UCLouvain

epl

École polytechnique de Louvain

# Low-rank matrix factorization for compressing Transformers

Author: **Nathan Monhart**
Supervisor: **Estelle Massart**
Readers: **Benoît Legat, Jana Jovcheva**
Academic year 2024–2025
Master [120] in Mathematical Engineering

# Abstract

A compression algorithm framework is developed in this master thesis to compress neural network models. Within this framework, weight matrices that are suitable for compression are identified and then reduced using low-rank approximation methods. Accordingly, a key component of the algorithm is determining the truncation rank to apply to the selected weight matrices. The framework is evaluated on Transformer models, and it is found that the truncation rank configuration chosen by the algorithm yields better compression results than baseline low-rank compression methods. Various scenarios are examined, including the compression of fine-tuned models and pre-trained models, both when samples from the target datasets are available and when they are not.

**Keywords** : Transformers · Neural network compression· SVD · Low-rank approximation · Matrix rank · Fine-tuning

## Acknowledgements

I would particularly like to thank my supervisor Estelle Massart for the time she dedicated to my work and for her many suggestions.

This master's thesis marks the end of my academic journey, so I would like to take this opportunity to thank the friends and family who have accompanied me over the years.

# Contents


**1 Introduction** **4**

**2 Background** **5**
- 2.1 Low-rank factorization . . . 5
- 2.2 The general Neural Network Model . . . 6
- 2.3 The Transformer architecture . . . 7
  - 2.3.1 BERT . . . 12
  - 2.3.2 GPT-2 . . . 13
  - 2.3.3 RoBERTa . . . 13
- 2.4 GLUE Benchmark . . . 14

**3 Literature review** **17**

**4 Proposed compression algorithm framework** **22**
- 4.1 Sorting criteria and rank approximation . . . 23
- 4.2 Output features . . . 27
- 4.3 Error measure . . . 27
- 4.4 Proxy dataset . . . 28
- 4.5 Matrix factorization method . . . 29
  - 4.5.1 Singular Value Decomposition (SVD) . . . 29
  - 4.5.2 classical matrix factorization methods . . . 30
  - 4.5.3 Atomic Feature Mimicking (AFM) . . . 32
- 4.6 Other hyper-parameters . . . 33

**5 Numerical results** **34**
- 5.1 Rank structure of Transformer models . . . 34
- 5.2 Fine-tuned model compression . . . 37
  - 5.2.1 Task-specific proxy . . . 38
  - 5.2.2 Random proxy . . . 41
- 5.3 Pre-trained model compression . . . 43
  - 5.3.1 Task-specific proxy . . . 43
  - 5.3.2 Random proxy . . . 46

**6 Conclusion** **49**

**7 Appendix A** **50**

**8 Appendix B** **52**

# 1 Introduction

Transformer models are everywhere, serving as the foundation for generative AI systems such as chatbots and image generators. These models are typically very large, making both training and everyday use computationally intensive, thus costly, and environmentally impactful due to datacenter emissions. Consequently, compression of AI models can provide significant benefits by reducing model size while preserving performance.

In this study, compression is achieved by exploiting the rank property of the matrices that constitute Transformer models. The rank quantifies the redundancy of information within a matrix. Matrices with low rank contain a lot of redundant information and are therefore prime candidates for compression. A modular framework for compressing Neural Network models in various contexts is explored and evaluated on Transformer models. The framework's design allows it to be applied beyond the scenarios examined in this work, enabling further extensions and adaptations. The study begins by reviewing key matrix properties related to low-rank factorization and by presenting the general transformer architecture, followed by a description of the specific models and benchmark datasets used here, and a literature review. The next section presents a modular compression framework, detailing its hyper-parameters and how they are chosen for various scenarios. Finally, the numerical results are shown and discussed in comparison to baseline compression methods. The result is a compression framework that works well and outperforms baseline methods on several tasks.

The code repository with scripts to reproduce the results is available at: https://drive.google.com/drive/folders/1AkpjavGN36AziuIbmQGZkZFRAy0F40li?usp=sharing

# 2 Background

## 2.1 Low-rank factorization

*Fundamental results and definitions related to low-rank matrix factorization are recalled here, and at the same time, some of the notations that will be used later are introduced.*

### Singular Value Decomposition (SVD)

The Singular Value Decomposition (SVD) of a matrix decomposes it into the product of 3 matrices.

**Theorem 1** (Singular Value Decomposition (SVD) [12])**.** *If $A$ is a real $m \times n$ matrix, then there exist orthogonal matrices $U = [u_1 \mid \cdots \mid u_m] \in \mathbb{R}^{m\times m}$ and $V = [v_1 \mid \cdots \mid v_n] \in \mathbb{R}^{n\times n}$ such that*

$$A = U\Sigma V^\top, \quad \text{where } \Sigma = \operatorname{diag}(\sigma_1, \ldots, \sigma_p) \in \mathbb{R}^{m\times n}, p = \min\{m, n\}, \tag{1}$$

*$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_p \geq 0$ are the singular values of $A$, and $u_i$, $v_i$ are the corresponding left and right singular vectors.*

The SVD of a matrix $A \in \mathbb{R}^{m\times n}$ is typically computed in two steps: first, reducing $A$ to a bidiagonal matrix (costing $O(mn^2)$ flops), and then computing its SVD (requiring $O(n)$ iterations, each of $O(n)$ flops). The total cost is $O(mn^2)$, or $O(n^3)$ for square matrices [21].

### Rank of a matrix

The column rank of a matrix $A \in \mathbb{R}^{m\times n}$ is the dimension of the subspace spanned by its columns. Similarly, the row rank is the dimension of the subspace spanned by its rows. The row rank and the column rank of a matrix $A$ are equal [2] (we will denote both terms as *rank*). The rank is also the number of non-zero singular values of a matrix [12]. We say a matrix $A \in \mathbb{R}^{m\times n}$ is *full rank* when $\operatorname{rank}(A) = \min\{m, n\}$. In practice, the rank indicates how redundant the information inside a matrix is (i.e. a low-rank matrix contains highly redundant information).

### SVD as best low-rank approximation

The best rank-k approximation of $A \in \mathbb{R}^{m\times n}$ is given by its truncated SVD.

**Theorem 2** (Eckart–Young Theorem [12])**.** *Let $A \in \mathbb{R}^{m\times n}$ have rank $r$, and suppose $k < r$. Let*

$$A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^\top, \tag{2}$$

*where $\sigma_i$ are the singular values of $A$ and $u_i$, $v_i$ are the corresponding left and right singular vectors. Then*

$$\min_{\operatorname{rank}(B)=k} \|A - B\|_2 = \|A - A_k\|_2 = \sigma_{k+1} \tag{3}$$

*The matrix $A_k$ is the closest rank-k matrix to $A$ in both the spectral and Frobenius norms.*

In practice, $A_k$ is often expressed as the product of three truncated matrices $U_k \in \mathbb{R}^{m\times k}$, $\Sigma_k \in \mathbb{R}^{k\times k}$, $V_k \in \mathbb{R}^{n\times k}$ where $V_k, U_k, V_k$ are respectively obtained by keeping the first $k$ columns of $U, \Sigma, V$ and such that $A_k = U_k\Sigma_k V_k^T$. In the following, $k$ will be denoted as the *truncation rank.*

## 2.2 The general Neural Network Model

*The concepts and notations used to describe general Neural Network models are introduced here.*

Let's consider some data $\{\mathbf{X}, \mathbf{Y}\}$, where $\mathbf{X} \in \mathbb{R}^{n\times d_0}$ contains $n$ input vectors of dimension $d_0$, and $\mathbf{Y} \in \mathbb{R}^{n\times c}$ contains their corresponding $c$-dimensional targets.

A neural network model with $L$ layers is denoted by $M_W^L$, where $W = \{W_k \in \mathbb{R}^{m_k\times n_k}\}_{k=1}^K$ is the set of weight matrices. The weights are grouped by layer, with $W^{(\ell)}$ containing all weights used in the $\ell$-th layer.

$$W = \bigcup_{\ell=1}^{L} W^{(\ell)}$$

The output of layer $\ell$ is denoted $\mathbf{h}^{(\ell)} \in \mathbb{R}^{n\times d_\ell}$, with $\mathbf{h}^{(0)} = \mathbf{X}$. This output is generally computed as:

$$\mathbf{h}^{(\ell)} = f_\ell\left(\mathbf{h}^{(\ell-1)}; W^{(\ell)}\right)$$

The *architecture* of a neural network can be conceptualized through $W$ and a sequence of functions $f_\ell : \mathbb{R}^{n\times d_{\ell-1}} \to \mathbb{R}^{n\times d_\ell}$, with $\ell = 1, \ldots, L$. Each $f_\ell$ specifies both the organization of weights $W^{(\ell)}$ and the operations that transforms the previous layer's output $\mathbf{h}^{(\ell-1)} \in \mathbb{R}^{n\times d_{\ell-1}}$ into the current layer's output $\mathbf{h}^{(\ell)} \in \mathbb{R}^{n\times d_\ell}$. Therefore, the final prediction after $L$ layers is :

$$\hat{\mathbf{Y}} = M_W^L(\mathbf{X}) = \mathbf{h}^{(L)}$$

The weight matrices $W$ are learned by minimizing a loss function $\mathcal{L}(\cdot)$ that measures the difference between predictions and targets.

$$W^* = \arg\min_W \mathcal{L}\left(\mathbf{Y}, M_W^L(\mathbf{X})\right)$$

For low-rank compression, let $R = (R_1, R_2, \ldots, R_K)$ be a *rank configuration* where each $R_k \leq \min(m_k, n_k)$. The set of compressed weight matrices $W_R$ is obtained by truncating each $W_k$ to rank $R_k$. The resulting compressed network is denoted $M_{W_R}^L$, and its weights are optimized via

$$W_R^* = \arg\min_W \mathcal{L}\left(\mathbf{Y}, M_{W_R}^L(\mathbf{X})\right)$$

In practice, we approximate $W_R^*$ rather than compute it exactly.

