x3, x4, x5, x6, x7, x8]
2. For layer L, we decide to keep the first 4 features: $S_L$ = ***diag***([1,1,1,1,0,0,0,0])
3. For layer L+1, we decide to keep the features 3-6: $S_{L+1}$ = ***diag***([0,0,1,1,1,1,0,0])
4. The width of each layer is the number of non-zero entries: ***nnz***($S_L$) = ***nnz***($S_{L+1}$) = 4

# The Problem in Action - II

1. **Input arrives from Layer L**: This vector is now aligned to Layer L's mask. Residual Path = [x1, x2, x3, x4, 0, 0, 0, 0]
2. **Block Path for Layer L + 1:** This block needs to be pruned by its own mask, which is aligned to [0, 0, 1, 1, 1, 1, 0, 0]
3. **The Conflict**: The residual connection for Layer L+1 is:

$$X_{L+1} = X_L + \text{Block}^{L+1}(X_L)$$

Here's the mismatch:

- The left side of the addition, $X_L$, is a vector living in the subspace defined by mask $S_L$.
- The right side of the addition, $\text{Block}^{L+1}(X_L)$, is a vector living in the subspace defined by mask $S_{L+1}$.

$$S_L^T @ S_{L+1} = \textbf{\textit{diag}}([1\times0,1\times0,1\times1,1\times1,0\times1,0\times1,0\times0,0\times0])$$

$$S_L^T @ S_{L+1} = \textbf{\textit{diag}}([0,0,1,1,0,0,0,0])$$

This is basically proposition 1 from the paper!

$$\textbf{\textit{nnz}}(S_L^T @ S_{L+1}) \leq \min\{\textbf{\textit{nnz}}(S_L), \textbf{\textit{nnz}}(S_{L+1})\}$$

# The Problem in Action - III

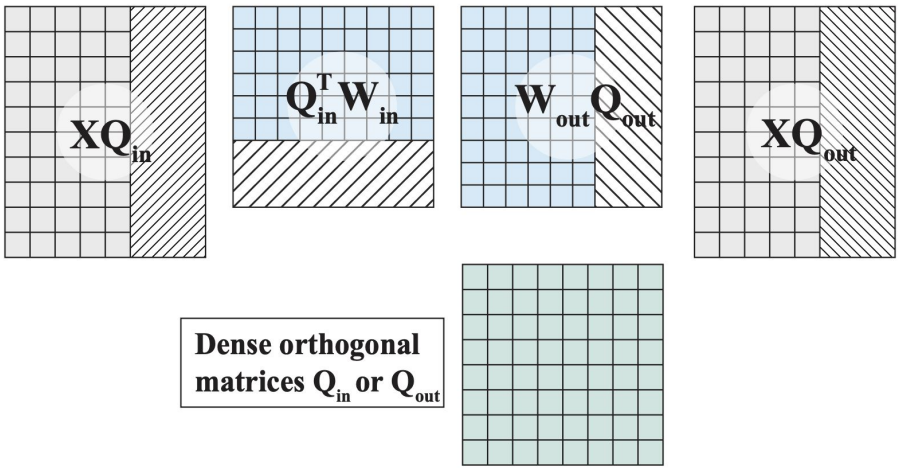
**Key Takeaways**

- If we naively apply $S_L$ for different layers, the model width **will progressively decrease** as we go deeper into the network.
- It also fails to provide different sets of features for different layers; instead, it merely **passes a subset of features** from the previous layer to the next.
- To avoid this restriction, all blocks **must share the same width** and the same pruned columns or rows.
- To enhance flexibility along the embedding dimension, bypassing the residual connections is crucial.

# SliceGPT

# Overview

- We want to insert a change in the middle of the network without altering the final output
- SliceGPT finds the most important features flowing between transformer blocks and gets a specific rotation matrix ($Q_L$) for each block-to-block connection.




SliceGPT

# Procedure - I

The data flow from the output of **Block L** to the input of **Block L+1**.

- **Start with the Output:** We begin with $X_L$, the output activation of Block L. This is a matrix of size [*sequence_length*×*d_model*].
- **Rotate the Activations:** We apply an orthogonal transformation (a rotation) to these activations using a matrix $Q_L$. This $Q_L$ is calculated using PCA on the activations $X_L$ from a calibration dataset. The new, rotated activations are $X_L' = X_L Q_L$.
- **Slice (Prune) the Rotated Activations:** Now that the important information is concentrated in the first few columns, we can safely discard the rest. Our final, compressed activation is $\tilde{X}_L = (X_L Q_L)\hat{S}$. This is the tensor that will be fed into the next block, Block L+1.

# Procedure - II

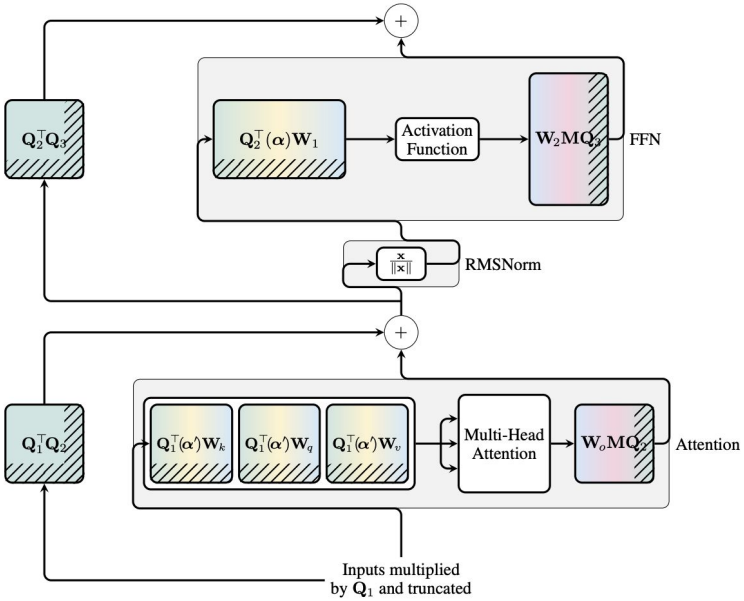
- **Invert the Transformation on the Next Layer's Weights:** We have changed the activations from $X_L$ to $\tilde{X}_L$. To ensure the model's output doesn't change, we must apply an inverse transformation to whatever consumes these activations.
    - The first thing in Block L+1 that processes $\tilde{X}_L$ is a weight matrix, let's call it $W_{L+1}$
    - The original computation was $X_L W_{L+1}$. Our new computation must be approximately equal: $\tilde{X}_L \tilde{W}_{L+1} \approx X_L W_{L+1}$.
    - To achieve this, we modify the weights: $\tilde{W}_{L+1} = \hat{S}^T Q_L^T W_{L+1}$.
- The math checks out!

