# Getting *free* Bits Back from Rotational Symmetries in LLMs

**Jiajun He**
University of Cambridge
jh2383@cam.ac.uk

**Gergely Flamich**
University of Cambridge
gf332@cam.ac.uk

**José Miguel Hernández-Lobato**
University of Cambridge
jmh233@cam.ac.uk

## Abstract

Current methods for compressing neural network weights, such as decomposition, pruning, quantization, and channel simulation, often overlook the inherent symmetries within these networks and thus waste bits on encoding redundant information. In this paper, we propose a format based on bits-back coding for storing rotationally symmetric Transformer weights more efficiently than the usual array layout at the same floating-point precision. We evaluate our method on Large Language Models (LLMs) pruned by SliceGPT (Ashkboos et al., 2024) and achieve a 3-5% reduction in total bit usage *for free* across different model sizes and architectures without impacting model performance within a certain numerical precision.

## 1 Introduction

Modern neural networks, particularly Large Language Models (LLMs), typically contain billions of parameters. Therefore, encoding and transmitting these models efficiently is gaining widespread interest. Currently, compression techniques of model weights mainly fall into four categories, including decomposition (e.g., Hu et al., 2022; Saha et al., 2023), pruning (e.g., Hoefler et al., 2021; Frantar and Alistarh, 2023; Ashkboos et al., 2024), quantization (e.g., Wang et al., 2023; Xu et al., 2024), and channel simulation (e.g., Havasi et al., 2019; Isik et al., 2023; He et al., 2024).

However, these techniques ignore the fact that neural networks typically exhibit symmetries in their weight space. For example, in feedforward networks, applying a random permutation to the neurons in one layer and its inverse to the weights in the subsequent layer leaves the output unchanged. Encoding weights without accounting for these symmetries will lead to suboptimal codelength.

In this work, we consider addressing this redundancy through bits-back coding. We focus on Transformers pruned by SliceGPT (Ashkboos et al., 2024), a recent method for pruning Transformer weights, which not only reduces the total number of parameters with minimal impact on performance but introduces rotation symmetries into the Transformer weights. Specifically:

- We introduce bits-back coding to compress neural network weights, which can reduce the storage by saving bits from the symmetries in the weight space.
- Particularly, we propose a practical bits-back coding scheme for rotation symmetries. We apply our approach to Large Language Models (LLMs) pruned by SliceGPT and demonstrate that our proposed approach can save additional free bits while preserving prediction accuracy within a certain numerical precision.
- We further showcase that by transmitting a small number of bits as a correction code, we can rescue the performance drops due to numerical inaccuracies. Applying our approach with this correction code, we can, in total, save 3-5% bits across different sizes and models *for free*.

## 2 Background

Before launching into our methods, we provide a brief introduction to bits-back coding (Frey and Hinton, 1996), Transformer (Vaswani et al., 2017), and SliceGPT (Ashkboos et al., 2024).

**Bits-back Coding.** Bits-back argument was first proposed by Hinton and Van Camp (1993) and turned into a coding algorithm by Frey and Hinton (1996). Bits-back applies to the cases where the source produces multiple codewords for a given symbol. Bits-back coding was then extended by Townsend et al. (2019) to cope with latent variable models for lossless data compression, and by Kunze et al. (2024) to encode unordered data structures. The basic idea behind bits-back coding is simple: When encoding a symbol, the encoder uses some bits from the bitstream to choose one of several possible codewords for that symbol. When decoding, the decoder identifies the symbol, extracts the bits used for selecting the codeword, and then restores those bits to the bitstream. This approach reduces the redundant bits from multiple codeword choices, achieving asymptotic efficiency. In this work, we apply the same principle to transformer weights with rotation symmetries, where the multiple codewords come from different rotations of network weights.

**Transformer Architecture and SliceGPT.** Transformer (Vaswani et al., 2017) is the cornerstone of most Large Language Models. Its basic component is the transformer block, as shown in Figure 1a. Each block consists of a multi-head attention layer, a LayerNorm (Ba et al., 2016), and a feedforward network (FFN). Two residual connections are added around the attention layer and FFN.

SliceGPT (Ashkboos et al., 2024) is a recently proposed method for pruning weights in Transformer models. The approach leverages the insight that the outcome of LayerNorm (more precisely, RMSNorm, i.e., $\mathbf{x} \leftarrow \mathbf{x}/||\mathbf{x}||$) is invariant if we apply a rotation to the input and its inverse to the output. This rotation matrix and its inverse can be absorbed into the weights before and after the normalization layer. Therefore, by performing PCA on the hidden states, we can choose rotation matrices that align with the principal components. This allows us to prune the rows and columns corresponding to the less significant eigenvalues in the hidden states, effectively reducing the model's complexity without drastically hurting the performance. We visualize each transformer block after rotation and pruning in Figure 1b. The shadow indices the pruned columns and rows.

## 3 Getting bits back from Rotation Symmetries

In this section, we describe our method, which is based on the observation of rotation symmetries in the Transformer block pruned by SliceGPT. Comparing Figure 1b and Figure 1a, we can see SliceGPT not only reduces the number of parameters (by pruning out columns and rows), but also introduces rotation symmetries. Concretely, in a SliceGPT-pruned Transformer, denoting the weights in the $\ell$-th transformer block with superscripts, we have:

**Remark 3.1.** *Outputs remain unchanged if rotating $\mathbf{W}_2^{(\ell-1)}$, $\mathbf{b}_2^{(\ell-1)}$ (if any) and $\mathbf{Q}_{\textit{skip\_mlp}}^{(\ell-1)}$ by $\mathbf{Q}$, and rotating $\mathbf{Q}_{\textit{skip\_att}}^{(\ell)}, \mathbf{W}_{qkv}^{(\ell)} = \left[\mathbf{W}_k^{(\ell)}, \mathbf{W}_q^{(\ell)}, \mathbf{W}_v^{(\ell)}\right]$ by $\mathbf{Q}^\top$ as follows:*

$$\mathbf{W}_2^{(\ell-1)} \leftarrow \mathbf{W}_2^{(\ell-1)}\mathbf{Q}, \quad \mathbf{b}^{(\ell-1)} \leftarrow \mathbf{b}^{(\ell-1)}\mathbf{Q}, \quad \mathbf{Q}_{\textit{skip\_mlp}}^{(\ell-1)} \leftarrow \mathbf{Q}_{\textit{skip\_mlp}}^{(\ell-1)}\mathbf{Q} \tag{1}$$

$$\mathbf{Q}_{\textit{skip\_att}}^{(\ell)} \leftarrow \mathbf{Q}^\top\mathbf{Q}_{\textit{skip\_att}}^{(\ell)}, \quad \mathbf{W}_{qkv}^{(\ell)} \leftarrow \mathbf{Q}^\top\mathbf{W}_{qkv}^{(\ell)} \tag{2}$$

