Optimal Block Sparse Attention Mask for Faster LLM Inference

Shiji Xin
shijixin@g.harvard.edu

Melissa Rosenberg
mel_rsbg@mit.edu

Abstract

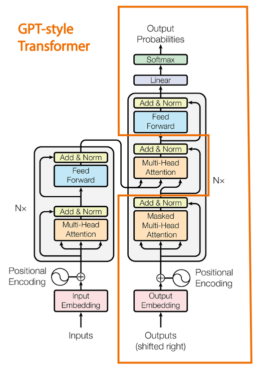
Large language models (LLMs) require significant computational resources. To reduce the cost of inference, we optimize block-sparse masks for the attention mechanism in the Transformer architecture, which avoid calculating specific sub-matrices within the matrix multiplication of the attention operation. Framing the problem as a mixed integer optimization, we achieve up to 75% sparsity with negligible performance loss compared to the default causal mask (43.75% sparsity). Experiments on a GPT-like model show that optimized masks outperform heuristic methods, maintaining high performance with reduced computational costs. This demonstrates the potential of block-sparse attention for efficient LLMs.

1 Introduction

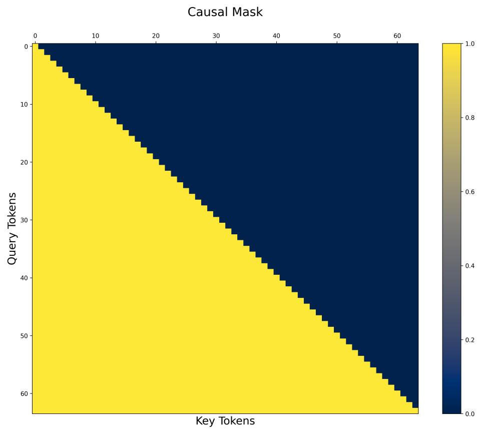
Large language models (LLMs) are foundational to a wide range of modern applications; however, training and deploying these models requires significant GPU resources. Our project aims to enhance the efficiency of a critical component of LLMs during inference. Most contemporary LLMs are built upon the Transformer architecture (Fig. 1a), where the attention mechanism (Eq. 1) plays a pivotal role. This mechanism involves matrix multiplication and a Softmax function, typically applied with a mask to selectively exclude parts of the $QK^\top$ matrix, with causal mask (Fig. 1b as default. Optimizing this operation is essential to reduce computational overhead and improve model performance during inference.

$$\text{Attention}(Q, K, V) = \text{Softmax}\left(\text{Mask}\left(\frac{QK^\top}{\sqrt{d_k}}\right)\right)V \tag{1}$$

Our project focuses on improving the efficiency of the attention operation without sacrificing the performance, with efficiency defined by the latency of generating a new token from the language model. A common optimization technique involves partitioning the $QK^\top$ calculation into smaller blocks and applying a block-sparse mask. For example, a 64×64 matrix can be divided into 64 sub-matrices of size 8×8, where only specific sub-matrices are computed. This approach enables the masked result to be obtained without performing unnecessary computations, significantly reducing runtime. Previous studies have explored heuristic methods for designing block-sparse masks [1, 2, 3]. Nonetheless, these improvements may fall short when handling requests involving longer contexts, which result in more complex patterns. To address this limitation, our project formalizes the design of block-sparse masks using optimization techniques. Specifically, we frame the problem as a Mixed Integer Optimization (MIO) task and solve it using Gurobi. This approach achieves up to 75% sparsity with negligible quality degradation, a significant improvement over the default 43.75% sparsity of the traditional causal mask (Fig. 1b) while having much better quality compared to masks constructed through heuristics.