$$\tilde{X}_L \tilde{W}_{L+1} = (X_L Q_L \hat{S})(\hat{S}^T Q_L^T W_{L+1}) \approx X_l (Q_L Q_L^T) W_{L+1} = X_L (I) W_{L+1} = X_L W_{L+1}$$

- **The Problem**: The input to the main path of Block L+1 is the transformed activation. However, the residual connection is still carrying the original, untransformed $X_L$. We cannot add these two together because they are in different coordinate systems!

# Procedure - III

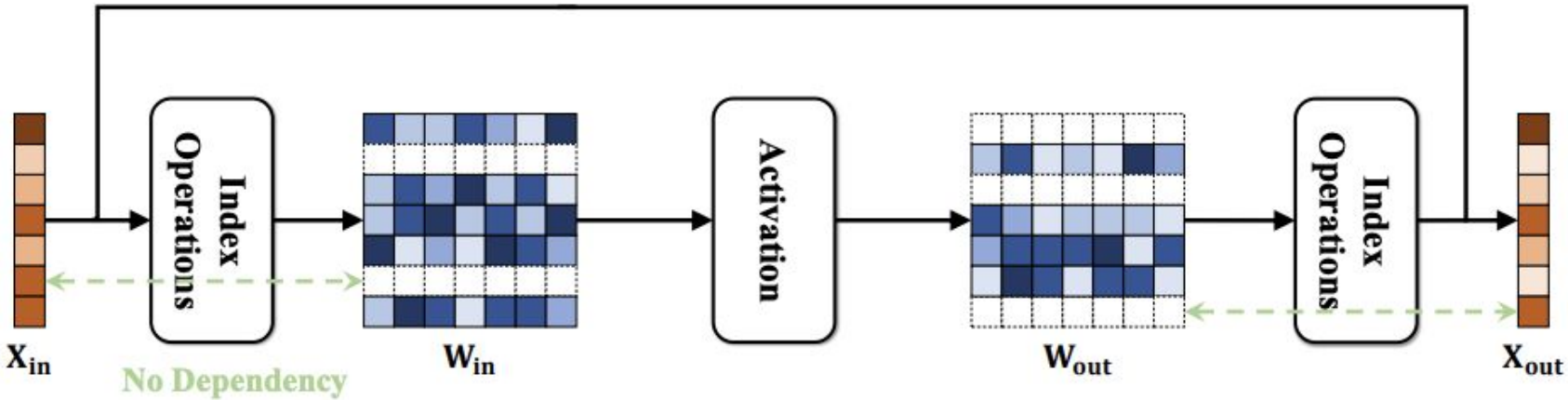
- SliceGPT must also **transform the residual connection** so it "matches" the coordinate system of the main path. Achieved by multiplying the residual connection by a linear transformation, $Q_L^T Q_{L+1}$, before it can be correctly processed within block L+1.
- The drawback is that this matrix $T = Q_L^T Q_{L+1}$ is a dense d×d matrix. You need one such transformation matrix for every block in the model. Performing this matrix multiplication $X_L \cdot T$ for every token at every residual connection adds a significant number of computations (**~ 5-13 % of entire parameter count**)



# DISP-LLM: The Core Details

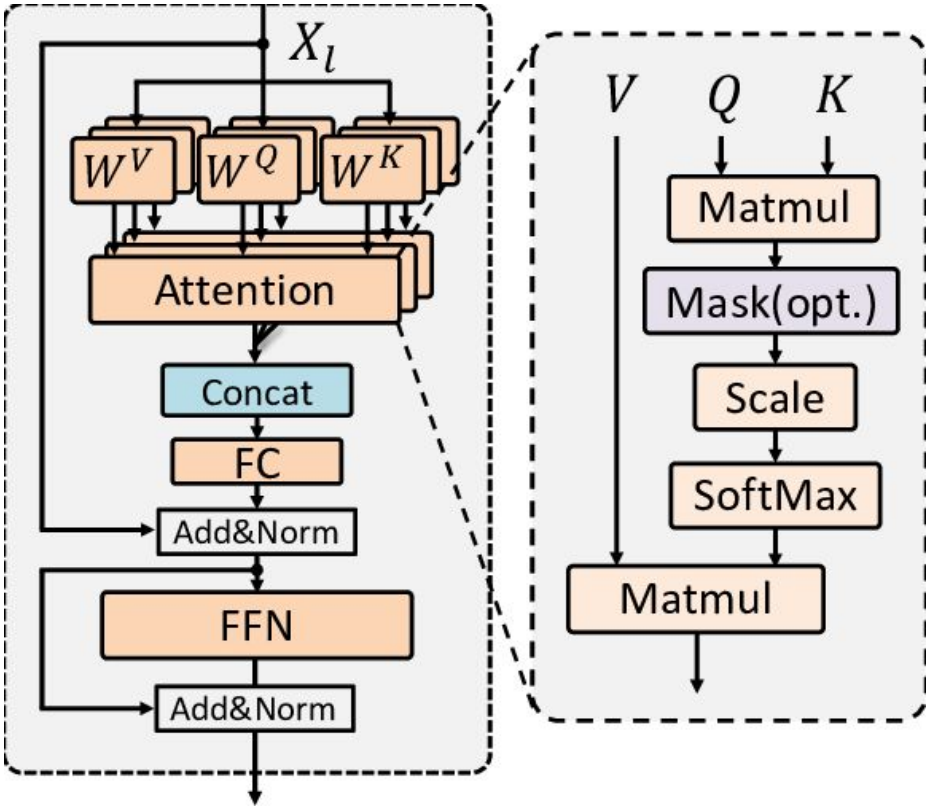
# Key Insight:  Breaking Structural Dependency