*Similarly, outputs remain unchanged if rotating $\mathbf{W}_o^{(\ell)}$, $\mathbf{b}_o^{(\ell)}$ (if any) and $\mathbf{Q}_{\textit{skip\_att}}^{(\ell)}$ by $\mathbf{Q}$, and rotating $\mathbf{Q}_{\textit{skip\_mlp}}^{(\ell)}$ and $\mathbf{W}_1^{(\ell)}$ by $\mathbf{Q}^\top$ as follows:*

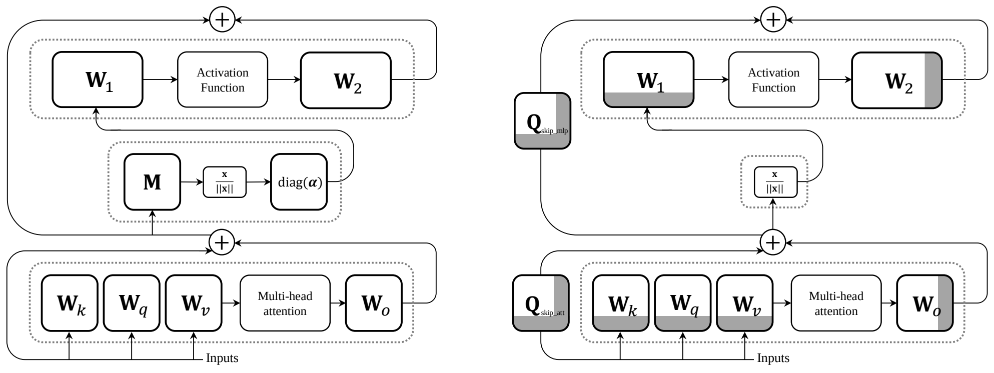
$$\mathbf{W}_o^{(\ell)} \leftarrow \mathbf{W}_o^{(\ell)}\mathbf{Q}, \quad \mathbf{b}_o^{(\ell)} \leftarrow \mathbf{b}_o^{(\ell)}\mathbf{Q}, \quad \mathbf{Q}_{\textit{skip\_att}}^{(\ell)} \leftarrow \mathbf{Q}_{\textit{skip\_att}}^{(\ell)}\mathbf{Q} \tag{3}$$

$$\mathbf{Q}_{\textit{skip\_mlp}}^{(\ell)} \leftarrow \mathbf{Q}^\top\mathbf{Q}_{\textit{skip\_mlp}}^{(\ell)}, \quad \mathbf{W}_1^{(\ell)} \leftarrow \mathbf{Q}^\top\mathbf{W}_1^{(\ell)} \tag{4}$$

This symmetry suggests that directly encoding the weights (e.g., in `float16`) would use more bits than necessary. In the following, we offer an informal explanation to clarify this redundancy:

For simplicity, let's denote the weights in a transformer as $\mathbf{\Theta}$. Assuming the coding distribution is $Q$[1], we need to spend about $-\log_2 Q(\mathbf{\Theta})$ bits to encode the weights directly. On the other hand, as

[1]If we encode the weights using `float16`, we are essentially assuming that all possible floating-point values ($2^{16}$ in total) have the same probability mass.



(a) A standard transformer block. (b) A transformer block with SliceGPT.

Figure 1: Visualization of a Standard Transformer Block and a SliceGPT-Pruned Transformer Block. (a) The standard Transformer block first maps the input through an attention layer; then it applies LayerNorm (Ba et al., 2016) and a 1-layer Feedforward Network (FFN). Two residual connections are added after the attention layer and the FFN. Here, we adopt the notation by Ashkboos et al. (2024), where $\mathbf{M} = \mathbf{I} - \frac{1}{D}\mathbf{1}\mathbf{1}^\top$ represents the operation that subtracts the mean in each row. (b) SliceGPT (Ashkboos et al., 2024) first absorbs $\mathbf{M}$ and $\operatorname{diag}(\boldsymbol{\alpha})$ into the weights before and after the normalization layer. It then rotates these weights by applying PCA to the hidden states, aligning them with their principal components (PCs). Subsequently, SliceGPT prunes rows and columns corresponding to the least significant PCs, indicated by gray shadows. It is important to note that the weights in (b) differ from those in (a) due to the absorption of $\mathbf{M}$ and $\operatorname{diag}(\boldsymbol{\alpha})$ and the rotation. For a more detailed explanation of SliceGPT, please refer to Figure 4 in Ashkboos et al. (2024).

discussed above, applying rotations (and its inversion) to some weights leaves the output invariant. Therefore, if we define *equivalence* in terms of outputs (and we do!), the weights with different rotations form an *equivalence class*, denoted by $[\boldsymbol{\Theta}]$. Encoding this equivalence class will require $-\log_2\left(\sum_{\boldsymbol{\Theta}\in[\boldsymbol{\Theta}]} Q(\boldsymbol{\Theta})\right)$ bits. In a finite-precision system, where the number of possible rotation matrices is limited, the equivalence class is finite. Assuming that each entry in the equivalence class has the same probability, and denoting the cardinality of the equivalence class by $\mathcal{C}$, we have $-\log_2\left(\sum_{\boldsymbol{\Theta}\in[\boldsymbol{\Theta}]} Q(\boldsymbol{\Theta})\right) = -\log_2\left(\mathcal{C}Q(\boldsymbol{\Theta})\right) = -\log_2 Q(\boldsymbol{\Theta}) - \log_2 \mathcal{C}$. This implies that directly encoding the weights wastes $-\log_2 \mathcal{C}$ bits more than necessary.

To this end, we apply bits-back coding to eliminate this redundancy. In short, each time we encode the weights in one transformer block (more precisely, $\mathbf{W}_2^{(\ell)}$ and $\mathbf{W}_o^{(\ell)}$), we start by decoding a random rotation from the current bitstream and applying it to the weights. We then encode the rotated weights into the bitstream. When decoding, we first decode the rotated weights and recover the rotation we applied to the original weights. Then, we encode the rotation matrix back to the bitstream. This process is repeated for every transformer block, and hence, the redundancy caused by the rotation symmetries will be eliminated asymmetrically.

However, there are two questions remaining unsolved: (a) *how can we recover the rotation given a rotated weight matirx?* (b) *how can we decode/encode a rotation (Orthogonal) matrix from/to the current bitstream?* We will answer these questions in Section 3.1 and Section 3.2, respectively. We then put things all together in Section 3.3 and describe the full encoding and decoding algorithms in Algorithms 5 and 6. Besides, as we only apply rotations to weight matrices with finite precision (e.g., `float16`), we may suffer from numerical inaccuracy, impacting the transformer's outputs. To handle this, we propose to send a simple correction code, which we discuss at the end of Section 3.3.

## 3.1 Rotating Transformer to its Canonical Direction

We now discuss how to recover the rotation from a rotated weight matrix. This is, in general, not feasible without additional information about the original weight. Fortunately, as noted in Remark 3.1, we can apply any rotation to the weights. This allows us to first rotate the weights to a *canonical direction* as a reference. We can define this canonical direction in multiple ways as long as we can recover it easily after applying a random rotation. In this work, we adopt eigenvalue decomposition to define the canonical direction, while future works could explore more sophisticated methods.