(a) Transformer architecture of GPT models



(b) Causal Mask

2 Problem formulation

We formulate the problem as minimizing the difference between the output of the attention operation using a block-sparse mask and the original attention output using the causal mask. Mathematically, this can be expressed as:

$$\min_{\mathbf{S}} \sum_{i\in[d]} \left\| \text{Softmax}\left(\mathbf{S} + \text{CausalMask}\left(\frac{Q_i K_i^\top}{\sqrt{d_k}}\right)\right) V_i - \text{Softmax}\left(\text{CausalMask}\left(\frac{Q_i K_i^\top}{\sqrt{d_k}}\right)\right) V_i \right\|_F^2 \tag{2}$$

$$\text{s.t.} \quad \mathbf{S} \text{ is block sparse}, \mathbf{S}_{ij} \in \{-\infty, 0\} \tag{3}$$

We minimize the Frobenius norm of the difference between two outputs: The first term represents the attention result when using a block-sparse mask $\mathbf{S}$. The second term is the reference attention result, computed using the full causal mask. The summation over $i \in [d]$ accounts for all examples in the training dataset of size d. d_k is the constant feature size used as normalization factor.

The Softmax function is applied row-wise to normalize attention scores. For a vector $(x_1, \ldots, x_n)$, the Softmax is defined as:

$$\text{Softmax}(x_1, \ldots, x_n) = \left(\frac{e^{x_1}}{\sum_i e^{x_i}}, \ldots, \frac{e^{x_n}}{\sum_i e^{x_i}}\right) \tag{4}$$

The causal mask selectively excludes attention to future tokens, maintaining autoregressive behavior. For a matrix A, the causal mask is defined as:

$$\text{CausalMask}(A) = \begin{bmatrix} -\infty & -\infty & \cdots & -\infty & -\infty \\ 0 & -\infty & \cdots & -\infty & -\infty \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & -\infty & -\infty \\ 0 & 0 & \cdots & 0 & -\infty \end{bmatrix} + A \tag{5}$$

where the $-\infty$ terms will be masked out to 0 after Softmax.

$\mathbf{S}$ is the block-sparse mask we aim to optimize. It reduces the number of computed entries in the attention matrix while preserving performance. In this project, we define $n \times n$ block sparse matrix $\mathbf{S}$ as:

$$\mathbf{S} = (\mathbf{1} - Z) \otimes N_k,$$

where:

1. Z is a binary matrix $(Z_{ij} \in \{0, 1\}, \forall i, j)$ that determines which blocks of S are active (nonzero).

2. N_k is a $k \times k$ matrix filled with negative infinity:

$$N_k = \begin{bmatrix} -\infty & -\infty & \cdots & -\infty \\ -\infty & -\infty & \cdots & -\infty \\ \vdots & \vdots & \ddots & \vdots \\ -\infty & -\infty & \cdots & -\infty \end{bmatrix}_{k \times k}$$

3. $\otimes$ is the Kronecker product, which expands Z by replacing each entry Z_{ij} with $(1-Z_{ij}) \cdot N_k$.
4. $\sum_{i,j} Z_{ij} \leq s$, where s is a constant controlling the sparsity of S.

This block-sparse formulation enables us to compute only the sub-matrices corresponding to nonzero entries in Z, reducing the number of calculations required to produce the masked attention result.

Direct optimization of the original objective is infeasible because the Softmax function introduces nonlinearity. To address this, we leverage the fact that the Softmax operation consists of two steps:

1. Exponential transformation: Applying the exponential function element-wise.
2. Row-wise normalization: Normalizing each row to ensure the entries sum to 1.

By reordering these steps, we can apply $\mathbf{S}'$,the exponential version of sparse mask $\mathbf{S}$, after the exponential transformation but before normalization. This reordering transforms the objective into:

$$\text{Softmax}\left(\mathbf{S} + \text{CausalMask}\left(\frac{Q_i K_i^\top}{\sqrt{d_k}}\right)\right) V_i = \text{Normalize}\left(\mathbf{S}' \odot \text{CausalMask}\left(\exp\left(\frac{Q_i K_i^\top}{\sqrt{d_k}}\right)\right)\right) V_i \tag{6}$$

where

$$\mathbf{S}' = \exp(\mathbf{S}) = Z \otimes J_k \tag{7}$$

$$J_k = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & 1 & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \cdots & 1 \end{bmatrix}_{k \times k} \tag{8}$$

We can see that $\mathbf{S}'$ is a block sparse $\{0, 1\}$-matrix.