- In methods like LLM-Pruner, if you prune a specific channel in one layer, you *must prune* **that same channel in all subsequent layers** (re: corollary 1) because the residual connection adds the input directly to the output.
- SliceGPT is able to avoid this **but adds overhead with the residual transformations**. It also does not require any post-pruning fine tuning
- DISP-LLM is able to achieve flexibility in pruning without adding any extra parameters to the model itself.
- How? Avoid altering the residual connection itself. It performs selection and merging operations inside each block's main computational path. This is achieved through two new operations: **Index Select** and **Index Add**.




**Dimension-Independent Structural Pruning (Ours)**

# The Proposed Solution - I



# The Proposed Solution - II

$$\text{Attention}(\mathbf{X}) = \text{MultiHead}(\mathbf{X}\mathbf{S}_1^\top\mathbf{W}_q, \mathbf{X}\mathbf{S}_1^\top\mathbf{W}_k, \mathbf{X}\mathbf{S}_1^\top\mathbf{W}_v)\mathbf{W}_o\mathbf{S}_2,$$

$$\text{MLP}(\mathbf{X}) = (\sigma(\mathbf{X}\mathbf{S}_3^\top\mathbf{W}_1\mathbf{S}_4) \odot (\mathbf{X}\mathbf{S}_3^\top\mathbf{W}_2\mathbf{S}_4))\mathbf{S}_4{}^\top\mathbf{W}_3\mathbf{S}_5,$$

**Algorithm 1:** Block inference after pruning.

**Input**: Feature map of the previous block $\mathbf{X}_{\text{in}}$. Preserved indices sets $\mathbf{Ind}_1, \mathbf{Ind}_2, \mathbf{Ind}_3, \mathbf{Ind}_5$.

1. $\hat{\mathbf{X}}_{\text{in}} = \text{LayerNorm}(\mathbf{X}_{\text{in}}[:, \mathbf{Ind}_1])$. ▷ *Index Selection for Attention*
2. $\mathbf{X}_{\text{att}} = \text{MultiHead}(\hat{\mathbf{X}}_{\text{in}}\hat{\mathbf{S}}_1^\top\mathbf{W}_q, \hat{\mathbf{X}}_{\text{in}}\hat{\mathbf{S}}_1^\top\mathbf{W}_k, \hat{\mathbf{X}}_{\text{in}}\hat{\mathbf{S}}_1^\top\mathbf{W}_v)\mathbf{W}_o\hat{\mathbf{S}}_2$.
3. $\mathbf{X}_{\text{res}} = \text{Index\_Add}(\mathbf{X}_{\text{in}}, \mathbf{X}_{\text{attn}}, \mathbf{Ind}_2)$. ▷ *Index Addition with the input*
4. $\hat{\mathbf{X}}_{\text{res}} = \text{LayerNorm}(\mathbf{X}_{\text{res}}[:, \mathbf{Ind}_3])$. ▷ *Index selection for MLP*
5. $\mathbf{X}_{\text{mlp}} = (\sigma(\hat{\mathbf{X}}_{\text{res}}\hat{\mathbf{S}}_3^\top\mathbf{W}_1\hat{\mathbf{S}}_4) \odot (\hat{\mathbf{X}}_{\text{res}}\hat{\mathbf{S}}_3^\top\mathbf{W}_2\hat{\mathbf{S}}_4))\hat{\mathbf{S}}_4{}^\top\mathbf{W}_3\hat{\mathbf{S}}_5$.
6. $\mathbf{X}_{\text{out}} = \text{Index\_Add}(\mathbf{X}_{\text{res}}, \mathbf{X}_{\text{mlp}}, \mathbf{Ind}_5)$ ▷ *Index Addition with the residual*

**Return** $\mathbf{X}_{\text{out}}$ for the next block.

# A Step-by-Step Example I: The Setup

- Let the model's embedding dimension be: d = 4
- The input to our block is $X_{in} \in R^{n \times 4}$, a single input vector could be $x_{in}=[x_1, x_2, x_3, x_4]$
- Let's say this block's attention layer decides to use input dimensions {0, 2} and produce an output that affects dimensions {0, 2}. So, its index sets are *Ind1*={0, 2} and *Ind2* = {0, 2}
- The subsequent MLP layer decides to use input dimensions {1, 2, 3} and produce an output that affects dimensions {0, 1, 3}. Its index sets are *Ind3* = {1, 2, 3} and *Ind5* = {0, 1, 3}
- Assume the set of indices (*Ind1, Ind2*, etc.) are given to us. We'll go into details about how to select them.

# A Step-by-Step Example II: Attention Calculation

- Before attention calculation, we **select a subset of features** from the input $X_{in}$
- $X_{attn\_in} = X_{in}[:, Ind1] = [x1, x3]$. We take the columns specified by $Ind1 = \{0, 2\}$ from $X_{in}$. The attention mechanism now operates on this smaller feature space.
- The weight matrices ($W_q$, $W_k$, $W_v$, $W_o$) have been permanently pruned beforehand to match these smaller dimensions. The original $W_q \in R^{4 \times d\_head}$ becomes a pruned $\tilde{W}_q \in R^{2 \times d\_head}$ by keeping only the rows corr. to *Ind1*, output weights $\tilde{W}_o$ pruned to have columns corr. to *Ind2*
- $X_{attn\_out} = \textbf{\textit{Attention}}(X_{attn\_in}) = [a1, a2]$
- Now, we add this low-dimensional output back to the original, full-dimensional input only at the indices specified by $Ind2 = \{0, 2\}$. $X_{res} = \textbf{\textit{Index\_Add}}(X_{in}, X_{attn\_out}, Ind2) = [x1 + a1, x2, x3 + a2, x4]$