**Note** : In the following sections **Y** will be reserved to denote model outputs instead of target labels.

## 2.3 The Transformer architecture

*The architecture of the models used in this work is described here. They all belong to the same class of models, namely Transformer models. For this reason, a breakdown of the transformer's general architecture, as well as the specific architectures used for benchmarking, is provided here.*

Transformer models were first introduced in 2017 by *Ashish Vaswani* and its team at Google Brain with the well-known paper *attention is all you need* [38]. Transformers are generally used in Natural Language Processing (NLP) to process or generate textual data. These models leverage an efficient mechanism known as multi-head attention to parallelize computation and highlight the context of the words in their respective sentences. An illustration of the general transformer architecture is given in Figure 1.

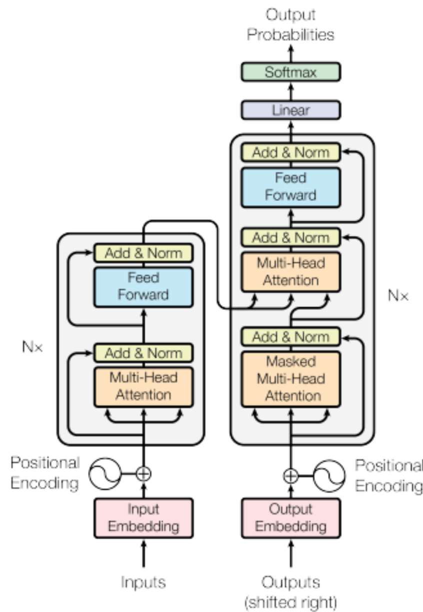



Figure 1: The Transformer architecture [38]. The encoder is on the left and the decoder is on the right. $N$ is the number of layers (denoted by $L$ in this work).

We can see that the Transformer is composed of different architectural blocks. These blocks will be described in the following sections. All the Transformer model implementations used in this work come from the HuggingFace's Transformers library [40].

**Word Embedding.** Before a Transformer can process raw text, the text must first be broken into discrete units called tokens. A tokenizer is the component that performs this step. The tokenizer takes a sentence and splits it into tokens, which may be words, subwords, or punctuation marks, depending on the specific tokenization scheme. Each token is then assigned to a unique integer index in a fixed vocabulary of size $n_{\text{vocab}}$. In practice, one often represents each token index as a one-hot vector of length $n_{\text{vocab}}$, so that a batch of $B$ sentences each of length *seq* can be viewed as a tensor $\mathbf{X} \in \mathbb{R}^{B \times \text{seq} \times n_{\text{vocab}}}$. The input embedding layer projects these sparse one-hot encodings into a dense, continuous feature space of dimension $d_{\text{model}}$. Concretely, it replaces each token's one-hot vector by the corresponding row of the word embedding layer $W \in \mathbb{R}^{n_{\text{vocab}} \times d_{\text{model}}}$. After this, the model's input becomes a tensor $\mathbf{X} \in \mathbb{R}^{B \times \text{seq} \times d_{\text{model}}}$, where each token is represented by a learned $d_{\text{model}}$-dimensional vector representing the *feature dimension*. For the following, we will set $n = \text{B} \times \text{seq}$, so that the input tensor becomes an input matrix denoted by $\mathbf{X} \in \mathbb{R}^{n \times n_{\text{vocab}}}$ and the output of the word embedding by $\mathbf{X} \in \mathbb{R}^{n \times d_{\text{model}}}$. More generally, $n$ will denote the number of tokens of any feature matrix.

**Positional Encoding.** The problem with this text input format is that the model has no information about the ordering of the words within the input sequence. To address this, a special vector called positional encoding is added to each word's embedding. This vector provides information about the position of the word in the sequence. Unlike word embeddings, positional encodings are not learned during training. Instead, they are computed using a fixed formula based on sine and cosine functions:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right)$$
$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right)$$

where *pos* is the position of the word in the sequence, $i$ is the index of the elements of its embedding vector in the word embedding layer and $d_{model}$ is the size of the word embeddings.

**Self-Attention**. The input of the self-attention mechanism is a set of 3 matrices $Q \in \mathbb{R}^{n \times d_k}$, $K \in \mathbb{R}^{n \times d_k}$, $V \in \mathbb{R}^{n \times d_v}$ called the query, key and value matrices. These input matrices can come from the outputs of some architectural blocks of the transformer. An *attention score* is computed by taking the dot product between the query matrix $Q \in \mathbb{R}^{n \times d_k}$ and the key matrix $K \in \mathbb{R}^{n \times d_k}$. The resulting attention score is then scaled by $\sqrt{d_k}$, to stabilize gradients during training. The scaled score is passed through a softmax function (e.g. Eq 4) to normalize it into a probability distribution. Finally, the normalized attention

score is multiplied by the value matrix $V \in \mathbb{R}^{n \times d_v}$ to produce the output of the self-attention layer. Mathematically, this process is expressed as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

When the query, key and value matrices are just copies of some token embedding sequence $x \in \mathbb{R}^{n \times d}$, with $d_k = d_v = d_{\text{model}}$, the output of the self-attention layer can be interpreted as a refined version of the input sequence, where the features of tokens that are strongly related to other tokens in the sequence are accentuated. This allows the model to capture contextual relationships within the input sequence. In other words, tokens that are more relevant (i.e., more strongly related to other words in the sequence) will have their features highlighted, while less relevant tokens will have their features weakened.

If $d_k = d_v = d_{\text{model}}$, the computational complexity of the attention mechanism is $O(n^2 d_{\text{model}}) = O(n^2)$ and is thus quadratic in the token embedding sequence length.

**Multi-Head Attention.**

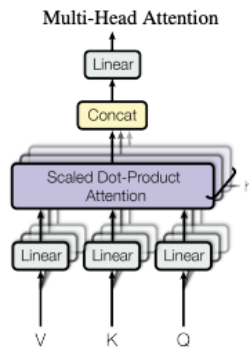



Figure 2: Multi-Head Attention block [38]

A Multi-Head Attention layer extends the Self-Attention mechanism by using multiple attention blocks in parallel. Each attention block is called an **attention head**. While a Self-Attention layer computes attention scores for a single set of Query ($Q$), Key ($K$), and Value ($V$) matrices, a Multi-Head Attention layer replicates this process $h$ times (where $h$ is a hyperparameter). In addition, each head, denoted by $\text{head}_i$, has its own set of learned weight matrices $W_Q^i \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_K^i \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_V^i \in \mathbb{R}^{d_{\text{model}} \times d_v}$, where $i \in \{1, \ldots, h\}$, that linearly project the input $Q, K, V \in \mathbb{R}^{d_{\text{model}} \times d_{\text{model}}}$.

The outputs of these $h$ heads are concatenated and then projected using an additional learned weight matrix $W_O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$. Mathematically, this is expressed as:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$
$$\text{where} \quad \text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

To maintain computational efficiency, the feature dimension of the Query, Key, and Value matrices in each head is typically divided by $h$ compared to their dimension in a single Self-Attention layer (i.e. $d_k = d_v = d_{\text{model}}/h$). This ensures that the overall computational cost of Multi-Head Attention remains similar to that of Self-Attention, while allowing the model to capture diverse patterns and relationships in the data through multiple attention heads.

**Masked Multi-Head Attention.** The Masked Multi-Head Attention is very similar to the Multi-Head Attention layer but here, the attention matrix is forced to be lower-diagonal (upper-diagonal values are set to 0) so that the model can not see in the future while predicting words during training.

**Add & norm.** Add & Norm layers are quite simple, they rely on 2 operations. The first operation is the addition : for instance, the input of a layer is added to its output. This addition part is often denoted as a *skip connection*. One of its role is to help the gradients to backpropagate during training, avoiding gradient vanishing problems [4]. The second operation is the normalization, where the output of a layer is normalized across the feature dimension.

**Feed Forward.** A feed forward layer is just a regular linear weight matrix that multiplies an input vector and adds a bias term.

**Encoder and Decoder**. The Encoder and the Decoder are the core building blocks of a transformer model. The encoder is the left block in Figure 1 and the decoder is the right block. In a transformer, an encoder is composed of a Multi-Head Attention layer with a Feed Forward layer, each of them connected with an Add & Norm layer. In the original paper [38], $L = 6$ decoder layers are stacked together. Intuitively, the goal of the encoder is to produce a low-dimensional representation of the input. A decoder is composed of a Masked Multi-Head Attention layer, a Multi-Head Attention layer that takes the output of the encoder as Query and Key matrices and the output of the Masked Multi-Head Attention layer as a Value matrix. The final block of a decoder is generally a Feed Forward layer and each blocks are connected with an Add & Norm layers. In the original paper [38], $L = 6$ decoder layers are stacked together. The goal of the decoder is to produce a word from the low-dimensional representation of the encoder and from the word it had predicted at the previous step (the words are predicted one by one). Therefore, the output of the encoder can be computed

once but a forward pass through the decoder has to be computed each time a word is predicted.

**Classification Head.** When using a Transformer model for task-specific applications like classification, additional layers are often appended to the architecture. For classification tasks, the model must produce a vector of classification scores for each class. Suppose the final layer of the Transformer outputs a matrix $Y \in \mathbb{R}^{n \times d_{\text{model}}}$. To classify inputs into $c$ classes, we add a classification head on top of the Transformer, which is a learnable weight matrix $W_{\text{cls}} \in \mathbb{R}^{d_{\text{model}} \times c}$. This matrix projects the Transformer's output to a matrix $Y \in \mathbb{R}^{n \times c}$, containing raw scores (i.e. *logits*) for each class. Note that $n$ can also represent the number of input sentences instead of the number of input tokens.