$\exp(A)$ is the element-wise exponential of matrix A, defined as:

$$\exp(A) = \begin{bmatrix} e^{a_{11}} & e^{a_{12}} & \dots & e^{a_{1n}} \\ e^{a_{21}} & e^{a_{22}} & \cdots & e^{a_{2n}} \\ \vdots & \vdots & \ddots & \vdots \\ e^{a_{n1}} & e^{a_{n2}} & \cdots & e^{a_{nn}} \end{bmatrix}$$

Normalize is applied row-wise to a vector $(x_1, \dots, x_n)$, defined as:

$$\text{Normalize}\,(x_1, \dots, x_n) = \left(\frac{x_1}{\sum_i x_i}, \dots, \frac{x_n}{\sum_i x_i}\right)$$

By precomputing the exponential transformation and applying the sparse mask $\mathbf{S}'$ afterward, we eliminate the nonlinearity introduced by the exponential step. This simplifies the optimization process and allows for efficient computation of the masked attention.

To incorporate the Normalize operation, we can define row_sum variables for each row and enforce

$$x_{\text{normalized},i,j} \times \text{row\_sum}_i = x_{i,j} \times \mathbf{S}'_{i,j} = x_{i,j} \times Z_{\lceil i/k \rceil, \lceil j/k \rceil} \forall i, j \tag{9}$$

where $Z_{\lceil i/k \rceil, \lceil j/k \rceil}$ maps the block indices of $\mathbf{S}'$ to the corresponding element in Z.

A complication arises if an entire row of the attention matrix is masked out, resulting in a $\text{row\_sum}_i = 0$. In this case, the previous constraint becomes invalid. To address this, we introduce an additional binary variable, $\text{is\_non\_zero}_{\lceil i/k \rceil}$, to indicate whether any element in the row is nonzero.

We linearize the constraints to handle this edge case as follows:

1. Ensure the normalized value is zero for masked rows:
$$x_{\text{normalized},i,j} \leq \text{ is\_non\_zero }_{\lceil i/k \rceil}, \quad \forall i,j \tag{10}$$
2. Relate is_non_zero to the block sparsity:
If a block has any nonzero values, is_non_zero $_{[i/k]}$ is set to 1:
$$\sum_{j=1}^{n_{\text{mb}}} Z_{\lceil i/k \rceil, j} \geq \text{is\_non\_zero }_{\lceil i/k \rceil} \tag{11}$$
If all elements in a block are zero, is_non_zero $_{\lceil i/k \rceil}$ is set to 0:
$$\sum_{j=1}^{n_{\text{sub}}} Z_{\lceil i/k \rceil, j} \leq n_{\text{sub}} \cdot \text{ is\_non\_zero }_{\lceil i/k \rceil}. \tag{12}$$
Here: $n_{\text{sub}} = n/k$ is the number of sub-blocks in each row or column of Z, while the upper bound ensures that the is_non_zero variable only activates when at least one block is nonzero.

During our experiments, we observed that Gurobi, running on a CPU, takes a significant amount of time to perform matrix multiplication, so we simplified the objective by removing the multiplication with V in the objective, focusing solely on minimizing the difference between the attention score matrices, leading to the final objective as

$$\min_{\mathbf{S}} \sum_{i\in[d]} \left\| \text{Normalize}\left(\mathbf{S}' \odot \text{CausalMask}\left(\exp\left(\frac{Q_i K_i^\top}{\sqrt{d_k}}\right)\right)\right) - \text{Softmax}\left(\text{CausalMask}\left(\frac{Q_i K_i^\top}{\sqrt{d_k}}\right)\right)\right\|_F^2 \tag{13}$$

which gives good results in practice with much faster solving speed.

The final optimization code is shown in Alg.1 in Appendix.