We detail the algorithm for the canonical direction in Algorithm 1. In short, for each transformer block, we can apply two free rotations according to Remark 3.1: the first rotation is applied to $\mathbf{W}_2^{(\ell-1)}$, $\mathbf{b}_2^{(\ell-1)}$, $\mathbf{Q}_{\text{skip\_mlp}}^{(\ell-1)}$, $\mathbf{Q}_{\text{skip\_att}}^{(\ell)}$, and $\mathbf{W}_{qkv}^{(\ell)}$. We hence define the canonical direction such that ${\mathbf{W}_2^{(\ell-1)}}^\top \mathbf{W}_2^{(\ell-1)}$ is diagnoal; the second rotation is applied to $\mathbf{W}_o^{(\ell)}$, $\mathbf{b}_o^{(\ell)}$, $\mathbf{Q}_{\text{skip\_att}}^{(\ell)}$, $\mathbf{Q}_{\text{skip\_mlp}}^{(\ell)}$ and $\mathbf{W}_1^{(\ell)}$. We hence define the canonical direction such that ${\mathbf{W}_o^{(\ell)}}^\top \mathbf{W}_o^{(\ell)}$ is diagnoal.

After rotating the transformer to its canonical direction, we can recover any rotation that is applied to the canonical $\mathbf{W}_2^{(\ell-1)}$ or $\mathbf{W}_o^{(\ell)}$ by eigenvalue decomposition. Specifically, let's consider a random rotation $\mathbf{Q}$ applied to $\mathbf{W}_2^{(\ell-1)}$ in its canonical direction as an example. Denoting the matrix after rotation is $\tilde{\mathbf{W}}_2^{(\ell-1)} \leftarrow \mathbf{W}_2^{(\ell-1)}\mathbf{Q}$, we can perform eigenvalue decomposition on $\tilde{\mathbf{W}}_2^{(\ell-1)\ \top}\tilde{\mathbf{W}}_2^{(\ell-1)}$, and the rotation matrix $\mathbf{Q}$ can then be recovered by stacking the eigenvectors together in columns. The weight matrix in the canonical direction can be obtained by $\mathbf{W}_2^{(\ell-1)} \leftarrow \tilde{\mathbf{W}}_2^{(\ell-1)}\mathbf{Q}^\top = \mathbf{W}_2^{(\ell-1)}\mathbf{Q}\mathbf{Q}^\top$.

A caveat exists in the above procedure: eigenvalue decomposition can result in eigenvectors with opposite signs. This will lead to undesired results when recovering the canonical weight matrix. We include a detailed explanation in Appendix B. To address this, we encode the sign of the summation of each row of the rotation matrix as side information. This only requires $D$ bits for a $D$-dimensional rotation matrix. After recovering eigenvectors through eigenvalue decomposition, we can use this side information to correct the sign for each eigenvector (i.e., rows in the rotation matrix). Algorithm 2 describes this process.

Another concern arises when ${\mathbf{W}_2^{(\ell-1)}}^\top \mathbf{W}_2^{(\ell-1)}$ (or $\mathbf{W}_o^{(\ell)\ \top}\mathbf{W}_o^{(\ell)}$) is not full-rank. In such cases, eigenvalue decomposition will not recover the rotation applied to these canonical weights. To address this, we can define the canonical direction by applying eigenvalue decomposition to $\mathbf{B}^\top\mathbf{B}$, where $\mathbf{B}^\top = \left[{\mathbf{W}_2^{(\ell-1)}}^\top, {\mathbf{b}_2^{(\ell-1)}}^\top, \mathbf{W}_k^{(\ell)}, \mathbf{W}_q^{(\ell)}, \mathbf{W}_v^{(\ell)}\right]$ (or $\mathbf{B}^\top = \left[{\mathbf{W}_o^{(\ell)}}^\top, {\mathbf{b}_o^{(\ell)}}^\top, \mathbf{W}_1^{(\ell)}\right]$). However, we actually found ${\mathbf{W}_2^{(\ell-1)}}^\top \mathbf{W}_2^{(\ell-1)}$ and $\mathbf{W}_o^{(\ell)\ \top}\mathbf{W}_o^{(\ell)}$ were already full-rank across all architectures in our experiments. This may be because SliceGPT has already pruned insignificant principal components in the hidden states, leading to more compact weight matrices.

## 3.2 Decoding and Encoding Rotation Matrices

Now, we discuss *how to decode/encode a rotation matrix from/to a given bitstream.* A naive approach is to directly decode and encode these $D^2$ entries in a rotation matrix $\mathbf{Q} \in \mathbb{R}^{D\times D}$, e.g., by `float16`. However, it is difficult to guarantee that $D^2$ elements decoded from a given bitstream can form a rotation matrix. In fact, a $D$-dimensional rotation matrix $\mathbf{Q}$ has only $D(D-1)/2$ degrees of freedom (DOF), which means that we only need to decode and encode $D(D-1)/2$ floats for the entire matrix. Therefore, the question becomes: *(a) how can we construct a random rotation matrix from $D(D-1)/2$ random floats; (b) how can we recover these floats given a rotation matrix?*

Ideally, we aim to generate a uniformly distributed random rotation matrix, i.e., a random rotation matrix from the Haar distribution. Following the method by Stewart (1980), we can construct the matrix by iteratively applying Householder transformations (Householder, 1958). This is also the approach in `scipy.stats.special_ortho_group` for generating random rotation matrices.

However, this algorithm is difficult to reverse: we need to reverse the householder transformations one by one, and hence, we will suffer from large numerical instability. Therefore, we propose a simple method to generate a rotation matrix. This approach does not result in a uniformly distributed rotation matrix. However, we found our approach works well in practice. Since our goal is not to design a theoretically optimal algorithm but rather a more practical approach to perform bits-back, we leave a better design for the rotation matrix to future works.

We describe the process of decoding and encoding a rotation matrix in Algorithms 3 and 4. In brief, to decode a rotation matrix, we first decode a symmetric matrix from the bitstream by decoding its diagonal and upper triangular parts and performing an eigenvalue decomposition. The eigenvalues are then encoded back into the bitstream. To encode this rotation matrix, we first decode its eigenvalues from the bitstream, reconstruct the symmetric matrix via matrix multiplication, and then encode its diagonal and upper triangular parts back into the bitstream. Notably, our approach requires only the number of bits corresponding to $D(D-1)/2$ floats, which aligns with the degrees of freedom of a random rotation matrix.

### 3.3 Putting Things Together and Handling Numerical Inaccuracy

Having discussed the canonical direction for the transformer and the algorithm for decoding and encoding a rotation matrix, we detail the complete algorithm for encoding and decoding the entire transformer using bits-back in Algorithms 5 and 6, respectively. In these algorithms, we use `Encode_to` and `Decode_from` to represent the process of appending or popping arrays of `float16` values into or from the current bitstream.