# A Step-by-Step Example III: MLP Sub-block

- Process repeats for the MLP, use the output of the attention sub-block, $X_{res}$ as input.
- We use *Ind3* = {1, 2, 3} => $X_{mlp\_in} = X_{res}[: Ind3] = [x2, x3 + a2, x4]$
- The MLP's weight matrices are also pre-pruned to match the dimensions defined by *Ind3*, *Ind4* (intermediate), and *Ind5*

$$X_{mlp\_out} = \textbf{\textit{MLP}}(X_{mlp\_in}) = [m1, m2, m3]$$

- ***IndexAdd*** for MLP Residual: We have $X_{res} = [x1 + a1, x2, x3 + a2, x4]$ and *Ind5* = {0,1,3}

$$X_{out} = [x1 + a1 + m1, x2 + m2, x3 + a2, x4 + m3]$$

- This final $X_{out}$ is passed to the next block, still in the full d=4 dimension, ready for the next block to select its own **independent set** of indices. **This is how the structural dependence is broken.**

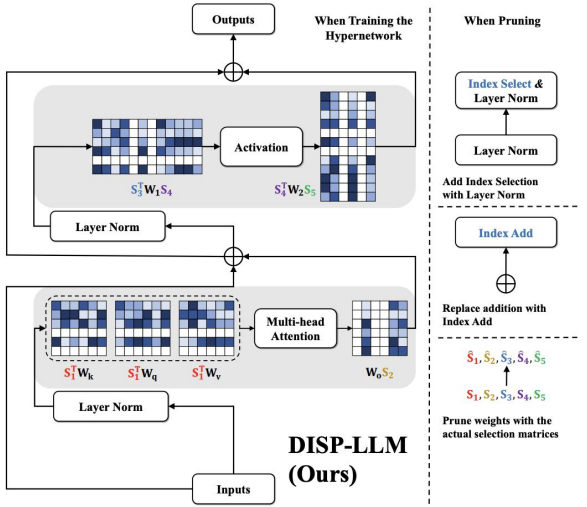



Figure 2: Our method, DISP-LLM, applies different selection matrices to the input and output dimension of the Attention layer and MLP layer ($S_1/S_2$: Attention in/out; $S_3/S_4/S_5$: MLP in/middle/out). When pruning the model, we add "Index Selection" before Layer Norm and we replace addition with "Index Add." $\hat{S}_1, \cdots, \hat{S}_5$ are applied for pruning weight matrices.

# Learning Which Indices to Prune - I

- How do we find the optimal index sets (*Ind1* to *Ind5*) for every layer? The search space is enormous
- **Problem**: Choosing to keep or discard an index is a binary decision, which is non-differentiable. You *can't* use standard backpropagation to learn it.
- **Proposed Solution**: Use a **gradient estimator**. The paper uses **ReinMax**, which is a technique that allows gradients to be estimated and passed through discrete, binary operations. This essentially makes the non-differentiable selection process "trainable."
- Instead of learning separate parameters for every index in every layer, the authors use a small hypernetwork (composed of a GRU and linear layers) **to generate the selection parameters** for the entire model. This is allows the network to learn relationships between the pruning decisions of different layers

# Learning Which Indices to Prune - II

**Algorithm 2:** Binary ReinMax

**Input**: $x$: sigmoid input;
$\tau$: temperature; $c$: constant bias.
**Output**: $\mathbf{x}$: binary vector.

1. $\pi_0 = \text{sigmoid}(x + c)$,
2. $B = \text{sample\_binary}(\pi_0)$,
3. $\pi_1 = \frac{B + \text{sigmoid}((x+c)/\tau)}{2}$,
4. $\pi_1 = \text{sigmoid}(\text{stop\_gradient}(\ln(\pi_1) - (x + c)) + (x + c))$,
5. $\pi_2 = 2\pi_1 - \frac{1}{2}\pi_0$,
6. $\mathbf{x} = \pi_2 - \text{stop\_gradient}(\pi_2) + B$

**Return** $\mathbf{x}$.

# Learning Which Indices to Prune - III

The hypernetwork is trained to minimize a combined objective function

$$\min_{\theta} \mathcal{L}(\mathcal{X}; W, s) + \lambda \mathcal{R}(T(s), pT_{\text{total}})$$

$$\mathcal{R}(T(\mathbf{s}), pT_{\text{total}}) = \log(\max(T(\mathbf{s}), pT_{\text{total}}) / \min(T(\mathbf{s}), pT_{\text{total}}))$$

where:

1. **L** is the standard language modeling loss (predicting the next word). This ensures the pruned model remains accurate
2. **s** represents the binary selection masks generated by the hypernetwork.
3. **R** is a regularization loss that penalizes the model if its current parameter count, **T(s)**, is different from the target parameter count, $\mathbf{pT_{total}}$ . This pushes the model toward the desired sparsity level
4. **Θ** represents the weights of the hypernetwork that we are optimizing