The logits are then passed through a softmax layer:

$$\text{softmax}(z)_{ij} = \frac{e^{z_{ij}}}{\sum_{k=1}^{c} e^{z_{ik}}}, \tag{4}$$

where $z_{ij} = Y_{i,j}$. The softmax transforms each row of the matrix into a probability distribution, ensuring the sum of probabilities for each input equals 1. The resulting element $(i, j)$ represents the probability that the $i$-th input belongs to class $j$.

**Practical considerations.** In order to make the connection with the notations introduced in section 2.2, each layers $\ell$ of a Transformer is an encoder and/or a decoder $f_\ell$ with its set of weight matrices $W^{(\ell)}$. These layers contain multi-head attention mechanisms with key, query, value, and feed forward weight matrices. The word embedding and the classification head are also considered as layers but they are respectively replicated once before and after the $L - 2$ encoder and/or decoder layers. We also assume that the feature dimension is constant across layers : $d_\ell = d_{\text{model}}$, with $\ell = 1, \ldots, L$.

In practice, we consider two types of Transformer models: pretrained models and classification models. Each pretrained model consists of two modules: the embedding module, which includes word embedding and position embedding layers, and the backbone module, which contains multi-head attention mechanisms and feed-forward weight matrices. Pretrained models are trained on very general data to learn efficient text representations but are not fine-tuned on any specific tasks. Classification models share this structure but include an additional classification head on top of their backbone module, which is specific to a given task. The input to a Transformer model is normally processed by the embedding module, although it can be bypassed so that the input is fed directly to the backbone. Similarly, the output of classification models may be taken either after the backbone module or after the classification head. In later sections, pretrained models will be denoted by $M^L_{W,\text{Pre}}$ and classification models by $M^L_{W,\text{Cls}}$. The notation $M^L_W$ serves as a general representation for any model

when no specific type is specified (i.e. not limited to transformer architectures, and including but not limited to pretrained or classification models). Figure 3 summarizes the structure of the modules.

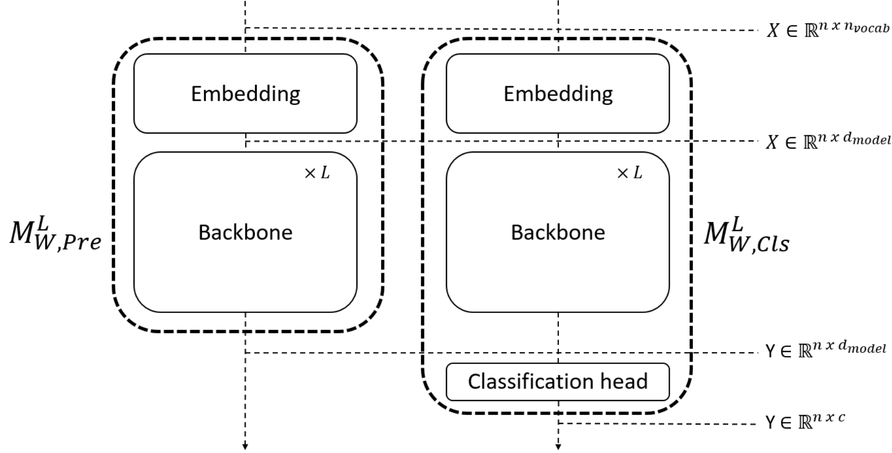



Figure 3: Left : pretrained model $M^L_{W,\text{Pre}}$ with embedding module followed by L backbone layers. Right : classification model $M^L_{W,\text{Cls}}$, with a classification head after the backbone layers. The input of the models can be fed before the embedding or before the backbone and the outputs can be taken after the backbone or after the classification head (for the classification model).

### 2.3.1 BERT

BERT-base is the smaller version of the BERT model [10], introduced by Google in 2018. It follows an *encoder-only* architecture, meaning it is composed entirely of Transformer encoder layers and does not include any decoder component. Specifically, BERT-base stacks $L = 12$ identical encoder blocks, each operating on token representations of feature dimension $d_{\text{model}} = 768$. Within each encoder block, the model applies multi-head self-attention without any masking, which allows every token to attend to all others in the sequence. This absence of masking is what enables bidirectionality (i.e. each token's representation can incorporate contextual information from both its left and right surroundings, rather than being limited to a single direction). In total, BERT-base comprises approximately 110 million trainable parameters. The dimension and relative importance of each architectural blocks are summarized in Table 3. The architecture is divided into 3 approximately equal parts : the word embedding layer, the attention mechanism (query, key, value and projection matrix) and two feed forward layers. The model also contains Add and norm modules, along with skip connections but they don't contribute significantly to the total number of parameters.

| Architecture blocks | Number of parameters | (%) |
|---|---|---|
| Word Embeddings | $30522 \times 768$ | 21.4% |
| Positional Encoding | $512 \times 768$ | 0.4% |
| Query | $12 \times 768 \times 768$ | 6.5% |
| Key | $12 \times 768 \times 768$ | 6.5% |
| Value | $12 \times 768 \times 768$ | 6.5% |
| Attention projection | $12 \times 768 \times 768$ | 6.5% |
| Feed Forward | $12 \times 768 \times 3072$ | 25.9% |
| Feed Forward | $12 \times 3072 \times 768$ | 25.9% |

Table 1: Weights of the main architectural block of the BERT-base Transformer

### 2.3.2 GPT-2

GPT-2 [33] is a Transformer-based language model developed by OpenAI and released in 2019. In contrast to BERT, GPT-2 follows a decoder-only architecture, meaning it consists exclusively of Transformer decoder blocks and does not include any encoder component. The version commonly referred to as GPT-2-small includes $N = 12$ stacked decoder blocks, each operating on token representations of feature dimension $d_{\text{model}} = 768$. Each decoder block uses masked multi-head self-attention to ensure that predictions for a given token can only depend on preceding tokens in the sequence. This masking enforces an autoregressive structure, where the model generates text one token at a time by conditioning only on the left context. Overall, GPT-2-small contains approximately 117 million trainable parameters, distributed across its attention mechanisms, feed-forward layers, and token embeddings.

| Architecture blocks | Number of parameters | (%) |
|---|---|---|
| Word Embeddings | $50257 \times 768$ | 31.0% |
| Positional Encoding | $1024 \times 768$ | 0.1% |
| (Query, Key, Value) | $12 \times 2304 \times 768$ | 17.1% |
| Attention projection | $12 \times 768 \times 768$ | 5.7% |
| Feed Forward | $12 \times 768 \times 3072$ | 22.8% |
| Feed Forward | $12 \times 3072 \times 768$ | 22.8% |
| Output Feed Forward | $768 \times 50257$ | 31.0% |

Table 2: Weight of the main architectural blocks of the GPT-2-small Transformer. The Output Feed Forward block reuse the same embedding matrix as the input.

### 2.3.3 RoBERTa

RoBERTa-base is the base variant of the RoBERTa model [26], introduced by Facebook AI in 2019. Its Transformer architecture is identical to that of BERT-base, it consists of $N = 12$ stacked encoder blocks, each with a feature dimension $d_{\text{model}} = 768$, and 12 attention heads. The only architectural difference lies in

the vocabulary size (50 265 tokens, whereas BERT-base uses 30 522 tokens). This larger vocabulary directly increases the size of the input embedding matrix, accounting for RoBERTa-base's approximately 125 million parameters compared to BERT-base's roughly 110 million.

| **Architecture blocks** | **Number of parameters** | **(%)** |
|---|---|---|
| Word Embeddings | $50265 \times 768$ | 31.0% |
| Positional Encoding | $514 \times 768$ | 0.3% |
| Query | $12 \times 768 \times 768$ | 5.7% |
| Key | $12 \times 768 \times 768$ | 5.7% |
| Value | $12 \times 768 \times 768$ | 5.7% |
| Attention projection | $12 \times 768 \times 768$ | 5.7% |
| Feed Forward | $12 \times 768 \times 3072$ | 22.7% |
| Feed Forward | $12 \times 3072 \times 768$ | 22.7% |

Table 3: Weights of the main architectural block of the RoBERTa-base Transformer

## 2.4 GLUE Benchmark

To compare transformer models, we benchmark them using standard datasets, and one of the most used dataset collection is the GLUE benchmark [39]. GLUE consists of nine datasets that cover a variety of NLP tasks. Its goal is to verify that a model performs well across different tasks and can share knowledge between them. The datasets are relatively small, which means models must learn to represent language effectively with only a few examples.

The first group of tasks in GLUE involves single sentences where the model assigns a label to each input. For example, the *acceptability* task asks whether a sentence is grammatically correct, and the *sentiment* task requires the model to decide whether a sentence expresses a positive or negative sentiment (Table 4). Another set of tasks measures how well the model can compare pairs of sentences. In these similarity and paraphrase tasks, the model either scores the sentences on a scale from one to five based on their meaning or determines if the two sentences convey the same information (Table 5). The final group of tasks focuses on inference. Here, the model reads two sentences (a premise and a hypothesis) and predicts whether the premise entails, contradicts, or is neutral with respect to the hypothesis. A related task checks if a sentence remains logically entailed when a pronoun is replaced (Table 6).

Different metrics are used to evaluate the performance of transformers on the GLUE benchmark. In our case, we will only use CoLA, MRPC, RTE, QNLI and WNLI datasets and their corresponding metrics (i.e. *Accuracy*, *F1-score* and *Matthews correlation*).

- In general, the main metric used is Accuracy [32], which is defined as follows:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{5}$$

In this formula, TP refers to true positives, which are the instances where the model correctly predicts the positive class. TN refers to true negatives, which are the instances where the model correctly predicts the negative class. FP refers to false positives, where the model incorrectly predicts the positive class when it is actually negative. FN refers to false negatives, where the model incorrectly predicts the negative class when it is actually positive.