However, since we only save rotated weights in finite precision (e.g., `float16`), we may suffer from numerical inaccuracy, and hence the rotation matrix recovered by Algorithm 2 in decoding will have deviations from the original rotation matrix applied to the canonical weights in encoding. This will lead to two unideal outcomes: (a) the bitstream after re-encoding the rotation matrices (as shown in lines 7 and 13 in Algorithm 6) will contain errors, which will affect the weights decoded subsequently from this bitstream; (b) the weight matrices rotated back to the canonical direction (as shown in lines 6 and 12 in Algorithm 6) will contain errors.

The first error can be fatal in a standard bits-back coding algorithm. This is because the bits-back information and the information in the original bitstream can be misaligned - the least significant bit (LSB) in the bits-back information can become the most significant bit (MSB) in the original information. Consequently, a small error in the bits-back information may significantly alter the original bitstream. Fortunately, this misalignment does not occur in our algorithms. This is because all numbers in the bitstream are stored as `float16`: we encode all weights in `float16`, and we also encode the rotation matrix using `float16` values, as described in Algorithms 3 and 4. Therefore, errors in LSB will always remain in the LSB and will not be amplified. The only exception is the bits representing the sign for each row of the rotation matrix (as shown in lines 9 and 14 in Algorithm 5). However, since the codelength for these signs is known in advance (determined by the transformer architecture), we can simply append these bits as a postfix to the entire bitstream and encode/decode them separately from the other float values.

However, although the bits-back process will not amplify the error, the error itself can still impact performance. Therefore, we propose transmitting an additional correction code to correct errors exceeding a certain threshold. Specifically, errors in (a) occur in the $D(D+1)/2$ floats obtained by Algorithm 3 when encoding the rotation matrix to the bitstream, and errors in (b) occur when rotating the weight matrices back to the canonical direction. Note that the encoder can simulate both procedures during encoding to determine the exact value that the decoder will obtain. If the error between the value obtained by the decoder and the one held by the encoder exceeds a certain threshold, the encoder can send a correction code containing the positions and the true values in `float16`. Correcting each value will require approximately $16 + \lceil \log_2 L \rceil$ bits, where $L$ is the total number of values the decoder will reconstruct that can have errors. For example, $L = D(D+1)/2$ for the error caused by (a), and $L$ represents the total number of parameters in the weight matrix for the error caused by (b).

A natural concern is that the correction code could become large if there are too many errors. Fortunately, as we show in Figure 2, only a tiny portion of values have relatively large errors. Therefore, the correction code requires only a small number of bits to transmit and does not significantly impact the overall coding efficiency. It is worth noting that this correcting strategy can be considered a simple error-correction code. Therefore, we may be able to adopt more complex error-correction codes, but we leave this design for future exploration.

## 4 Experiments and Results

We evaluate our proposed approach in this section. We first test our method on the Open Pretrained Transformer Language Models (OPT, Zhang et al., 2022) and Llama-2 (Touvron et al., 2023) pruned by SliceGPT (Ashkboos et al., 2024) with different slicing rates. Then, we investigate the effectiveness of the correction codes proposed in Section 3.3.

Table 1: Compression rates and prediction performances before and after our proposed method. Our method reduces further 3-5% bits and has very minor influence on the performance.

| **Model** | **SliceGPT** | **Compress Rate** | **Compress Rate** | **Performance (before/after bits-back)** | | | |
|---|---|---|---|---|---|---|---|
| | **Slicing** | **after SliceGPT** | **after bits-back** | PPL ($\downarrow$) | PIQA (%, $\uparrow$) | WinoGrande (%, $\uparrow$) | HellaSwag (%, $\uparrow$) |
| OPT-1.3B | 20% | -9.53% | -13.77% | **16.59**/16.60 | **64.91**/64.80 | **54.78**/54.38 | 45.26/**45.32** |
| | 25% | -14.84% | -18.61% | **17.78**/17.86 | **63.55**/63.33 | 52.80/**53.28** | **43.20**/43.11 |
| | 30% | -20.53% | -23.81% | **19.60**/19.66 | **60.88**/60.50 | 52.88/**53.28** | **40.25**/40.06 |
| OPT-2.7B | 20% | -9.19% | -13.84% | **13.89**/13.95 | **68.44**/68.12 | **58.88**/58.72 | **51.35**/51.17 |
| | 25% | -15.07% | -19.09% | **14.85**/14.87 | **66.70**/66.76 | 57.30/**57.70** | **48.41**/48.38 |
| | 30% | -20.88% | -24.43% | **16.31**/16.33 | 64.64/**64.69** | 55.80/**56.04** | 44.52/**44.57** |
| OPT-6.7B | 20% | -9.29% | -14.07% | **11.63**/11.71 | 72.91/**73.01** | **61.33**/61.17 | 60.53/**60.55** |
| | 25% | -15.16% | -19.29% | **12.12**/12.15 | 71.00/**71.22** | 60.30/**60.77** | **57.76**/57.55 |
| | 30% | -21.18% | -24.84% | **12.81**/12.91 | 69.31/**69.42** | **59.75**/59.59 | **53.64**/52.94 |
| OPT-13B | 20% | -9.18% | -14.01% | **10.75**/10.77 | 74.27/74.27 | **64.96**/64.88 | 65.74/**65.79** |
| | 25% | -15.27% | -19.51% | 11.08/**11.07** | **74.27**/73.72 | 63.46/**63.93** | **63.48**/63.09 |
| | 30% | -21.29% | -24.97% | **11.55**/11.59 | 72.69/**73.01** | 61.96/**62.43** | **60.12**/60.05 |
| Llama-2-7B | 20% | -9.38% | -14.13% | **6.86**/6.98 | **69.53**/69.42 | 64.17/**64.72** | **58.96**/58.89 |
| | 25% | -15.34% | -19.53% | **7.56**/7.59 | 67.03/**67.57** | 62.98/**63.38** | **54.29**/53.93 |
| | 30% | -21.45% | -25.09% | **8.63**/8.69 | **64.69**/64.09 | **62.75**/62.12 | **49.13**/49.07 |

**Compression rate and performances.** We evaluate our method on OPT-1.3B/2.7B/6.7B/13B and Llama-2-7B, pruned by SliceGPT with different slicing rates. We report perplexity (PPL) and accuracy on three downstream tasks (PIQA, Bisk et al. (2020); WinoGrande, Sakaguchi et al. (2021); and HellaSwag, Zellers et al. (2019)) to assess our method's impact on performance. Our approach saves an additional 3-5% in bits with negligible impact on performance. Notably, the performance changes are inconsistent, with occasional improvements after bits-back, suggesting that the changes in the performance are more likely due to randomness than a clear degradation. We also note that this codelength reduction is smaller than the theoretical estimates provided in Appendix D. The primary reason for this discrepancy is that our analysis does not account for the substantial size of the head and embedding layers.