# SliceGPT vs DISP-LLM

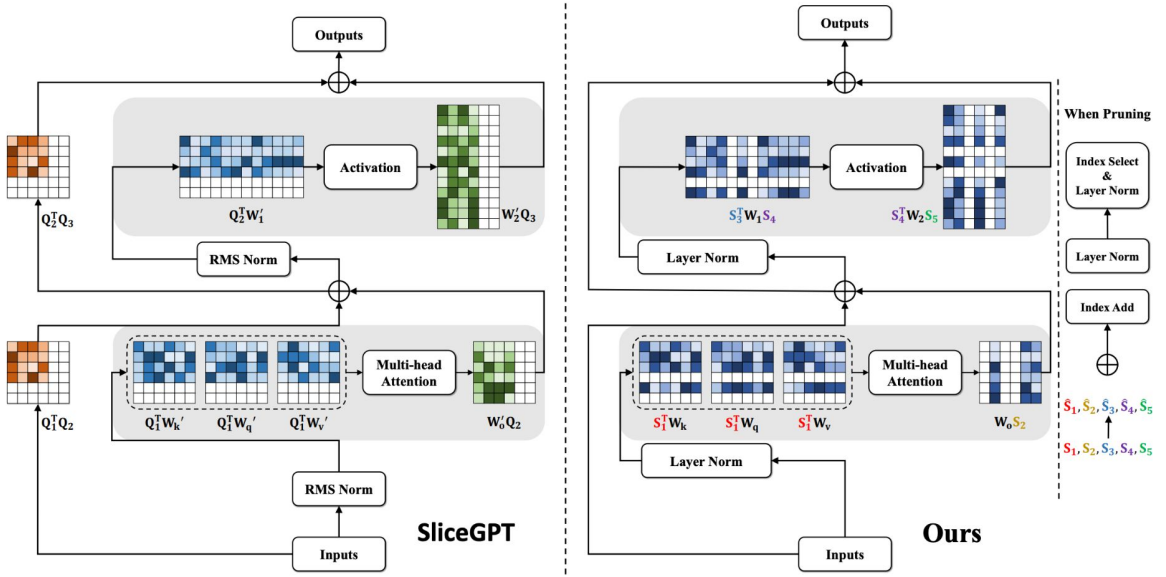



Figure 7: **Left:** SliceGPT inserts $\mathbf{Q}_l^\top \mathbf{Q}_{l+1}$ to the residual connection and brings additional parameters. It also modifies the weights and Layer Norms within the original model. The selection matrix $\mathbf{S}$ is omitted for consistency. **Right:** Our method, DISP-LLM, applies different selection matrices to the input and output dimension of the Attention layer and MLP layer ($\mathbf{S}_1/\mathbf{S}_2$: Attention in/out; $\mathbf{S}_3/\mathbf{S}_4/\mathbf{S}_5$: MLP in/middle/out).

# Results

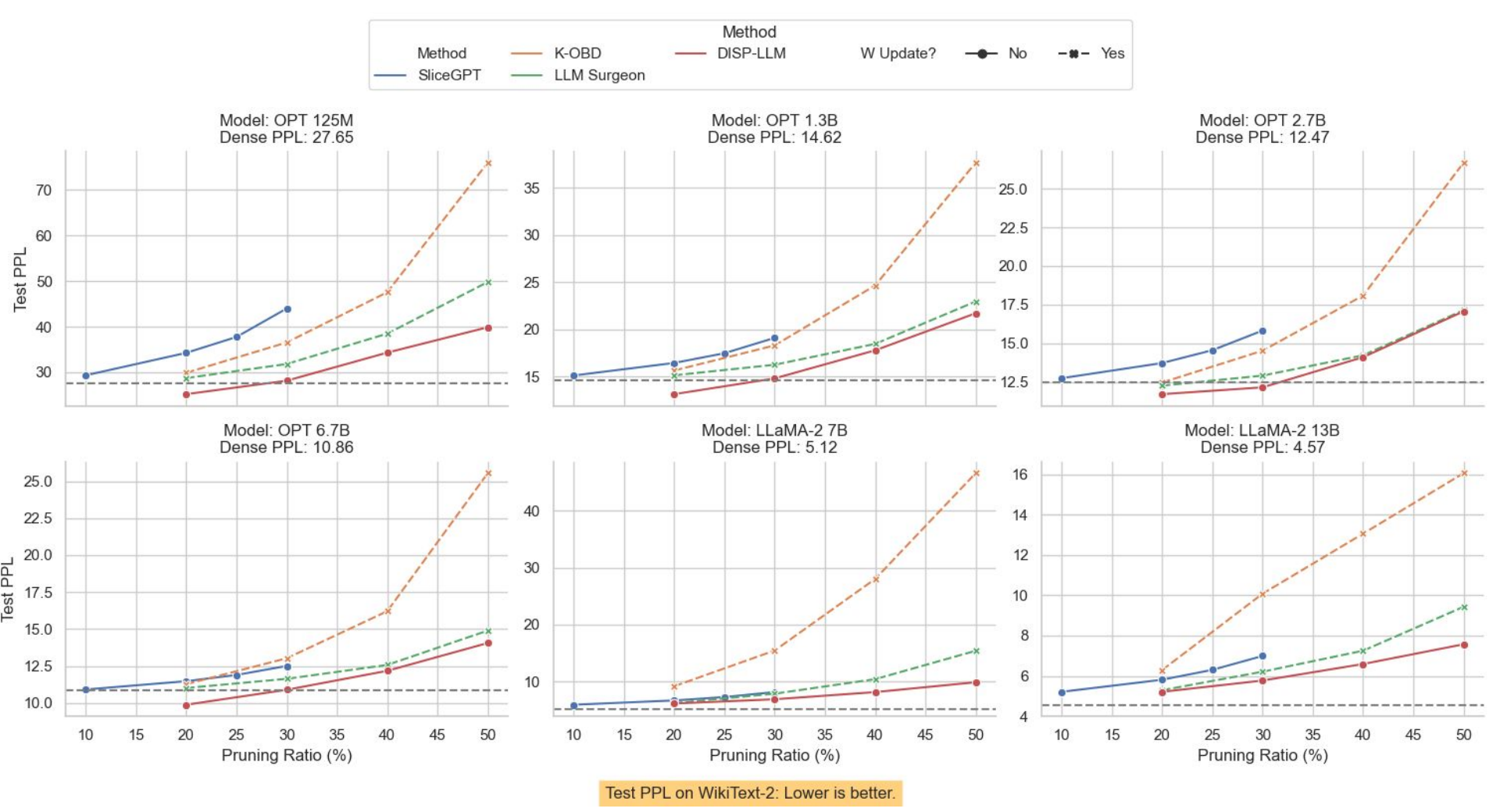
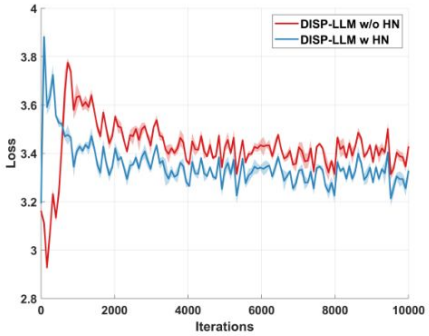