- The F1-score is the harmonic mean of precision and recall. *Precision* measures the proportion of correctly identified positive instances among all predicted positive instances, while *recall* (also known as sensitivity) measures the proportion of actual positive instances correctly identified by the model [36]. These are defined as:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{6}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{7}$$

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} = \frac{2 \cdot \text{TP}}{2 \cdot \text{TP} + \text{FP} + \text{FN}} \tag{8}$$

The F1-score ranges from 0 (worst) to 1 (perfect) and is particularly useful for imbalanced datasets.

- The Matthews Correlation Coefficient (MCC) evaluates binary classifications by considering all four confusion matrix categories (TP, TN, FP, FN). It is defined as [8]:

$$\text{MCC} = \frac{\text{TP} \cdot \text{TN} - \text{FP} \cdot \text{FN}}{\sqrt{(\text{TP} + \text{FP}) \cdot (\text{TP} + \text{FN}) \cdot (\text{TN} + \text{FP}) \cdot (\text{TN} + \text{FN})}} \tag{9}$$

It ranges from $-1$ (total disagreement) to $+1$ (perfect prediction), with 0 equivalent to random guessing. MCC is known to be robust to class imbalance.

**Single-Sentence Tasks**

| Corpus | Train | Test | Task | Metrics | Domain |
|---|---|---|---|---|---|
| CoLA | 8.5k | 1k | acceptability | Matthews corr. | misc. |
| SST-2 | 67k | 1.8k | sentiment | acc. | movie reviews |

Table 4: Single-sentence tasks datasets from the GLUE benchmark

**Similarity and Paraphrase Tasks**

| Corpus | Train | Test | Task | Metrics | Domain |
|---|---|---|---|---|---|
| MRPC | 3.7k | 1.7k | paraphrase | acc./F1 | news |
| STS-B | 7k | 1.4k | sentence similarity | Pearson/Spearman corr. | misc. |
| QQP | 364k | 391k | paraphrase | acc./F1 | social QA questions |

Table 5: Similarity and paraphrase tasks datasets from the GLUE benchmark

**Inference Tasks**

| Corpus | Train | Test | Task | Metrics | Domain |
|---|---|---|---|---|---|
| MNLI | 393k | 20k | NLI | matched/mismatched acc. | misc. |
| QNLI | 105k | 5.4k | QA/NLI | acc. | Wikipedia |
| RTE | 2.5k | 3k | NLI | acc. | news, Wikipedia |
| WNLI | 634 | 146 | coreference/NLI | acc. | fiction books |

Table 6: Inference tasks datasets from the GLUE benchmark

# 3 Literature review

*Large neural networks and Transformers are typically compressed using pruning, quantization, and knowledge distillation to reduce computational costs while preserving performance. However, there are other (less common) methods which can also deliver excellent compression results. In this section, we will focus on low-rank factorization methods and on the optimization of the attention mechanisms in Transformers.*

**Pruning**. Pruning is a technique aimed at reducing the size and complexity of neural networks by removing parameters that contribute minimally to the model's output. This can involve eliminating weights with small magnitudes, entire neurons, or even attention heads in transformer architectures. The idea is to find and remove parts that have little effect on performance, simplifying the model and making it more efficient. For instance, the Lottery Ticket Hypothesis [7, 11] states that within a dense, randomly-initialized network, there exists a sparse sub-network (a "winning ticket") that can be trained in isolation to achieve comparable accuracy to the original model. Additionally, studies have shown that in transformer models, a significant number of attention heads can be pruned without substantial loss in performance, indicating redundancy in multi-head attention mechanisms [30].

**Quantization**. Quantization involves reducing the numerical precision of a model's parameters and activations, typically converting 32-bit floating-point numbers to lower-bit representations like 8-bit integers. This reduces the model's memory requirements and speeds up inference, making it ideal for deployment on devices with limited ressources. However, quantization can lead to a drop in model accuracy. To mitigate this, techniques such as quantization-aware training are employed, where the model is trained with quantization effects simulated during the training process, allowing it to maintain performance [17].

**Knowledge Distillation**. Knowledge Distillation is a method where a smaller, simpler model (the student) is trained to replicate the behavior of a larger, more complex model (the teacher). The student learns not only from the hard labels, which are the single correct outputs, but also from the soft outputs, which are the full probability distributions that the teacher assigns to every class. These soft outputs reveal how the teacher views the similarity between classes (e.g. showing that one error is far more likely than another) which cannot be captured by hard labels alone. By matching these richer targets, the student is guided toward the nuanced decision boundaries learned by the teacher, leading to better generalization and greater efficiency. This concept was introduced by Hinton et al., who showed that distilling knowledge in this way produces a compact model whose performance nearly matches that of its larger counterpart [14].

**Low-rank approximation**. Another compression method is Low-rank ap-

proximation. It offers a general method for compressing neural networks of various architectures. The approach decomposes a weight matrix $W \in \mathbb{R}^{n\times d}$ into smaller matrices $A \in \mathbb{R}^{n\times r}$ and $B \in \mathbb{R}^{r\times d}$ such that $W \approx AB$. Compression is achieved when the total parameters $(nr + rd)$ are fewer than the original $nd$, requiring $r < nd/(n+d)$. [37] demonstrated the effectiveness of this technique for compressing deep neural networks in speech recognition and language modeling applications. [31] extended low-rank approximation to pre-trained Transformer models by first initializing each layer's weights using SVD decomposition and followed by fine-tuning with a composite loss function. This composite loss function combines three components: standard cross-entropy loss for task performance, logit distillation to match the original model's outputs, and feature distillation that aligns intermediate layer representations between compressed and original models. A key limitation of both approaches is the lack of a systematic method for determining the optimal rank configuration across network layers, leaving this critical aspect of compression to manual selection. Some existing methods adress this problem.

**Heuristic methods for optimal rank configuration**. Some low-rank approximation methods simply factorize the model layers with a uniform truncation rank. However, the layers are generally not as important as one another. We therefore need to find methods that automatically determine the truncation rank to be applied to each layer (i.e. the *rank configuration*). Methods for determining the rank configuration of a given model are generally heuristic, and vary substantially from paper to paper. But what they have in common is that they formulate the compression problem as a combinatorial optimization problem, the aim of which is to find the rank configuration that maximizes/minimizes a certain objective function while respecting certain constraints. Some of these methods are presented below.

- The approach introduced in [15] is founded on an alternating sequence of compression and learning phases. In the compression phase, the singular value decomposition of each layer's weight matrix is computed and a low-rank approximation is selected by solving

$$r_k^* = \min_{r=0,\dots,R_k} \left[\lambda\, C_k(r) + \tfrac{\mu}{2} \sum_{i=r+1}^{\min(a_k,b_k)} s_i^2\right]$$

  where $k$ represents the index of the layer being compressed, $a_k, b_k$ are the dimensions of the weight matrix and its discarded singular values $s_{r+1}, \dots$ quantify the approximation error. $C_k(r) = \alpha_k\, r$ represents the resource cost with $\alpha_k = a_k + b_k$, and $\lambda, \mu$ control the trade-off. In the learning phase, the original loss augmented by a quadratic penalty enforcing proximity to the low-rank factors is minimized by gradient descent on the full-rank weights. These two phases are alternated until convergence, resulting in jointly optimized layer ranks and weight parameters.

- The method proposed by [18] addresses how to optimally select ranks for all layers jointly rather than layer-by-layer. The approach introduces two complementary metrics to evaluate how each potential rank configuration affects network accuracy. First, a PCA energy metric analyzes the singular values of each layer's weight matrix, where retaining more singular values (higher rank) preserves more of the layer's original energy. Second, a measurement-based metric evaluates actual accuracy on a validation set when compressing individual layers to different ranks. These metrics are combined to predict the accuracy of any potential rank configuration across the entire network. The optimal ranks are selected by searching for configurations that best balance the two metrics.

- The authors of [19] present a systematic approach for determining rank configurations in low-rank CNN compression. The core optimization problem simultaneously considers all layers through the objective:

$$\arg\min_{R} C(R), \quad \text{s.t. } f(R) > \tau_a \tag{10}$$

where $R$ represents the rank configuration, the cost $C(R)$ can be either memory or computational complexity, $f(R)$ is the accuracy resulting from the rank configuration $R$ and $\tau_a$ is a lower bound on the required accuracy.

The method manages the complexity of the search space by iteratively refining three constraints: establishing bounds for each layer's possible ranks, limiting the cost variation between successive iterations, and strictly enforcing the accuracy threshold $\tau_a$ to eliminate suboptimal configurations. This global optimization approach enables the discovery of non-intuitive configurations where rank reduction in certain layers can compensate for others, a capability lacking in sequential layer-wise methods. The final stage involves fine-tuning the most promising candidates to select an optimal compressed model.

- The paper [42] also presents a systematic approach for determining rank configurations in CNN compression. It was observed that the amount of PCA energy retained (the sum of the kept eigenvalues) correlates with model accuracy. Rank selection was posed as the following constrained optimization problem.

$$\max_{\{d'_l\}} \prod_l \sum_{a=1}^{d'_l} \sigma_{l,a} \quad \text{subject to} \quad \sum_l \frac{d'_l}{d_l} C_l \leq C$$

where $\sigma_{l,a}$ denotes the $a$-th eigenvalue of layer $l$, $d'_l$ is the number of kept eigenvalues, $d_l$ is the original number of eigenvalues, and $C_l$ measures the layer's complexity. After compression, the layer's complexity becomes $\frac{d'_l}{d_l} C_l$. The goal is to maximize the retained energy while keeping the overall complexity below some upper bound. To solve this, a greedy procedure is

used : starting from the uncompressed network, at each step the eigenvalue whose removal causes the smallest ratio of relative energy loss,

$$\frac{\Delta E/E}{\Delta C} = \frac{\sigma_{l,d'_l} / \sum_{a=1}^{d'_l} \sigma_{l,a}}{C_l/d_l}$$

is discarded. This process repeats until the total complexity meets the constraint. Although this method works directly with eigenvalues rather than singular values, its key idea will be adapted in section 4.