3 Methodology

The problem is implemented creatively in three steps: first, extracting the weights of the standard causal mask; second, optimizing the mask and making the causal mask sparser; third, changing the attention operation in NanoGPT's model by using the newly obtained causal masks.

3.1 Extracting the standard causal mask

We start by training the model with a standard causal mask on a small dataset. After training, the attention weights from all the heads are extracted, representing how the model distributes attention under the standard mask. These weights form the basis for optimization in the subsequent step, where the mask structure is refined using Gurobi.

3.2 Optimization of the Causal Mask

The optimization of the causal mask refines the structure used during training to enhance efficiency and attention allocation while preserving the sequential nature of the problem. The causal mask ensures that each token attends only to previous tokens in the sequence, preventing future information from influencing past computations. Sparsity constraints further limit the number of non-zero entries, creating a minimal, computationally efficient mask while adhering to the autoregressive property.

The optimization problem is formulated as a mixed-integer quadratic programming (MIQP) task, where the objective is to minimize the difference between the predicted and actual weighted sums under the refined mask. Gurobi solves this MIQP using a branch-and-bound algorithm combined with cutting planes and heuristics. It iteratively relaxes the binary constraints, solves the relaxed problem, branches on fractional variables, and prunes subproblems based on bounds, effectively navigating the solution space. The result is an optimized block-diagonal causal mask that respects the autoregressive nature of the task while ensuring computational efficiency and improved attention allocation.

3.3 Testing the model with sparse causal mask

The final step involves testing the model with the optimized sparse causal mask obtained from the Gurobi optimization process. The sparse mask replaces the standard causal mask used during training, preserving the autoregressive property while significantly reducing the number of active attention connections. This step evaluates the model's performance in generating predictions under the new constraints, measuring the impact of the sparse mask on both computational efficiency and model accuracy. By applying the optimized mask, the model operates with fewer attention connections, leading to faster computations while maintaining or improving the quality of attention allocation. The results from this step help validate the effectiveness of the optimization process and highlight the trade-offs between sparsity and performance.

4 Experiment Settings

For experiments, we start with training a GPT-like model on Tiny ShakeSpeare using AdamW based on NanoGPT. The training takes around 12 minutes on a single RTX 3090. We used smaller model hyperparameters compared to the original model to make the post-training optimization feasible.

Hyperparameter	**Value**
Context length	64
Number of Layers	6
Number of Heads	8
Embedding Dimension	128
Dropout	0.2
Number of Iterations	50000
Maximum Learning Rate	1e-3

Table 1: Model Hyperparameters

The loss curve for training is shown in Fig.2.

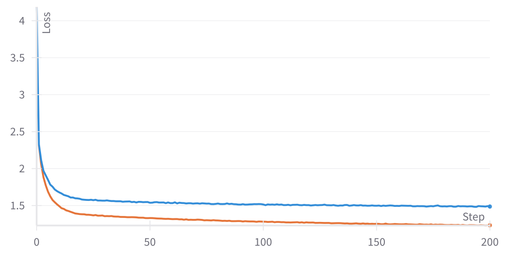


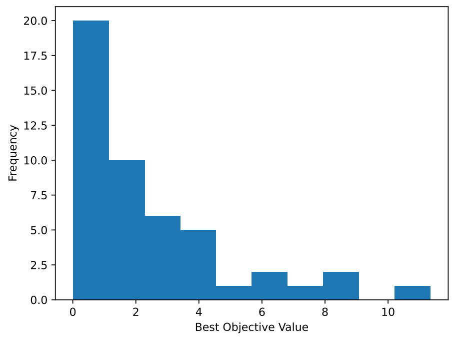
Figure 2: Loss curve for model training, training loss is in blue, validation loss is in orange

After training, we extract the Q, K, V terms for all the attention heads on a small subset of the training set, i.e., 4 training samples and run the optimization to get the optimized block sparse attention mask.