Method
Method
SliceGPT
K-OBD
LLM Surgeon
DISP-LLM
W Update?
No
Yes
Model: OPT 125M
Dense PPL: 27.65
Model: OPT 1.3B
Dense PPL: 14.62
Model: OPT 2.7B
Dense PPL: 12.47
Model: OPT 6.7B
Dense PPL: 10.86
Model: LLaMA-2 7B
Dense PPL: 5.12
Model: LLaMA-2 13B
Dense PPL: 4.57
Test PPL
Pruning Ratio (%)
Test PPL on WikiText-2: Lower is better.

Table 2: Comparison of our method against semi-structure pruning methods on WikiText-2.

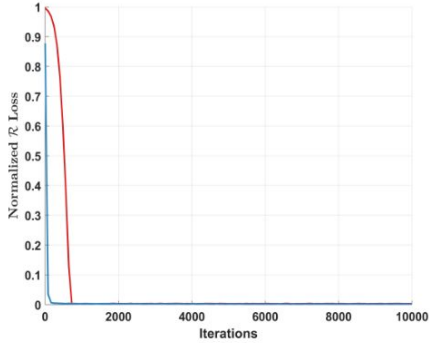
| Method | Pruning Ratio | **W** Update? | Structure? | Test Performance (PPL) | | | |
|---|---|---|---|---|---|---|---|
| | | | | LLaMA 7B | LLaMA 13B | LLaMA-2 7B | LLaMA-2 13B |
| Dense | 0% | - | - | 5.68 | 5.09 | 5.12 | 4.57 |
| Magnitude | 2:4 | ✗ | ✗ | 42.13 | 18.37 | 54.59 | 8.33 |
| SparseGPT [9] | 2:4 | ✓ | ✗ | **11.00** | 9.11 | 10.17 | 8.32 |
| Wanda [34] | 2:4 | ✗ | ✗ | 11.53 | 9.58 | 11.02 | 8.27 |
| DISP-LLM (ours) | 50% | ✗ | ✓ | 11.47 | **8.15** | **9.84** | **7.11** |

Table 3: Zero-shot performance of the compressed LLaMA 7B, LLaMA-2 7B and Phi models. The structure of *DISP-LLM* is based on the WikiText dataset, and the structure of *DISP-LLM Alpaca* is based on the Alpaca dataset.

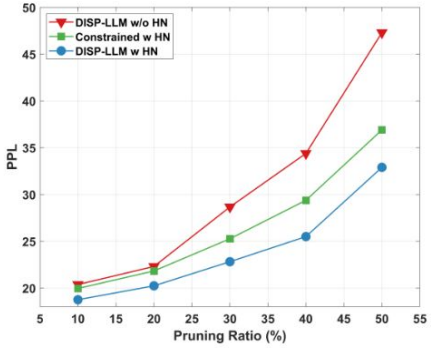
| Pruning Ratio | Method | W Update? | WinoGrande | HellaSwag | ARC-e | ARC-c | PIQA | Avg |
|---|---|---|---|---|---|---|---|---|
| | | | acc | acc-norm | acc-norm | acc-norm | acc-norm | |
| 0% | LLaMA 7B | - | 69.85 | 76.21 | 72.81 | 44.71 | 79.16 | 68.55 |
| 20% | LLM-Pruner [30] | ✗ | 61.33 | 65.34 | 59.18 | 37.12 | 75.57 | 59.71 |
| | +finetuning | ✓ | 65.11 | 68.11 | 63.43 | **37.88** | 76.44 | 62.19 |
| | DISP-LLM (Ours) | ✗ | **66.54** | **68.75** | 59.60 | 35.24 | 74.97 | 61.02 |
| | DISP-LLM Alpaca (Ours) | ✗ | 64.72 | 68.39 | **64.81** | 37.12 | **76.66** | **62.34** |
| 50% | LLM-Pruner [30] | ✗ | 53.20 | 35.64 | 33.50 | 27.22 | 59.63 | 41.84 |
| | +finetuning | ✓ | 55.09 | 47.56 | 46.46 | 28.24 | **68.82** | 49.23 |
| | DISP-LLM (Ours) | ✗ | **58.41** | 47.71 | 44.40 | 28.50 | 64.09 | 48.62 |
| | DISP-LLM Alpaca (Ours) | ✗ | 56.91 | **48.76** | **48.91** | **31.57** | 67.46 | **50.72** |
| 0% | LLaMA-2 7B | - | 69.14 | 75.99 | 74.58 | 46.15 | 79.11 | 68.99 |
| 30% | SliceGPT [2] | ✗ | 61.33 | 49.62 | 51.77 | 31.23 | 63.55 | 51.50 |
| | K-OBD [34] | ✓ | 56.83 | 53.07 | 51.05 | 33.11 | 71.82 | 53.18 |
| | LLM Surgeon [34] | ✓ | 61.09 | 60.72 | **63.09** | 36.69 | 73.56 | 59.03 |
| | DISP-LLM (Ours) | ✗ | 62.27 | **63.43** | 59.81 | 33.19 | 71.82 | 58.10 |
| | DISP-LLM Alpaca (Ours) | ✗ | **63.93** | 62.87 | 60.10 | **37.03** | **73.72** | **59.53** |
| 50% | K-OBD [34] | ✓ | 53.04 | 36.84 | 36.11 | 26.71 | 60.66 | 42.67 |
| | LLM Surgeon [34] | ✓ | 52.57 | 40.29 | 44.91 | 26.28 | 64.36 | 45.68 |
| | DISP-LLM (Ours) | ✗ | 54.54 | 46.33 | 43.06 | 25.85 | 63.93 | 46.72 |
| | DISP-LLM Alpaca (Ours) | ✗ | **56.20** | **49.35** | **51.14** | **30.20** | **68.34** | **51.05** |
| 0% | Phi-1.5 | - | 72.77 | 62.58 | 73.11 | 48.04 | 75.63 | 66.43 |
| 30% | SliceGPT [2] | ✗ | **64.96** | 42.54 | 53.66 | 31.91 | 65.45 | 51.52 |
| | DISP-LLM (Ours) | ✗ | 61.48 | **47.97** | **57.66** | **33.01** | **71.08** | **54.24** |
| 0% | Phi-2 | - | 75.61 | 73.86 | 78.24 | 54.01 | 79.11 | 72.17 |
| 30% | SliceGPT [2] | ✗ | 63.14 | 47.56 | 53.03 | 30.29 | 65.94 | 51.99 |
| | DISP-LLM (Ours) | ✗ | **65.19** | **54.43** | **63.59** | **38.48** | **73.34** | **59.00** |