- The common denominator of these heuristic methods is that they find some rank configuration and try to compress the model as much as possible while preserving the performance. The performance are generally correlated to the energy preserved after compression, which is measured using the kept singular values. While the previously seen methods do not necessarily applies to Transformer models, [41] applies a heuristic method to find an optimal rank configuration of a Transformer model leveraging some properties about the rank of the features in a Transformer model. The paper notices that the ratio of dimensions kept when 90% singular-value energy is retained is lower for the input of a Transformer model than for its input and its Query, Key and Value matrices. Their conclusion is then that weights are full rank while features are low-rank. Based on this observation, they design a factorization method that decomposes the features of a matrix instead of its weights. This method is called Atomic Feature Mimicking and is described in detail in section 4.5.3. As a result, the original layer is split into two sequential layers by factorizing the output features. Once the compression method is established, the most challenging task is to find the optimal rank configuration (e.g. the compression rank applied to each layer). To do so, the authors define a sensitivity metric for each layer. This sensitivity metric is computed as follows.

$$S_i(k) = \sum_{x \in \mathcal{D}} D_{\text{KL}} \left( M(x, w) \parallel M(x, w_i(k)) \right)$$

where $M(x, w)$ is the output of the original model and $M(x, w_i(k)))$ is the output of the model with layer $i$ being compressed with compression rank $k$ and $D_{KL}$ is the KL divergence metric [23] that computes the difference between the probability distributions of the two model outputs. The best rank configuration is then defined as the one that minimizes the sum of sensitivities across layers while having less parameters than a given target $P_{\text{target}}$.

$$\min_{\{k_i\}_{i=1}^{l}} S = \sum_{i=1}^{l} S_i(k_i) \quad \text{s.t.} \quad \sum_{i=1}^{l} P_i(k_i) \leq P_{\text{target}}$$

where $P_i(k_i)$ counts the number of parameters of layer i.

In order to compensate for accuracy loss, a post-fine tuning on the compressed model is performed, with a loss function defined as the difference between the original model and the compressed model.

**Optimization of the Attention mechanism**. While model compression techniques such as pruning, quantization, knowledge distillation and low-rank factorization aim to reduce the parameter count of Transformer models, it is useful to notice that an alternative line of work seeks to lower the time and memory complexity of the self-attention mechanism itself. Two well-known examples of this approach are Reformer [22] and Performers [9], which reformulate attention to achieve sub-quadratic or even linear scaling with sequence length.

In Reformer, let $L$ denote the sequence length. Standard dense attention has $\mathcal{O}(L^2)$ complexity in time and memory, since each of the $L$ queries attends to all $L$ keys. Reformer replaces this with Locality-Sensitive Hashing (LSH), resulting in attention with $\mathcal{O}(L \log L)$ complexity. In order to do so, each query and key vector is projected onto random hyperplanes, then tokens with similar directions (i.e. high cosine similarity) tend to share the same pattern (i.e. the same hash). After sorting tokens by hash and splitting them into buckets of a given size (e.g. 128), attention is computed only within each bucket and its neighbors. This reduces the number of comparisons and allows sequences of tens of thousands of tokens to be processed with resources comparable to a standard Transformer handling a few thousand tokens.

Performers reduce attention complexity to $\mathcal{O}(L)$ by approximating the softmax attention mechanism using random feature maps. Instead of computing the full pairwise dot products between all queries and keys, the method uses orthogonal random projections to map queries and keys into a new space where attention can be computed more efficiently.

# 4 Proposed compression algorithm framework

*In this section, a framework that can be used to compress any Neural Network model is introduced. The framework is highly modular, so the majority of this section will be devoted to the different ways in which its degrees of freedom can be set. In the next section, the benchmarking of this framework on Transformer models will be presented.*

Following the notations introduced in section 2.2, the model compression problem via low-rank matrix factorization can be formulated as follows:

Let $M_W^L(\cdot)$ denote any $L$-layer neural network model with the set of weight matrices

$$W = \{\, W_k \in \mathbb{R}^{m_k \times n_k} \,\}_{k=1}^{K}$$

grouped by layer as $W = \bigcup_{\ell=1}^{L} W^{(\ell)}$. Let $R = (R_1, R_2, \ldots, R_K)$ be a *rank configuration* with $R_k \leq \min(m_k, n_k)$. We write $W_R$ for the set of compressed weight matrices obtained by truncating each $W_k$ to rank $R_k$, and denote the compressed network by $M_{W_R}^L(\cdot)$.

Given unlabeled input data $\mathbf{X}$, we seek a low-rank model $M_{W_R}^L$ whose weights $W_R$ solve:

$$\begin{aligned} \underset{W_R}{\text{minimize}} \quad & \mathcal{C}(W_R) \\ \text{subject to} \quad & E(M_W^L(\mathbf{X}) \, , \, M_{W_R}^L(\mathbf{X})) \;\leq\; \tau, \end{aligned}$$

where $\mathcal{C}(W_R)$ is the total number of parameters in the compressed model, $E(\cdot,\cdot)$ is some error measure and $\tau > 0$ is a user-specified error tolerance.

The key idea is that with a good choice of $\tau$, the compressed model's outputs will stay close to the original model's outputs. This works similarly to knowledge distillation [14], where the original model acts as the *teacher* and the compressed model acts as the *student*. Just like in distillation, we want the student's outputs to match the teacher's outputs as closely as possible (controlled by $\tau$). If the compressed model can mimic the original model's behavior, it might retain the same useful information for making predictions.

Though this is a combinatorial optimization problem (finding the best combination of ranks), there are so many possible combinations that checking them all is not feasible. For instance, compressing $K = 100$ weight matrices (each of size 768×768) would require testing up to $10^{1536}$ different rank combinations. To address this, we use a generic stepwise strategy. First, we sort the weight matrices based on how suitable they are for compression. We then start compressing

them one by one using some initial truncation rank $r_0$. If the compression of a layer causes the output error to exceed $\tau$, we loosen its compression (increase $r$) until the error drops below $\tau$. This procedure continues until we reach a predefined maximum truncation rank $r_{\max}$. The process is formalized in Algorithm 1 and can be repeated with different starting ranks $r_0$.

**Algorithm 1** Compression framework

**Require:** $M_W^L$, truncation rank $r_0, r_{\max}$, $\Delta r$, threshold $\tau$, proxy dataset $\mathbf{X}$

Sort weight matrices $W_k$ according to some criteria
    $r \leftarrow r_0$
**for** each weight matrix $W_k$ **do**
    Compress $W_k$ with truncation rank $r$ to obtain $M_{\Theta_R}$
    **while** $E(M_W^L(\mathbf{X}), M_{W_R}^L(\mathbf{X})) > \tau$ **and** $r \leq r_{\max}$ **do**
        Compress $W_k$ with $r \leftarrow r + \Delta r$
    **end while**
**end for return** compressed model $M_{W_R}^L$

The compression framework described in Algorithm 1 offers great flexibility due to its many configurable hyperparameters. These include choices such as the matrix sorting criterion, the module used to generated the output of the model (e.g. backbone or classification head), the compression method, the threshold parameter $\tau$, and the selection of some *proxy dataset*. Each hyperparameter influences the framework's behavior and outcomes, and their practical implications will be explored in subsequent sections.

## 4.1 Sorting criteria and rank approximation

In Algorithm 1, the weight matrices are sorted according to some criteria that measures how suitable are weight matrices for compression. For a weight matrix $W \in \mathbb{R}^{m \times n}$, and taking inspiration from [42] : the sorting criteria $s(W)$ can be defined as $s(W) = \frac{size(W)}{rank(W)}$, where $size(W)$ is just $\max\{m, n\}$ and $rank(W)$ is the rank of $W$. However, in practice matrices are almost always full rank as their rows or columns are rarely exact linear combinations of one another. Therefore, we need to find robust metrics that approximate the rank. In the following, we will consider $\sigma_1(A) \geq \sigma_2(A) \geq \cdots \geq \sigma_p(A)$ (where $p = \min\{m, n\}$) to be the singular values of $A \in \mathbb{R}^{m \times n}$.

**Numerical rank.** A matrix $A$ is said to have an $\epsilon$-numerical rank of $k$ if it satisfies:

$$\sigma_1(A) \geq \sigma_2(A) \geq \cdots \geq \sigma_k(A) \geq \epsilon > \sigma_{k+1}(A) \geq \cdots \geq \sigma_p(A) \tag{11}$$

This definition depends critically on the choice of $\epsilon$. In practice, $\epsilon$ is often

set in relation to machine precision. For instance, the NumPy library [13] uses

$$\epsilon = \sigma_{\max} \cdot \max\{m, n\} \cdot \epsilon_{\text{datatype}}$$

where $\epsilon_{\text{datatype}}$ is the machine precision associated with the data type of the matrix. One major drawback of this metric is its high sensitivity to the value of $\epsilon$. The default setting in NumPy is designed to be robust against perturbations on the order of rounding errors. As a result, real-world matrices (e.g. weight matrices in neural networks) are often assigned full numerical rank, even if they exhibit low-rank structure. Moreover, the numerical rank does not account for the distribution of the singular values. It effectively applies a hard threshold, ignoring the fact that in many matrices, only a few singular values may contain most of the informative content.