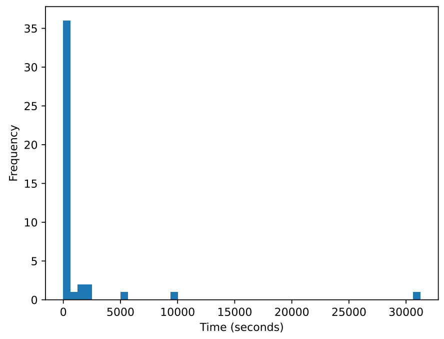
For optimization, we use 4 data points, with size of Q, K, V as $[4, 8, 64, 16]$, corresponding to data size, number of heads, context length, and head size. This result in a $QK^\top$ with size $[4, 8, 64, 64]$, we use a block size of 8, splitting the 64×64 matrix into 8×8 sub-matrices, leading to the sparse block mask Z with size $[8, 8, 8]$. We then optimize for each attention head individually, with a total of 6 (layer) $\times 8$ (head per layer) $= 48$ attention heads.

5 Results

The optimization task took around hours to finish on single Intel(R) Core(TM) i9-12900K. The distribution of the final objective and used time are shown in Fig. 3. Most optimizations finishes quickly, but the 5 slowest masks to optimize take 90% of the time to finish.



(a) Distribution of optimization loss in attention heads.



(b) Distribution of optimization time used.

Layer and Attention Head	Time (minutes)
Layer 6, Head 2	521.03
Layer 2, Head 7	156.98
Layer 5, Head 4	88.78
Layer 2, Head 6	40.11
Layer 1, Head 1	37.23

(c) Optimization time for the most time-intensive attention heads.

Figure 3: Overview of optimization results, including the distribution of optimization loss (a), distribution of optimization time (b), and a table of the most time-intensive attention heads (c).

We evaluated the results by swapping the causal attention mask in the trained model with optimized mask, sliding window mask, attention sink [2], and random sparse mask (all with the same level of sparsity) leveraging FlexAttention, the loss on evaluation set is shown in Table 2. We can see that the mask from optimization has little loss in performance compared to both sliding window and random sparse mask.

We also present some generation results in Fig.4. We can see that generation using optimized sparse mask has correct spelling and are mostly coherent within the sentence, while random sparse mask gives wrong spelling and nonsense generation, demonstrating the effectiveness of the optimization.

Used mask	Validation Loss
Original	1.4879
Optimized (Swapping all 6 layers)	**1.5206**
Sliding window (Context window= 16)	1.9240
Attention sink[2] (sink=context=8)	2.1560
Random block sparse mask	2.1899
Optimized (Swapping first 1 layer)	1.4887
Optimized (Swapping first 3 layers)	1.4901
Optimized (Keeping 3 heads and changing others to sliding window)	1.5244

Table 2: Validation Losses

Optimized sparse mask (ours):

```
Clown:
Repent with the sorrow waits and
extremant,
Shall do ala
---------------
MENENIUS:
No word, my lord: let me, away!

ROMEO:
I would must
---------------
ROMEO:
This is my mildness but sometime,
I heard it that virtue
---------------
Here in heaven that speaks no more.

KING RICHARD II:
That she w
---------------
LADY ANNE:
Good patiently; but you do love
In all so duty of yo
```

Random sparse mask:

```

When thy true sworn to laut, thas as my belooved
Brides thegath
---------------
MENENIUS:
No wholy veary ddily way, away, my famond,
An other t
---------------
ROMEO:
GLOUCESTER:
O hed the bemy father he wils is the overse,
---------------
Here in heart.

WARWICK:
No Gor Romeo, my loreven stand airs hat
---------------
Let us not be more to the hopenine, thousandery,--
And tho I be
---------------
To your doubt a cheering stome mor, for unch,
Anount
The sunders
```

Figure 4: Generation results compared with a random mask (note that the last word is cut off)

The analysis in Appendix B highlights two key insights.

First, the value of optimization over heuristics. Sparse masks often follow a sliding window pattern, but the sliding window heuristic alone is insufficient for effectively reducing validation loss. Instead, optimization proves essential, whether it refines masks closely aligned with the sliding window heuristic or significantly different ones.