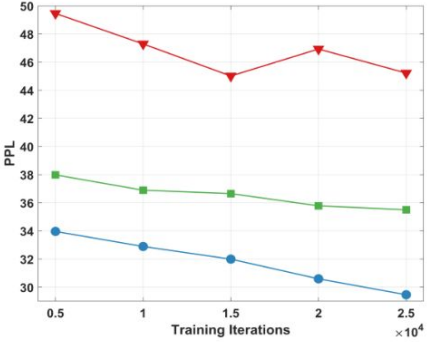

(a) Ablation Loss $\mathcal{L}$



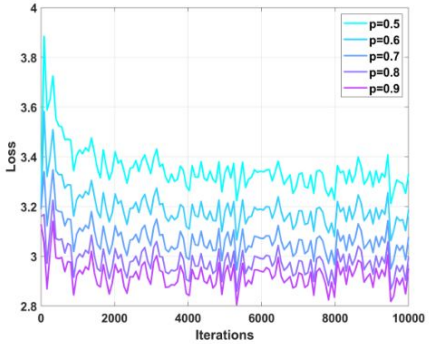

(b) Ablation Loss $\mathcal{R}$



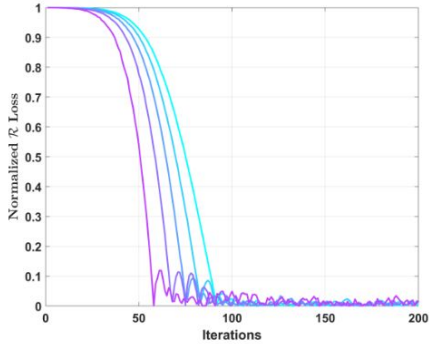

(c) Ablation $p$



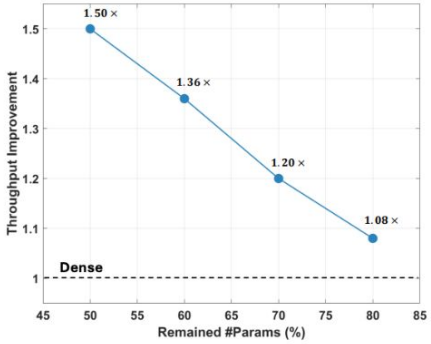

(d) Ablation Iterations



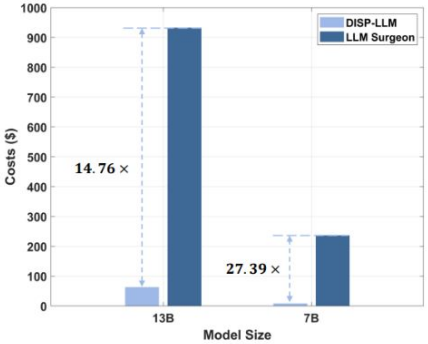

(e) Given $p$ Loss $\mathcal{L}$




(f) Given $p$ Loss $\mathcal{R}$




(g) Acceleration




(h) Costs

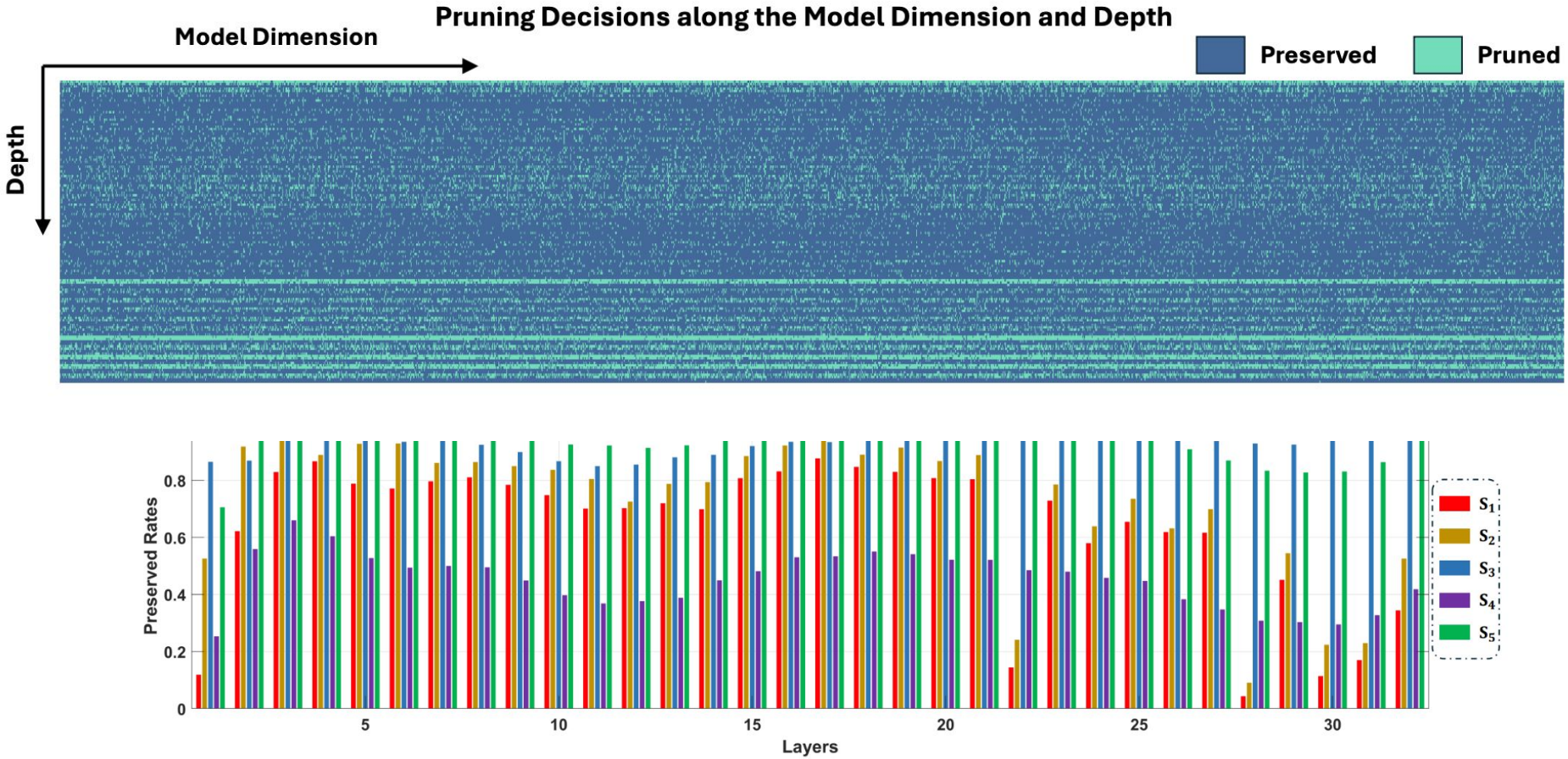



Figure 6: Model width after pruning for the LLaMA-2 7B model when the pruning ratio equals 50%.

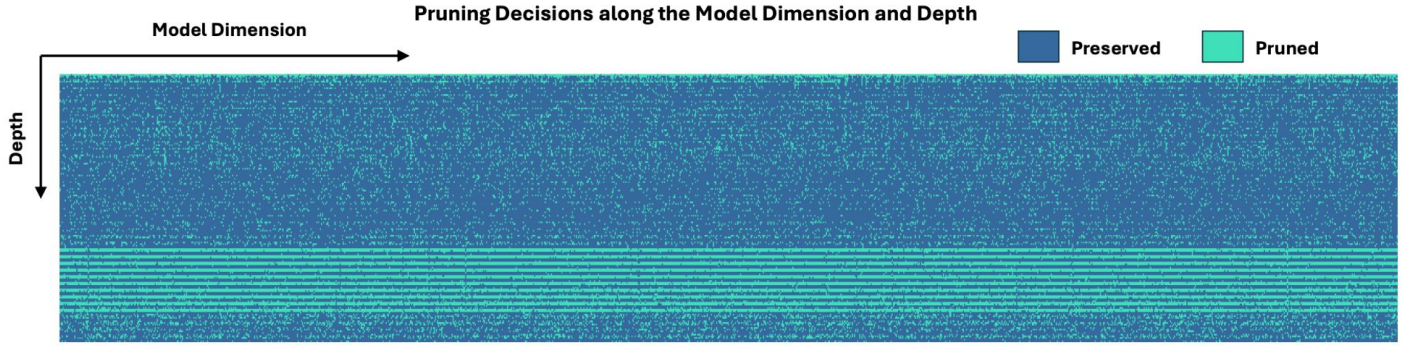



Figure 8: The pruned model architecture along the embedding dimension (model dimension) for the LLaMA-2 13B model when the pruning ratio equals 50%.