**Nuclear norm** (also known as the *trace norm*). The nuclear norm of a matrix $A$ is defined as [27]:

$$\|A\|_* = \sum_{i=1}^{p} \sigma_i(A) \tag{12}$$

This metric is widely used in optimization because, when $\|A\|_2 \leq 1$, the nuclear norm serves as the convex envelope of the rank function [27]. While this property is valuable in certain optimization problems, convexity is not a requirement for our purposes.

A key drawback of the nuclear norm is its sensitivity to scaling. Let $\lambda_i(A)$ denote the $i$-th eigenvalue of $A$. Since the singular values satisfy $\sigma_i(A) = \lambda_i(\sqrt{A^\top A})$, scaling $A$ by a constant $c \in \mathbb{R}$ yields:

$$\sigma_i(cA) = \lambda_i\left(\sqrt{(cA)^\top (cA)}\right) = |c|\lambda_i\left(\sqrt{A^\top A}\right) = |c|\sigma_i(A)$$

$$\|cA\|_* = \sum_{i=1}^{p} \sigma_i(cA) = |c| \sum_{i=1}^{p} \sigma_i(A) = |c| \|A\|_*$$

This linear dependence on the scaling factor $c$ highlights that the nuclear norm does not provide a scale-invariant measure of rank.

**Stable rank**. The stable rank of a matrix $A \in \mathbb{R}^{m \times n}$ is defined as [16]:

$$\text{sr}(A) := \frac{\|A\|_F^2}{\|A\|_2^2} = \frac{\sum_i \sigma_i^2(A)}{\sigma_1^2(A)} \tag{13}$$

The sum of the squared singular values is normalized by the largest singular value, preventing the stable rank from scaling with the norm of $A$. When the largest singular value dominates the others, it suggests that the matrix's information can be effectively captured in a one-dimensional space, resulting in a low stable rank. A major drawback of this property is that the metric becomes highly sensitive to the distribution of singular values relative to the largest one.

However, a matrix may still contain redundant information, even if its content cannot be adequately represented by a single dominant singular value.

**Effective rank.** The effective rank of a matrix $A \in \mathbb{R}^{m \times n}$ is defined in terms of the Shannon entropy of its normalized singular-value spectrum [34]. Let

$$p_i = \frac{\sigma_i(A)}{\sum_{j=1}^{p} \sigma_j(A)}, \quad \sum_{i=1}^{p} p_i = 1$$

the effective rank is given by

$$\text{erank}(A) = \exp\Big(-\sum_{i=1}^{Q} p_i \log p_i\Big) \tag{14}$$

The effective rank remains invariant under scaling of $A$, since each $p_i$ is unchanged if all $\sigma_i$ are multiplied by the same constant. When all nonzero singular values are equal (e.g. $A = I_k$), one has $p_i = 1/k$ for $i = 1, \ldots, k$ and thus $\text{erank}(A) = k$.

**Comparison.** To gain intuition about rank metrics, we evaluate them on simple matrices and observe their behavior. Consider the following matrices and their corresponding rank metrics (Table 7):

$$A_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad A_2 = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{bmatrix},$$
$$A_3 = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad A_4 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{bmatrix},$$
$$A_5 = \begin{bmatrix} 1 & 1 & 1 \\ 10^{-16} & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad A_6 = \begin{bmatrix} 1 & 1 & 1 \\ 10^{-2} & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

| | $\epsilon$-rank | $\lVert . \rVert_*$ | sr(.) | erank(.) |
|---|---|---|---|---|
| $A_1$ | 3 | 3.000 | 3.000 | 3.000 |
| $A_2$ | 3 | 6.000 | 3.000 | 3.000 |
| $A_3$ | 3 | 4.000 | 1.500 | 2.828 |
| $A_4$ | 3 | 5.000 | 2.250 | 2.872 |
| $A_5$ | 1 | 1.732 | 1.000 | 1.000 |
| $A_6$ | 2 | 1.740 | 1.000 | 1.030 |

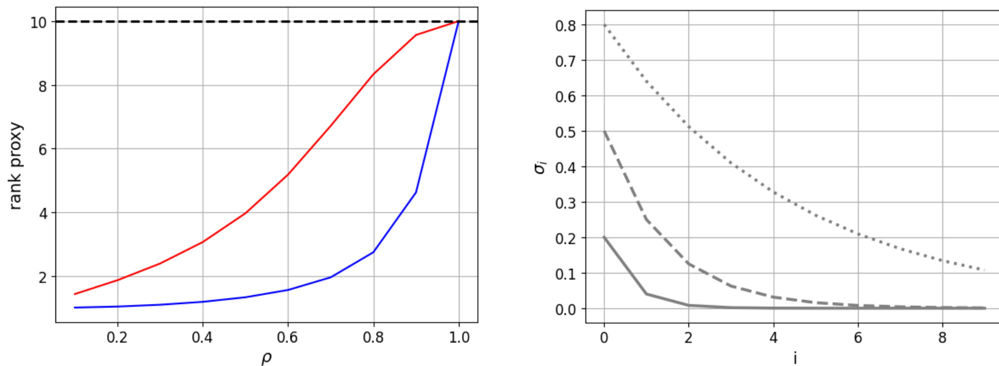
Table 7: Rank metric values for simple matrices. the numerical rank is denoted as $\epsilon$-rank

For matrix $A_5$, only the first column $(1, 0, 0)$ contains meaningful information, while the second column is corrupted by rounding error noise $(10^{-16})$.

The numerical rank robustly ignores this perturbation and correctly returns 1. However, when the corruption factor increases to $10^{-2}$ (as in $A_6$), the default $\epsilon$ parameter for numerical rank becomes insufficient, resulting in a rank of 2. This demonstrates the sensitivity of numerical rank to the $\epsilon$ threshold. Comparing $A_1$ and $A_2$, we observe the nuclear norm scales linearly with the matrix entries ($\|A_2\|_* = 2\|A_1\|_*$), consistent with the property $\|cA\|_* = |c|\|A\|_*$. For full-rank matrices $A_3$ and $A_4$, the stable rank shows high sensitivity to the ratio between the largest singular value and others. Specifically: $\text{sr}(A_3) = \frac{2^2+1^2+1^2}{2^2} = 1.5$, and $\text{sr}(A_4) = \frac{2^2+2^2+1^2}{2^2} = 2.25$. In contrast, the effective rank remains closer to full-rank values, demonstrating its reduced sensitivity to singular value dominance. To further illustrate this behavior, consider a diagonal full-rank matrix $A_\rho \in \mathbb{R}^{n\times n}$ parametrized by some $\rho \leq 1$ :

$$A_\rho = \text{diag}(\rho^1, \rho^2, \ldots, \rho^n) = \begin{bmatrix} \rho^1 & 0 & \cdots & 0 \\ 0 & \rho^2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \rho^n \end{bmatrix}$$

As $\rho$ decreases, the singular value distribution becomes steeper, amplifying the relative importance of $\sigma_1$. Consequently, the stable rank collapses to 1 for small $\rho$ (despite the matrix being full-rank), while the effective rank remains less sensitive to this decay, as shown in Figure 4.




(a) Comparison of stable rank (blue) and effective rank (red) for $A_\rho \in \mathbb{R}^{10\times 10}$. The stable rank collapses to 1 for small $\rho$, while the effective rank better preserves rank information. Dashed black line indicates the true rank.

(b) Singular value distributions for $\rho = 0.8$ (dotted), $\rho = 0.5$ (dashed), and $\rho = 0.2$ (solid). Smaller $\rho$ values produce steeper decays.

Figure 4: Sensitivity of rank metrics to singular value distribution.

The effective rank emerges as the preferred metric due to its robustness to scaling ($A_1$ vs. $A_2$), singular value dominance ($A_3$ vs. $A_4$), and perturbations ($A_5$ vs. $A_6$).

## 4.2 Output features

As explained in section 2.3, there are multiple ways to compute the output of a model $M_W^L$ depending on if it is a pretrained model $M_{W,Pre}^L$ or a classification model $M_{W,Cls}^L$. In this work, the output of a pretrained model will be the output of the backbone module but for a classification model, the output can be either the output of the backbone or the output of the classification head.

**Backbone output**. The output of the backbone module corresponds to the decoder's output (or the encoder's output in encoder-only architectures). This output representation $\mathbf{Y} \in \mathbb{R}^{n \times d_{model}}$ is not directly optimized for any particular downstream task.

**Classification head output**. As discussed in Section 2.3, transformer models designed for classification include a classification head that generates an output matrix $\mathbf{Y} \in \mathbb{R}^{n \times c}$, where $c$ is the number of classes. This output is typically passed through a softmax layer to produce label probability distributions. When this output is retained, it is referred to as the **soft output** $Y_{\text{soft}} \in \mathbb{R}^{n \times c}$, which consists of floating-point values representing confidence scores. In contrast, the **hard output** $Y_{\text{hard}} \in \{0, 1\}^{n \times c}$ is generated by setting the largest value in each row to 1 and all others to 0, producing a one-hot encoded classification decision.

Soft outputs contain a lot of information. Indeed, as noted in [14], the relative magnitudes of the values in the probability vector reveal insights into the model's generalization behavior. For example, if the model assigns the highest logit to the correct class but assigns identical values to all other logits, this suggests limited ability to distinguish between alternative classes. Hard outputs, however, are essential for final decision-making.

A balanced approach combines both outputs through a linear combination. Following [14], this **distillation output** is defined as $Y_{\text{distill}} = (1 - \alpha)Y_{\text{soft}} + \alpha Y_{\text{hard}}$, where $\alpha$ controls the trade-off between probabilistic and deterministic assignments. This strategy is particularly effective in knowledge distillation, where a compact student model learns to emulate the behavior of a larger teacher model, which is quite similar to our algorithm settings. The choice of $\alpha$ will be discussed in sections 5.2.1 and 5.3.