Second, the importance of optimization over randomness. Figure 4 underscores this distinction by comparing the generation results of a random mask and an optimized mask. While the random mask produces incoherent outputs, the optimized mask generates meaningful results, illustrating the critical role of mask optimization in achieving superior outcomes.

6 Conclusion and Future Works

This project demonstrates the potential of optimizing block-sparse masks to improve the efficiency of attention mechanisms in large language models (LLMs). By leveraging block-sparsity to avoid computing unnecessary sub-matrices, we achieve reductions in computational cost without sacrificing performance. Our formulation as a mixed integer optimization problem, solved using Gurobi, allows us to reach up to 75% sparsity with negligible degradation in attention quality compared to the default causal mask. Experimental results on a GPT-like model show that our optimized masks maintain high performance and outperform heuristic methods such as sliding window and random sparse masks.

For future work, we will focus on scaling this approach to longer context length and larger models, which requires improving optimization speed and integrating GPU-based computation into the solving process to handle increased complexity. To further enhance performance, we plan to explore a dynamic sparsity strategy by allowing less sparsity in attention heads that contribute more significantly to model accuracy. This adaptive approach aims to balance efficiency and performance by targeting critical attention pathways while maintaining computational savings.

7 Contribution

- Shiji Xin: Formulation of the optimization problem, model training
- Melissa Rosenberg: Mask optimization, results analysis, presentation

References

[1] Guangxuan Xiao, Jiaming Tang, Jingwei Zuo, Junxian Guo, Shang Yang, Haotian Tang, Yao Fu, and Song Han. Duoattention: Efficient long-context llm inference with retrieval and streaming heads. *arXiv*, 2024.

[2] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. *arXiv*, 2023.

[3] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. Big bird: Transformers for longer sequences. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 17283–17297, 2020.

A Optimization code

```
def problem(n, d, b, kk, q, k, v):
    m = gp.Model()

    # Create variables
    A = np.zeros((b, n, n))
    exp_A = np.zeros((b, n, n))

    # precompute A and exp_A
    for i in range(b):
        A[i, :, :] = np.dot(q[i, :, :], k[i, :, :].T)
        A[i, :, :] /= np.sqrt(d)
        A[i, :, :] -= np.max(A[i, :, :], axis=1, keepdims=True)
        exp_A[i, :, :] = np.exp(A[i, :, :])
        exp_A[i, :, :] = np.tril(exp_A[i, :, :])

    n_sub = n // kk # Number of blocks

    # Variables for all batches
    z = m.addMVar((n_sub, n_sub), vtype=GRB.BINARY, name='z')
    row_sum = np.sum(exp_A, axis=2, keepdims=True)  # Shape: (b, n, 1)
    normalized = exp_A / row_sum  # Shape: (b, n, n)

    masked_row_sum = m.addMVar((b, n), lb=0, name='masked_row_sum')
    masked_normalized = m.addMVar((b, n, n), lb=0, name='
masked_normalized')
    is_non_zero = m.addMVar((b, n_sub), vtype=GRB.BINARY, name='
is_non_zero')

    # Objective: minimize sum of squared differences for all batches
    obj = quicksum(((normalized[batch] - masked_normalized[batch]) * (
normalized[batch] - masked_normalized[batch])).sum() for batch in
range(b))
    m.setObjective(obj, GRB.MINIMIZE)

    # Constraints for each batch
    for batch in range(b):
        for i in range(n):
            block_i = i // kk
            m.addConstr(
                masked_row_sum[batch, i] == quicksum(
                    exp_A[batch, i, j] * z[block_i, j // kk] for j in range(n)
                )
            )
            for j in range(n):
                block_j = j // kk
                m.addConstr(
                    masked_normalized[batch, i, j] * masked_row_sum[batch, i]
                    == exp_A[batch, i, j] * z[block_i, block_j]
                )
                m.addConstr(
                    masked_normalized[batch, i, j] <= is_non_zero[batch, block_i]
                )
        for i in range(n_sub):
            m.addConstr(
                quicksum(z[i, j] for j in range(n_sub)) <= n_sub * is_non_zero[batch, i]
            )
            m.addConstr(
                quicksum(z[i, j] for j in range(n_sub)) >= is_non_zero[batch, i]
            )

    # Apply causal mask at block level
    for i in range(n_sub):
        for j in range(i + 1, n_sub):
            m.addConstr(z[i, j] == 0)

    # Total number of non-zero entries in z
    m.addConstr(
        quicksum(z[i, j] for i in range(n_sub) for j in range(n_sub)) <= 2 * n_sub
    )

    # Solve
    m.optimize()

    # Get the solution
    z_array = np.zeros((n_sub, n_sub))
    for i in range(n_sub):
        for j in range(n_sub):
            z_array[i, j] = z[i, j].X
    return z_array
```