**Numerical inaccuracy and the effectiveness of the correction codes.** We now examine the impact of numerical inaccuracies and the effectiveness of correction codes proposed in Section 3.3. To provide an intuitive understanding of the numerical issue, we use the weights matrix $\mathbf{W}_o$ from the last layer of OPT-6.7B as an example and visualize the error between the reconstructed weights and the original weights in Figure 2 in Appendix. As we can see, only a tiny fraction of the weights exhibit relatively large errors. Therefore, we can transmit the positions and true values of weights whose deviations exceed a certain threshold, using negligible bits to correct the numerical error.

The threshold is a hyperparameter that balances the codelength and accuracy. In Figure 3, we examine the impact of threshold selection using the OPT-2.7B model. Setting a relatively small threshold (0.005-0.01) effectively mitigates nearly all performance drops due to numerical inaccuracies, while still providing a significant reduction compared to the compression rate without bits-back coding. In our experiments, we use a threshold of 0.01 for OPT models and 0.005 for Llama models.

## 5 Conclusion and Limitations

In this work, we introduce bits-back coding to encode Transformers in Large Language Models pruned by SliceGPT. Our approach can save 3-5% additional bits *for free* across several different architectures and sizes. While bits-back coding has long been applied in data compression, its application to neural networks, where redundancy and symmetry are prevalent, has been underexplored. Our work attempts to bridge this gap, opening a new direction for model compression. Key takeaway is that by re-parameterizing and pre-processing network weights to explicitly capture symmetries, as demonstrated in SliceGPT, we can employ bits-back coding to eliminate redundant bits.

Future research can focus on designing improved algorithms for encoding and decoding the random rotation matrix, developing better error-correction codes to manage large deviations caused by numerical instability, and integrating our method with other model compression techniques, such as the extremely quantized networks proposed by Ma et al. (2024). Our method's major concern is the numerical instability. While we discuss reducing large deviations by sending a small number of bits as a correction code, the challenge of making this approach efficient for extremely quantized networks or incorporating it with other pruning techniques remains open.

## Acknowledgments

JH was supported by the University of Cambridge Harding Distinguished Postgraduate Scholars Programme. JH and JMHL acknowledge support from a Turing AI Fellowship under grant EP/V023756/1. GF acknowledges funding from DeepMind.

## References

Saleh Ashkboos, Maximilian L Croci, Marcelo Gennari do Nascimento, Torsten Hoefler, and James Hensman. Slicegpt: Compress large language models by deleting rows and columns. In *The Twelfth International Conference on Learning Representations*, 2024.

Edward J Hu, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. Lora: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022.

Rajarshi Saha, Varun Srivastava, and Mert Pilanci. Matrix compression via randomized low rank and low precision factorization. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 18828–18872. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/3bf4b55960aaa23553cd2a6bdc6e1b57-Paper-Conference.pdf.

Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *Journal of Machine Learning Research*, 22(241):1–124, 2021.

Elias Frantar and Dan Alistarh. Sparsegpt: Massive language models can be accurately pruned in one-shot. In *International Conference on Machine Learning*, pages 10323–10337. PMLR, 2023.

Hongyu Wang, Shuming Ma, Li Dong, Shaohan Huang, Huaijie Wang, Lingxiao Ma, Fan Yang, Ruiping Wang, Yi Wu, and Furu Wei. Bitnet: Scaling 1-bit transformers for large language models. *arXiv preprint arXiv:2310.11453*, 2023.

Yuzhuang Xu, Xu Han, Zonghan Yang, Shuo Wang, Qingfu Zhu, Zhiyuan Liu, Weidong Liu, and Wanxiang Che. Onebit: Towards extremely low-bit large language models. *arXiv preprint arXiv:2402.11295*, 2024.

Marton Havasi, Robert Peharz, and José Miguel Hernández-Lobato. Minimal random code learning: Getting bits back from compressed model parameters. In *7th International Conference on Learning Representations, ICLR 2019*, 2019.

Berivan Isik, Francesco Pase, Deniz Gunduz, Tsachy Weissman, and Zorzi Michele. Sparse random networks for communication-efficient federated learning. In *The Eleventh International Conference on Learning Representations*, 2023.

Jiajun He, Gergely Flamich, Zongyu Guo, and José Miguel Hernández-Lobato. Recombiner: Robust and enhanced compression with bayesian implicit neural representations. In *The Twelfth International Conference on Learning Representations*, 2024.

B.J. Frey and G.E. Hinton. Free energy coding. In *Proceedings of Data Compression Conference - DCC '96*, pages 73–81, 1996. doi: 10.1109/DCC.1996.488312.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30, 2017.

Geoffrey E Hinton and Drew Van Camp. Keeping the neural networks simple by minimizing the description length of the weights. In *Proceedings of the sixth annual conference on Computational learning theory*, pages 5–13, 1993.

James Townsend, Thomas Bird, and David Barber. Practical lossless compression with latent variables using bits back coding. In *International Conference on Learning Representations*, 2019.

Julius Kunze, Daniel Severo, Giulio Zani, Jan-Willem van de Meent, and James Townsend. Entropy coding of unordered data structures. In *The Twelfth International Conference on Learning Representations*, 2024.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

G. W. Stewart. The efficient generation of random orthogonal matrices with an application to condition estimators. *SIAM Journal on Numerical Analysis*, 17(3):403–409, 1980. ISSN 00361429. URL http://www.jstor.org/stable/2156882.

Alston S Householder. Unitary triangularization of a nonsymmetric matrix. *Journal of the ACM (JACM)*, 5(4):339–342, 1958.

Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

Yonatan Bisk, Rowan Zellers, Jianfeng Gao, Yejin Choi, et al. Piqa: Reasoning about physical commonsense in natural language. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 7432–7439, 2020.

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: An adversarial winograd schema challenge at scale. *Communications of the ACM*, 64(9):99–106, 2021.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence? In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4791–4800, 2019.

Shuming Ma, Hongyu Wang, Lingxiao Ma, Lei Wang, Wenhui Wang, Shaohan Huang, Li Dong, Ruiping Wang, Jilong Xue, and Furu Wei. The era of 1-bit llms: All large language models are in 1.58 bits. *arXiv preprint arXiv:2402.17764*, 2024.

## A Additional Experimental Results

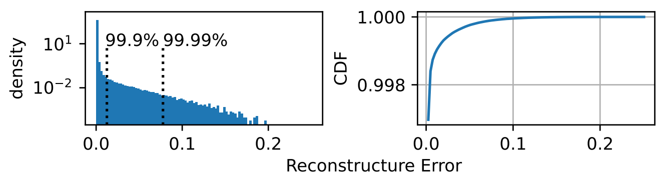


Figure 2: Histogram and empirical CDF of the error between the reconstructed weights and the original weights before encoding, using $\mathbf{W}_o$ in the final layer of OPT-6.7B as an example. The pattern in this plot generalizes well to other weights and models. As shown, only a small fraction of the weights exhibit relatively large deviations. Therefore, we can allocate a negligible number of bits to transmit the positions and true values of these weights, effectively correcting the error caused by numerical inaccuracies.