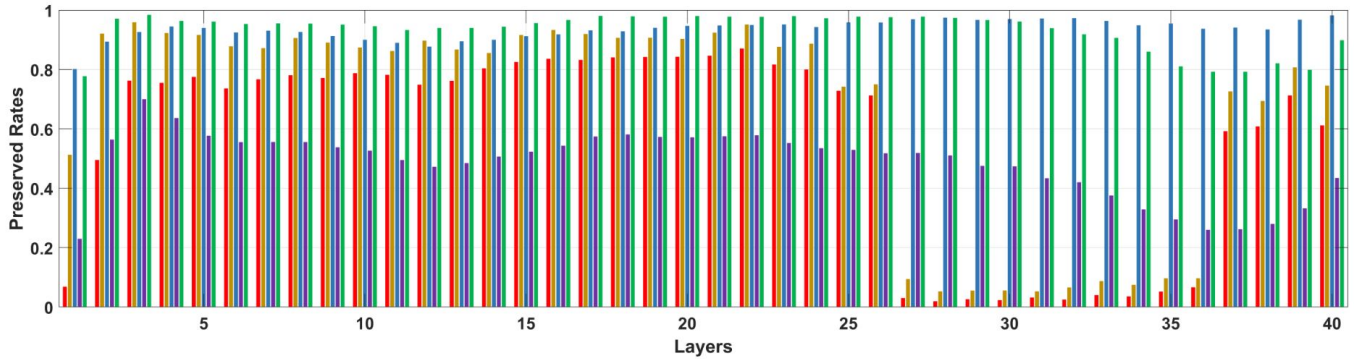



Figure 9: Model width after pruning for the LLaMA-2 13B model when the pruning ratio equals 50%.

# Future Directions

- The throughput improvements from the method are not consistent across all models. Authors hypothesize that this is because the standard PyTorch implementations for index selection and addition operations are not fully optimized
- The current implementation focuses on selecting from feature maps, which involves indexing overhead during every forward pass. An alternative, and potentially faster, approach would be to **pre-slice the weight matrices** and use kernels that take advantage of zeroed rows/columns.
- The hypernetwork itself could become a bottleneck in the search process for extremely large models.

# Thank you + Q&A