## 4.3 Error measure

The errors measure between models $M_W^L$ and $M_{W_R}^L$, denoted as $E(M_W^L(\mathbf{X}), M_{W_R}^L(\mathbf{X}))$, can be quantified using different methodologies. In this work, we define it as the normalized Frobenius norm:

$$E(M_W^L(\mathbf{X}), M_{W_R}^L(\mathbf{X})) := \frac{\|M_W^L(\mathbf{X}) - M_{W_R}^L(\mathbf{X})\|_F}{\|M_W^L(\mathbf{X})\|_F}$$

This metric serves as a practical proxy for compression error. A high output error indicates that the compressed model $M_{W_R}^L$ inadequately replicates the behavior of the original model $M_W^L$, potentially leading to significant performance

degradation on downstream tasks. The normalization term $\|M_W^L(\mathbf{X})\|_F$ will later facilitate the selection of the threshold parameter $\tau$, as discussed in subsequent sections.

## 4.4 Proxy dataset

In order to compute the compression error (see Section 4.3), the compression algorithm requires a proxy dataset $\mathbf{X} \in \mathbb{R}^{n \times k}$, with $k \in \{n_{\text{vocab}}, d_{\text{model}}\}$. The choice of this proxy dataset has a decisive impact on the observed compression quality. Two main strategies are commonly considered. In a task-specific approach, $\mathbf{X}$ consists of real samples drawn from the target dataset, ensuring that the compressed model preserves performance on that particular distribution. By contrast, a random proxy takes $\mathbf{X}$ to be a collection of synthetic inputs sampled i.i.d. from a standard normal distribution. This forces the compression to respect the model's overall behavior rather than its performance on any single dataset. Both approached will now be described in more detail.

**Task-specific proxy**. Here $\mathbf{X} \in \mathbb{R}^{n \times n_{vocab}}$ is sampled from a target dataset and fed through the transformer's embedding module exactly as during fine-tuning or inference on the real task. The advantage of a task-specific proxy is that it directly controls the trade-off between compression ratio and output error on the application of interest. However, because the compression is optimized for a single dataset, the compressed model may overfit and fail to generalize to other inputs.

**Random proxy**. When task-specific data are limited or when we want the compressed model to work well in general, we build $\mathbf{X} \in \mathbb{R}^{n \times d_{model}}$ by sampling each entry independently from $\mathcal{N}(0, 1)$. This idea comes from compressed sensing [5], where one measures a sparse signal $f \in \mathbb{R}^N$ using a random matrix $\Phi \in \mathbb{R}^{M \times N}$ to obtain

$$y \;=\; \Phi\, f$$

Since $f$ can be written as $f = \Psi\, \alpha$ in some basis $\Psi$ with a sparse coefficient vector $\alpha$, one recovers $\alpha$ by solving an $\ell_1$ minimization from the measurements $y$.

In our compression framework, we make the same analogy. We view the original model's output on random inputs $\Phi$ as

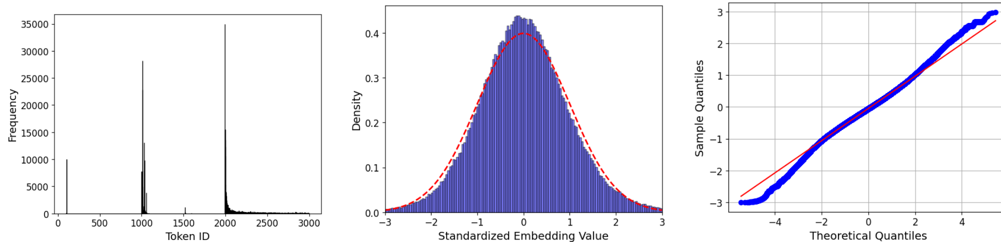
$$y = \Phi\, f,$$

and we view the compressed model's output as

$$\hat{y} = \Phi\, (\Psi\, \alpha)$$

However, in our case the compressed model is not necessarily a sparse representation of the original network. By forcing $\hat{y}$ to match $y$ over many random matrices $\Phi$, we make the compressed model replicate the original model's behavior across a wide range of inputs, rather than fitting it only to a single task dataset.

For the method to be relevant, it must be ensured that the data presented to a Transformer approximate a Gaussian distribution. However, this is not the case for raw token IDs. When a tokenized input $\mathbf{X} \in \mathbb{R}^{n \times n_{vocab}}$ is examined (where each entry corresponds to a token index) it is observed that these indices do not follow a normal distribution (see Figure 5(a)). Instead, focus is turned to $\mathbf{X} \in \mathbb{R}^{n \times d_{model}}$, the output of the word embedding layer (after positional encoding) that and represents the continuous feature embeddings. By processing the entire WikiText dataset [29] through the embedding layer and then flattening and standardizing its outputs, a distribution is obtained that closely matches a Gaussian, as shown in Figure 5(b). To confirm this fit, a QQ-plot (Figure 5(c)) was generated after suppressing a small number of extreme outliers (see Appendix A for more details). A QQ-plot compares the sorted sample values against the theoretical quantiles of a standard normal distribution (if the points lie approximately on a straight line, the data may be regarded as Gaussian). In this case, the resulting QQ-plot is nearly linear, confirming that the embedding outputs are normally distributed. It is therefore justified to replace the embedding layer with a Gaussian proxy when evaluating compression. The random input tensor can be drawn from $\mathcal{N}(0, 1)$ and fed directly into the Transformer's subsequent encoder blocks.




(a) Distribution of the first 3000 token IDs in the WikiText dataset, using the BERT tokenizer.

(b) Distribution of standardized values from the output of the word embedding layer on the WikiText dataset.

(c) QQ-plot of the standardized embedding output features. The near-linear pattern indicates that the data follow a normal distribution.

Figure 5: Comparison between the distribution of raw token IDs and the distribution of embedding outputs from a Transformer model on the WikiText dataset [24, 29]. While token IDs are discrete and non-Gaussian, the continuous embedding outputs approximate a Gaussian distribution after standardization.

## 4.5 Matrix factorization method

### 4.5.1 Singular Value Decomposition (SVD)

Singular Value Decomposition can be used to obtain the best low-rank approximation of a dense layer in a neural network. Let $X \in \mathbb{R}^{n \times d_{\text{in}}}$ be an input matrix. A dense layer, defined by a weight matrix $W \in \mathbb{R}^{d_{\text{in}} \times d_{\text{out}}}$ and a bias

term $b \in \mathbb{R}^{d_{\text{out}}}$, produces an output $Y = XW + b \in \mathbb{R}^{n \times d_{\text{out}}}$.

Following the notation introduced in Section 2.1, the best rank-$k$ approximation of the weight matrix is given by $W \approx U_k \Sigma_k V_k^T = W_1 W_2$, where $W_1 = U_k \Sigma_k \in \mathbb{R}^{d_{\text{in}} \times k}$ and $W_2 = V_k^T \in \mathbb{R}^{k \times d_{\text{out}}}$. This leads to an approximate output $Y \approx (XW_1)W_2 + b$. Therefore, the compressed dense layer can be interpreted as a sequence of two smaller dense layers.

### 4.5.2 classical matrix factorization methods

The problem with SVD decomposition is its $O(n^3)$ complexity. Some faster low-rank approximation methods exist in the literature but they sacrify some approximation accuracy for a smaller complexity. Therefore, a good tradeoof needs to be found. [21] contains a litterature review on low-rank approximation methods, the methods presented in this study will be introduced and compare in this section.

**Truncated SVD**. We can obtain the rank-$k$ approximation directly by computing a *truncated SVD*, which focuses only on the leading $k$ singular values and vectors. This approach avoids the full decomposition and significantly reduces computational cost. In practice, truncated SVD can be implemented using partial QR factorization followed by post-processing, requiring only $\mathcal{O}(kmn)$ flops.

**Randomized SVD**. Randomized SVD provides an efficient approximation to the singular value decomposition of a large matrix by first compressing its column space with a random projection and then applying a conventional SVD to a much smaller matrix. A random test matrix $\Omega \in \mathbb{R}^{n \times s}$ with $s \geq k$ is drawn and used to form $Y = A\Omega \in \mathbb{R}^{m \times s}$. A QR factorization is performed on $Y$, producing an orthonormal matrix $Q \in \mathbb{R}^{m \times k}$. The original matrix is then projected into this subspace via $B = Q^T A \in \mathbb{R}^{k \times n}$, and a full SVD of $B$ is computed : $B = \widetilde{U} \Sigma V^T$. The approximate left singular vectors of $A$ are recovered by setting $U = Q\widetilde{U}$, giving the factorization $U \Sigma V^T$, which closely approximates the rank-$k$ SVD of $A$. The cost is dominated by the multiplication $A\Omega$ and the SVD of the small matrix $B$, together requiring on the order of $\mathcal{O}(mn \log k + k^2(m+n))$.

**Pivoted QR Decomposition**. Given a matrix $A \in \mathbb{R}^{m \times n}$ (with $m \geq n$), the pivoted QR decomposition factorizes $A$ as

$$AP = QR,$$

where $P \in \mathbb{R}^{n \times n}$ is a permutation matrix that reorders the columns of $A$, $Q \in \mathbb{R}^{m \times n}$ is an orthonormal matrix, and $R \in \mathbb{R}^{n \times n}$ is an upper triangular matrix. The permutation matrix $P$ is chosen via a column pivoting strategy that selects columns of $A$ in order of decreasing importance, measured by their norms. A rank-$k$ approximation of $A$ can be constructed by partitioning the factors as

$$Q = [Q_1^{(k)} \quad Q_2^{(m \times (n-k))}], \quad R = \begin{bmatrix} R_{11}^{(k \times k)} & R_{12} \\ 0 & R_{22} \end{bmatrix}$$

where $Q_1^{(k)}$ consists of the first $k$ columns of $Q$, and $R_{11}^{(k\times k)}$ is the leading principal submatrix of $R$, to obtain

$$\hat{A}_k = Q_1 R_{11} R_{12}$$

The remaining block $R_{22}$ is small in norm, indicating that the discarded columns contribute minimally to the approximation error. The pivoted QR decomposition can be computed with $\mathcal{O}(kmn)$ complexity, which is better than SVD but less accurate.