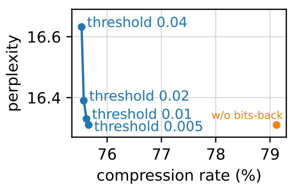


Figure 3: The effectiveness of the correction codes with different thresholds. Setting a threshold around 0.005-0.01 can effectively rescue all performance drops due to numerical inaccuracies while still significantly reducing bits compared to the compression rate without bits-back.

## B Why we need to encode the sign of each eigenvector?

First, assume we apply a random rotation matrix $\mathbf{Q}$ to some canonical weight matrix $\mathbf{W}$, and obtain $\tilde{\mathbf{W}} \leftarrow \mathbf{WQ}$. We can write this rotation matrix as a stack of orthonormal vectors:

$$\mathbf{Q} = \begin{bmatrix} — & \mathbf{q}_1^\top & — \\ & \cdots & \\ — & \mathbf{q}_D^\top & — \end{bmatrix} \tag{5}$$

When we recover the canonical weight matrix, we apply eigenvalue decomposition to $\tilde{\mathbf{W}}^\top \tilde{\mathbf{W}}$. This is possible as $\mathbf{W}^\top \mathbf{W}$ is defined to be diagonal. Therefore, $\mathbf{Q}$ is one solution of eigenvalue decomposition:

$$\tilde{\mathbf{W}}^\top \tilde{\mathbf{W}} = \mathbf{Q}^\top \mathbf{W}^\top \mathbf{W} \mathbf{Q} = \mathbf{Q}^\top \mathbf{\Lambda} \mathbf{Q} \tag{6}$$

However, the solution is not unique. We can write

$$\mathbf{Q}^\top \mathbf{\Lambda} \mathbf{Q} = \begin{bmatrix} | & \cdot & | \\ \mathbf{q}_1 & \cdot & \mathbf{q}_D \\ | & \cdot & | \end{bmatrix} \mathbf{\Lambda} \begin{bmatrix} — & \mathbf{q}_1^\top & — \\ \cdot & \cdot & \cdot \\ — & \mathbf{q}_D^\top & — \end{bmatrix} = \sum_d \lambda_d \mathbf{q}_d \mathbf{q}_d^\top \tag{7}$$

Changing the sign of any $\mathbf{q}_d$ will not influence the results of its outer product. Therefore, we can change the sign of each $\mathbf{q}_d$, and this will still be a valid solution to the eigenvalue decomposition. As an example, WLG, assume by eigenvalue decomposition, we obtain

$$\mathbf{Q}' = \begin{bmatrix} — & -\mathbf{q}_1^\top & — \\ — & \mathbf{q}_2^\top & — \\ & \cdots & \\ — & \mathbf{q}_D^\top & — \end{bmatrix} \tag{8}$$

We recover canonical weight matrix by

$$\tilde{\mathbf{W}} \mathbf{Q}'^\top = \mathbf{W} \mathbf{Q} \mathbf{Q}'^\top = \mathbf{W} \begin{bmatrix} — & \mathbf{q}_1^\top & — \\ & \cdots & \\ — & \mathbf{q}_D^\top & — \end{bmatrix} \begin{bmatrix} | & \cdot & | \\ -\mathbf{q}_1 & \cdot & \mathbf{q}_D \\ | & \cdot & | \end{bmatrix} = \mathbf{W} \begin{bmatrix} -1 & & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \end{bmatrix} \neq \mathbf{W} \tag{9}$$

Therefore, if we do not control the sign of each eigenvector. We cannot recover the original canonical weight matrix.

## C Algorithms

**Algorithm 1** Rotate Transformer to its Canonical Direction.

```
Input: Transformer weights with SliceGPT: $\mathbf{W}_{\text{emb}}$, $\mathbf{Q}^{(\ell)}_{\text{skip\_att}}$, $\mathbf{W}^{(\ell)}_q$, $\mathbf{W}^{(\ell)}_k$, $\mathbf{W}^{(\ell)}_v$, $\mathbf{W}^{(\ell)}_o$, $\mathbf{b}^{(\ell)}_q$, $\mathbf{b}^{(\ell)}_k$,
    $\mathbf{b}^{(\ell)}_v$, $\mathbf{b}^{(\ell)}_o$, $\mathbf{Q}^{(\ell)}_{\text{skip\_mlp}}$, $\mathbf{W}^{(\ell)}_1$, $\mathbf{W}^{(\ell)}_2$, $\mathbf{b}^{(\ell)}_1$, $\mathbf{b}^{(\ell)}_2$, $\mathbf{W}_{\text{head}}$, $\mathbf{b}_{\text{head}}$, $\ell = 1, 2, \cdots, L$;
Output: Rotated weights.
  # rotate input embeddings:
  $\mathbf{Q} \leftarrow$ Eigenvalue Decompsition($\mathbf{W}_{\text{emb}}^\top \mathbf{W}_{\text{emb}}$);
  $\mathbf{W}_{\text{emb}} \leftarrow \mathbf{W}_{\text{emb}} \mathbf{Q}$;
  for $\ell \in [1, \cdots, L]$ do
    # rotate skip connection and attention:
    $\mathbf{Q}^{(\ell)}_{\text{skip\_att}} \leftarrow \mathbf{Q}^\top \mathbf{Q}^{(\ell)}_{\text{skip\_att}}$; $\mathbf{W}^{(\ell)}_q \leftarrow \mathbf{Q}^\top \mathbf{W}^{(\ell)}_q$; $\mathbf{W}^{(\ell)}_k \leftarrow \mathbf{Q}^\top \mathbf{W}^{(\ell)}_k$; $\mathbf{W}^{(\ell)}_v \leftarrow \mathbf{Q}^\top \mathbf{W}^{(\ell)}_v$;
    # rotate attention output weight:
    $\mathbf{Q} \leftarrow$ Eigenvalue Decompsition($\mathbf{W}^{(\ell)\top}_o \mathbf{W}^{(\ell)}_o$);
    $\mathbf{W}^{(\ell)}_o \leftarrow \mathbf{W}^{(\ell)}_o \mathbf{Q}$; $\mathbf{b}^{(\ell)}_o \leftarrow \mathbf{Q}^\top \mathbf{b}^{(\ell)}_o$;
    # rotate skip connection and MLP input weight:
    $\mathbf{Q}^{(\ell)}_{\text{skip\_att}} \leftarrow \mathbf{Q}^{(\ell)}_{\text{skip\_att}} \mathbf{Q}$; $\mathbf{Q}^{(\ell)}_{\text{skip\_mlp}} \leftarrow \mathbf{Q}^\top \mathbf{Q}^{(\ell)}_{\text{skip\_mlp}}$; $\mathbf{W}^{(\ell)}_1 \leftarrow \mathbf{Q}^\top \mathbf{W}^{(\ell)}_1$;
    # rotate skip connection and MLP output weight:
    $\mathbf{Q} \leftarrow$ Eigenvalue Decompsition($\mathbf{W}^{(\ell)\top}_2 \mathbf{W}^{(\ell)}_2$);
    $\mathbf{Q}^{(\ell)}_{\text{skip\_mlp}} \leftarrow \mathbf{Q}^{(\ell)}_{\text{skip\_mlp}} \mathbf{Q}$; $\mathbf{W}^{(\ell)}_2 \leftarrow \mathbf{W}^{(\ell)}_2 \mathbf{Q}$; $\mathbf{b}^{(\ell)}_2 \leftarrow \mathbf{Q}^\top \mathbf{b}^{(\ell)}_2$;
  end for
  # rotate heads:
  $\mathbf{W}_{\text{head}} \leftarrow \mathbf{Q}^\top \mathbf{W}_{\text{head}}$;
```