Listing 1: Optimization code

B Visualization of optimized masks

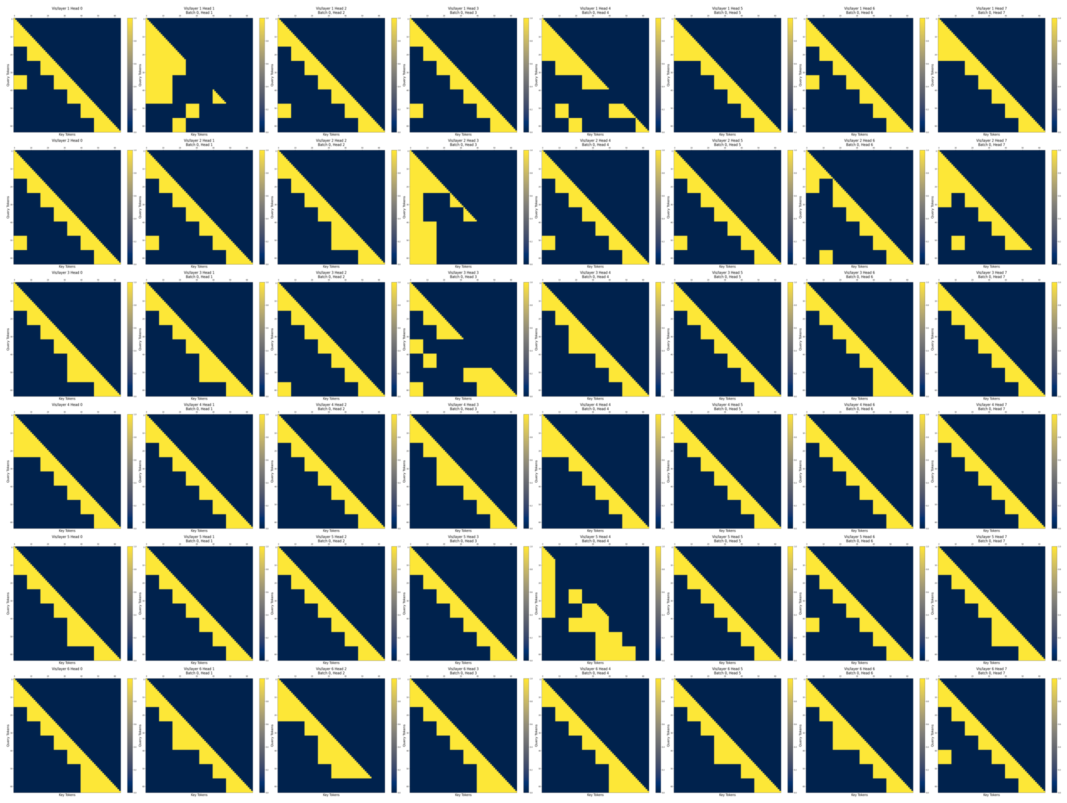


Figure 5: Visualization of attention heads across layers. Each block represents a head in a specific layer, with row corresponding to layer and column corresponding to head