**Algorithm 2** Recover rotation matrix from rotated weight.

```
Input: Rotated matrix $\mathbf{W}$, reference signs $\mathbf{s}$ (vector of $\pm 1$-s).
Output: Rotation matrix $\mathbf{Q}$:
  $\mathbf{Q} \leftarrow$ Eigenvalue Decompsition($\mathbf{W}^\top \mathbf{W}$).          ▷ rotate $\mathbf{W}^{(\ell)}_2$ to canonical direction
  for $r \in |\text{row}(\mathbf{Q})|$ do
    $\mathbf{Q}_r \leftarrow \begin{cases} \mathbf{Q}_r, & \text{if sign}(\mathbf{Q}_r\texttt{.sum()}) = \mathbf{s}_r; \\ -\mathbf{Q}_r, & \text{otherwise.} \end{cases}$          ▷ change sign of $\mathbf{Q}$
  end for
```

**Algorithm 3** Decode a rotation matrix from the current bitstream. We use **red** to represent adding bits to the bitstream; **green** for removing bits from the bitstream.

```
Input: Bitstream $\mathcal{M}$;
Output: Rotation matrix $\mathbf{Q} \in \mathbb{R}^{D \times D}$.
  $\mathbf{X} \leftarrow \mathbf{0} \in \mathbb{R}^{D \times D}$;
  Decode $D(D-1)/2$ floats from bitstream $\mathcal{M}$;
  Fill the upper triangular of $\mathbf{X}$ with these floats;
  $\mathbf{X} \leftarrow \mathbf{X} + \mathbf{X}^\top$;
  Decode $D$ floats from bitstream $\mathcal{M}$;
  Fill the diagonal of $\mathbf{X}$ with these floats;
  $\mathbf{Q}, \boldsymbol{\lambda} \leftarrow$ Eigenvalue Decomposition($\mathbf{X}$);
  $\boldsymbol{\lambda} \rightarrow \texttt{Encode\_to}(\mathcal{M})$.
```

**Algorithm 4** Encode a rotation matrix to the current bitstream. We use **red** to represent adding bits to the bitstream; **green** for removing bits from the bitstream.

```
Input: Rotation matrix $\mathbf{Q}$, Bitstream $\mathcal{M}$;
Output: Updated bitstream $\mathcal{M}$.

  $\boldsymbol{\lambda} \leftarrow \texttt{Decode\_from}(\mathcal{M})$.
  $\mathbf{X} \leftarrow \mathbf{Q}\ \text{diag}(\boldsymbol{\lambda})\ \mathbf{Q}^\top$.
  Retrieve floats in the diagonal of $\mathbf{X}$;
  Encode these $D$ floats into $\mathcal{M}$.
  Retrieve floats in the upper triangular of $\mathbf{X}$;
  Encode these $D(D-1)/2$ floats into $\mathcal{M}$.
```

**Algorithm 5** Bits-back Encoding for transformers (processed by SliceGPT). We use **red** to represent adding bits to the bitstream; **green** to represent removing bits from the bitstream.

```
Input: Transformer weights: $\mathbf{W}_{\text{emb}}$, $\mathbf{Q}^{(\ell)}_{\text{skip\_att}}$, $\mathbf{W}^{(\ell)}_q$, $\mathbf{W}^{(\ell)}_k$, $\mathbf{W}^{(\ell)}_v$, $\mathbf{W}^{(\ell)}_o$, $\mathbf{b}^{(\ell)}_q$, $\mathbf{b}^{(\ell)}_k$, $\mathbf{b}^{(\ell)}_v$, $\mathbf{b}^{(\ell)}_o$,
    $\mathbf{Q}^{(\ell)}_{\text{skip\_mlp}}$, $\mathbf{W}^{(\ell)}_1$, $\mathbf{W}^{(\ell)}_2$, $\mathbf{b}^{(\ell)}_1$, $\mathbf{b}^{(\ell)}_2$, $\mathbf{W}_{\text{head}}$, $\mathbf{b}_{\text{head}}$, $\ell = 1, 2, \cdots, L$;
Output: Binary message $\mathcal{M}$.
 1: $\mathcal{M} \leftarrow \perp$.                                  ▷ initialization empty bitstream.
 2: Rotate the transformer to its canonical direction by Algorithm 1.
 3: # encode weights with bits-back:
 4: $\mathbf{W}_{\text{emb}}, \mathbf{b}_{\text{emb}} \rightarrow$ Encode_to($\mathcal{M}$).                   ▷ encode input embeddings
 5: for $\ell \in [1, \cdots, L]$ do
 6:     $\mathbf{Q}^{(\ell)}_{\text{skip\_att}}, \mathbf{W}^{(\ell)}_q, \mathbf{W}^{(\ell)}_k, \mathbf{W}^{(\ell)}_v, \mathbf{b}^{(\ell)}_q, \mathbf{b}^{(\ell)}_k, \mathbf{b}^{(\ell)}_v \rightarrow$ Encode_to($\mathcal{M}$).
 7:     $\mathbf{Q} \leftarrow$ Decode rotation matrix from $\mathcal{M}$ by Algorithm 3.     ▷ decode a random rotation
 8:     $\mathbf{W}^{(\ell)}_o \leftarrow \mathbf{W}^{(\ell)}_o \mathbf{Q}$.
 9:     sign($\mathbf{Q}$.sum(-1)) $\rightarrow$ Encode_to($\mathcal{M}$).             ▷ encode sign of $\mathbf{Q}$ (overhead)
10:     $\mathbf{W}^{(\ell)}_o, \mathbf{b}^{(\ell)}_o, \mathbf{Q}^{(\ell)}_{\text{skip\_mlp}}, \mathbf{W}^{(\ell)}_1, \mathbf{b}^{(\ell)}_1 \rightarrow$ Encode_to($\mathcal{M}$).
11:                                       ▷ encode rotated $\mathbf{W}^{(\ell)}_o$ and other weights
12:     $\mathbf{Q} \leftarrow$ Decode rotation matrix from $\mathcal{M}$ by Algorithm 3.     ▷ decode a random rotation
13:     $\mathbf{W}^{(\ell)}_2 \leftarrow \mathbf{W}^{(\ell)}_2 \mathbf{Q}$.
14:     sign($\mathbf{Q}$.sum(-1)) $\rightarrow$ Encode_to($\mathcal{M}$).             ▷ encode sign of $\mathbf{Q}$ (overhead)
15:     $\mathbf{W}^{(\ell)}_2, \mathbf{b}^{(\ell)}_2 \rightarrow$ Encode_to($\mathcal{M}$).      ▷ encode rotated $\mathbf{W}^{(\ell)}_2$ and other weights
16: end for
17: $\mathbf{W}_{\text{head}}, \mathbf{b}_{\text{head}} \rightarrow$ Encode_to($\mathcal{M}$).                          ▷ encode heads
```

**Algorithm 6** Bits-back Decoding for transformers (processed by SliceGPT). We use **red** to represent adding bits to the bitstream; **green** to represent removing bits from the bitstream.

```
Input: Binary message $\mathcal{M}$.
Output: Transformer weights: $\mathbf{W}_{\text{emb}}$, $\mathbf{Q}^{(\ell)}_{\text{skip\_att}}$, $\mathbf{W}^{(\ell)}_q$, $\mathbf{W}^{(\ell)}_k$, $\mathbf{W}^{(\ell)}_v$, $\mathbf{W}^{(\ell)}_o$, $\mathbf{b}^{(\ell)}_q$, $\mathbf{b}^{(\ell)}_k$, $\mathbf{b}^{(\ell)}_v$, $\mathbf{b}^{(\ell)}_o$,
    $\mathbf{Q}^{(\ell)}_{\text{skip\_mlp}}$, $\mathbf{W}^{(\ell)}_1$, $\mathbf{W}^{(\ell)}_2$, $\mathbf{b}^{(\ell)}_1$, $\mathbf{b}^{(\ell)}_2$, $\mathbf{W}_{\text{head}}$, $\mathbf{b}_{\text{head}}$, $\ell = 1, 2, \cdots, L$.
 1: $\mathbf{W}_{\text{head}}, \mathbf{b}_{\text{head}} \leftarrow$ Decode_from($\mathcal{M}$).                        ▷ decode heads
 2: for $\ell \in [L, \cdots, 1]$ do
 3:     $\mathbf{W}^{(\ell)}_2, \mathbf{b}^{(\ell)}_2 \leftarrow$ Decode_from($\mathcal{M}$).     ▷ decode rotated $\mathbf{W}^{(\ell)}_2$ and other weights
 4:     $\mathbf{s} \leftarrow$ Decode_from($\mathcal{M}$).                              ▷ decode sign of $\mathbf{Q}$
 5:     $\mathbf{Q} \leftarrow$ Recover rotation matrix by Algorithm 2 from $(\mathbf{W}^{(\ell)}_2, \mathbf{s})$.
 6:     $\mathbf{W}^{(\ell)}_2 \leftarrow \mathbf{W}^{(\ell)}_2 \mathbf{Q}^\top$                        ▷ recover canonical direction
 7:     $\mathbf{Q} \rightarrow$ Encode rotation matrix to $\mathcal{M}$ by Algorithm 4.      ▷ encode the random rotation
 8:     $\mathbf{W}^{(\ell)}_o, \mathbf{b}^{(\ell)}_o, \mathbf{Q}^{(\ell)}_{\text{skip\_mlp}}, \mathbf{W}^{(\ell)}_1, \mathbf{b}^{(\ell)}_1 \leftarrow$ Decode_from($\mathcal{M}$).
 9:                                       ▷ decode rotated $\mathbf{W}^{(\ell)}_o$ and other weights
10:     $\mathbf{s} \leftarrow$ Decode_from($\mathcal{M}$).                              ▷ decode sign of $\mathbf{Q}$
11:     $\mathbf{Q} \leftarrow$ Recover rotation matrix by Algorithm 2 from $(\mathbf{W}^{(\ell)}_o, \mathbf{s})$.
12:     $\mathbf{W}^{(\ell)}_o \leftarrow \mathbf{W}^{(\ell)}_o \mathbf{Q}^\top$                        ▷ recover canonical direction
13:     $\mathbf{Q} \rightarrow$ Encode rotation matrix to $\mathcal{M}$ by Algorithm 4.      ▷ encode the random rotation
14:     $\mathbf{Q}^{(\ell)}_{\text{skip\_att}}, \mathbf{W}^{(\ell)}_q, \mathbf{W}^{(\ell)}_k, \mathbf{W}^{(\ell)}_v, \mathbf{b}^{(\ell)}_q, \mathbf{b}^{(\ell)}_k, \mathbf{b}^{(\ell)}_v \rightarrow$ Decode_from($\mathcal{M}$).
15: end for
16: $\mathbf{W}_{\text{emb}}, \mathbf{b}_{\text{emb}} \leftarrow$ Decode_from($\mathcal{M}$).                  ▷ decode input embeddings
```

## D Analysis of the Codelength

Here, we analyze the codelength reduction achieved by our proposed approach from a practical standpoint. For simplicity's sake, we assume there is no bias vector in our transformer architecture. This is a reasonable assumption, as some modern architectures like LLaMA (Touvron et al., 2023) follow similar designs. Additionally, we assume the transformer has no output head or embedding layer. This assumption can be interpreted as modeling an extremely deep transformer, where the effects of the head and embedding layers become negligible. However, it is important to note that this is not a realistic assumption in practical scenarios. This is the main reason for the discrepancy between our analysis in this section and the results we present in Section 4.

In one transformer block, as shown in Figure 1b, there exist eight matrices after SliceGPT, including six sliced weight matrices and two skip connection matrices. If the slicing rate is $s\%$ and the weights are stored in `float16`, the total codelength (in bits) can be expressed as:

$$(\underbrace{6 \cdot r\%D^2}_{\text{6 weight matrices}} + \underbrace{2 \cdot (r\%D)^2}_{\text{2 skip connection}}) \cdot 16 \tag{10}$$

where we denote $r\% = 1 - s\%$ as the remaining rate after slicing. Using bits-back, we decode two rotation matrices from the bitstream during encoding, leading to a reduction in codelength by:

$$\left(2 \cdot \frac{(r\%D)(r\%D-1)}{2}\right) \cdot 16 = (r\%D) \cdot (r\%D-1) \cdot 16 \tag{11}$$

We disregard the overhead from storing the signs of the eigenvectors (line 9 in Algorithm 5) and the correction codes (discussed in Section 3.3), as these contributions are negligible.

Thus, the overall reduction in codelength is:

$$(r\%D) \cdot (r\%D-1)/(6 \cdot r\%D^2 + 2 \cdot (r\%D)^2) \approx r\%/(6 + 2r\%) \tag{12}$$

For a slice rate of $s\% = 20 - 30\%$, this results in approximately a $10\%$ reduction in codelength.