**Interpolative Decomposition**. Interpolative decomposition seeks to approximate a matrix $A \in \mathbb{R}^{m\times n}$ of rank $k$ by selecting $k$ of its columns and expressing the remaining columns as linear combinations of those. Concretely, one finds an index set $J$ of size $k$ and a coefficient matrix $P \in \mathbb{R}^{k\times n}$ whose entries satisfy $|P_{ij}| \leq 1$ and whose columns include the $k \times k$ identity, so that

$$A \approx A_{:,J}\, P$$

and the error in the spectral or Frobenius norm is bounded by a constant multiple of the $(k+1)$-th singular value of $A$. Deterministic algorithms based on strong rank-revealing QR achieve this in $\mathcal{O}(mn^2)$ time, while randomized approaches can reduce the cost to $\mathcal{O}(k\, m\, n \log(n))$.

**Skeleton Decomposition (ID)**. Skeleton decomposition approximates $A \in \mathbb{R}^{m\times n}$ by using a small set of its columns and rows. One first selects $k$ column indices $J = (j_1, \ldots, j_k)$ (for example via an interpolative decomposition or pivoted QR) and forms $C = A_{:,J} \in \mathbb{R}^{m\times k}$. Next, one chooses $k$ row indices $I = (i_1, \ldots, i_k)$ so that the $k \times k$ intersection submatrix

$$M = A_{I,J} \in \mathbb{R}^{k\times k}$$

has near maximal volume (i.e. maximal absolute determinant). In practice an algorithm called the *maxvol* algorithm finds $I$ in $\mathcal{O}(mk)$ time. Finally one sets $R = A_{I,:} \in \mathbb{R}^{k\times n}$ and $G = M^{-1}$ (or its pseudoinverse if $M$ is ill conditioned), giving

$$A \approx C\, G\, R$$

The method is of $\mathcal{O}\big((m+n)k^2\big)$ complexity.

**Comparison**. The execution times of the various low-rank factorization methods are compared in Figure 6 for a $768 \times 768$ matrix (typical of Transformer weight layers). When the target rank is very small (for example, $k = 10$), the original SVD method is actually the lowest. As $k$ grows beyond around 100, the other low-rank methods slow down, because they are designed to exploit the case $k \ll \min(m, n)$. Since SVD remains the most accurate option, as shown in Section 2.1, we will continue to use full SVD for our $768 \times 768$ matrices, as none of the alternative methods offer significant time savings.

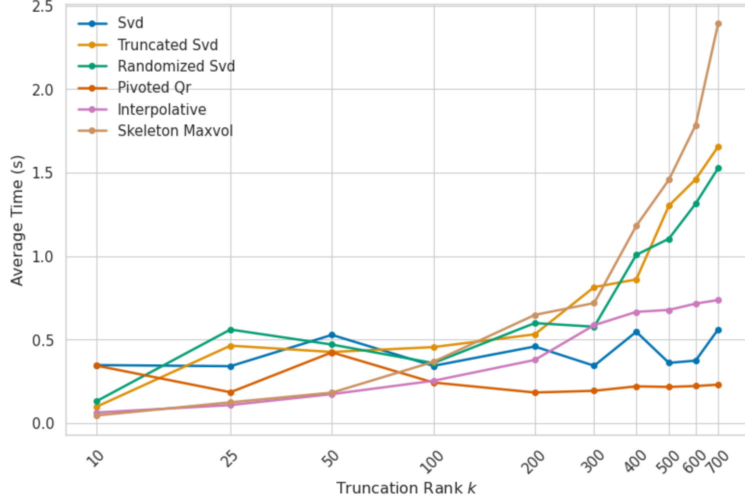



Figure 6: Comparison of the computation time for different low-rank compression methods on a $768 \times 768$ matrix with different truncation ranks. The results are averaged over 10 trials.

### 4.5.3 Atomic Feature Mimicking (AFM)

Atomic Feature Mimicking is a compression method introduced in [41] and used to find the optimal rank configuration of a Transformer model leveraging some properties about the rank of the features. The paper notices that the ratio of dimensions kept when 90% singular-value energy is retained is lower for the input of a Transformer model than for its input and its Query, Key and Value matrices. Their conclusion is then that weights are full rank while features are low-rank. Based on this observation, they design a factorization method that decomposes the features of a matrix instead of its weights. The method can be described as follows :

Let $Y \in \mathbb{R}^{n \times d_{\text{model}}}$ be the output of some weight matrix, defined as $Y = XW$, where $X \in \mathbb{R}^{n \times d_{\text{model}}}$ is sampled from a target dataset and passed through the model, and $W \in \mathbb{R}^{d_{\text{model}} \times d_{\text{model}}}$. We define $\mathbb{E}[\cdot]$ as the expectation operator over the token dimension. The covariance matrix of $Y$ is given by:

$$\text{Cov}(Y) = \mathbb{E}[YY^T] - \mathbb{E}[Y]\,\mathbb{E}[Y]^T$$

Since the covariance matrix is symmetric, we can compute its spectral decomposition as in [6]:

$$\text{Cov}(Y) = U\Sigma U^T, \quad \text{where } U \text{ is orthogonal}$$

Let $U_k$ be the truncated version of $U$, obtained by keeping its first $k$ columns. Since $U_k U_k^T \approx I_n$, we can apply the following approximation:

$$Y - \mathbb{E}[Y] \approx (Y - \mathbb{E}[Y])U_k U_k^T$$

$$Y \approx (Y - \mathbb{E}[Y])U_k U_k^T + \mathbb{E}[Y]$$
$$Y \approx Y U_k U_k^T + \mathbb{E}[Y] - \mathbb{E}[Y]U_k U_k^T$$
$$Y \approx (XW + b - \mathbb{E}[Y])U_k U_k^T + \mathbb{E}[Y] - \mathbb{E}[Y]U_k U_k^T$$
$$Y \approx (XW + b)U_k U_k^T + \mathbb{E}[Y] - \mathbb{E}[Y]U_k U_k^T$$
$$Y \approx (XWU_k + bU_k)U_k^T + \mathbb{E}[Y] - \mathbb{E}[Y]U_k U_k^T$$
$$Y \approx (XW_1 + b_1)W_2 + b_2$$

This sequence illustrates how we can approximate $Y$ using a low-rank factorization strategy, where $W_1 = WU_k$, $b_1 = bU_k$, $W_2 = U_k^T$, and $b_2 = \mathbb{E}[Y] - \mathbb{E}[Y]U_k U_k^T$. The original layer is thus split into two sequential layers by factorizing the output features. The choice between AFM and SVD will be discussed depending on the use cases in section 5.2.1.

## 4.6 Other hyper-parameters

In practice, the allowable range of truncation ranks $[r_0, r_{max}]$ is set to $[0.1, 0.6] \times 768$, with $\Delta r = 0.1 \times 768$. Enlarging the interval and reducing the step-size would result in a greater search space, providing potentially better results but increasing computation time. Reducing the interval and increasing the step-size would result in an acceleration of computation time, at the cost of missing interesting configurations. The hyper-parameter choice made here seems to be a good tradeoff.

The error threshold $\tau$ can be chosen in several ways. A straightforward method for selecting a good value is to try different candidates within a given range, run the compression algorithm for each one, and evaluate which model performs best. To decide which value of $\tau$ gives the best results, we will use the one that maximizes the ratio between the accuracy of the compressed model and the fraction of parameters it retains after compression. This is the criterion we will use for benchmarking. Because the error is adjusted by a normalization factor, we will explore values of $\tau$ between 0.25 and 1, and in some cases up to 1.5 when targeting stronger compression.

# 5 Numerical results

*This section is focused on the evaluation of how the compression algorithm performs on the GLUE benchmark. The structure of the compressed Transformer models will also be examined in a few particularly interesting cases.*

## 5.1 Rank structure of Transformer models

The compression framework is designed to sort the weight matrices of Transformer models based on a *sorting criteria*, which is computed as the ratio between the largest dimension of a weight matrix and its approximate rank, as determined by the *effective rank*. As a recall, this sorting criteria measures how suitable are weight matrices for compression. For this reason, it is helpful to examine the effective rank and sorting criteria configuration of the models used for benchmarking, since these weight matrices will be prioritized for compression. The BERT and GPT-2 models are considered in this section. The corresponding configurations for RoBERTa are provided in Appendix B, as its architecture is similar to that of BERT.

In the following scenarios, only the most important weight matrices of the backbone module of each Transformer model are taken into account, specifically the attention and feed-forward weight matrices. The embedding layer is excluded from compression for practical reasons and to ensure consistency across scenarios, as some configurations feed inputs directly into the backbone, bypassing the embedding layer. As a result, $K = 72$ weight matrices are eligible for compression in the BERT and RoBERTa models, and $K = 48$ in the GPT-2 model.

The effective rank of the weight matrices in the BERT model is visualized in Figure 7, where the rank values are normalized by the maximum effective rank among the weight matrices in the architecture. It can be observed that the feedforward layers exhibit the highest effective rank, while the attention weight matrices display lower effective ranks. A slight increase in rank with the depth of the layers can also be noticed. For the GPT-2 model shown in Figure 8, the feedforward layers similarly show the highest rank. However, the attention weight matrices, except for the projection weight matrices, also have high effective ranks comparable to those of the feedforward layers. The attention projection weight matrices display lower effective ranks relative to the other weight matrices. A similar slight trend of increasing effective rank with layer depth is observed.

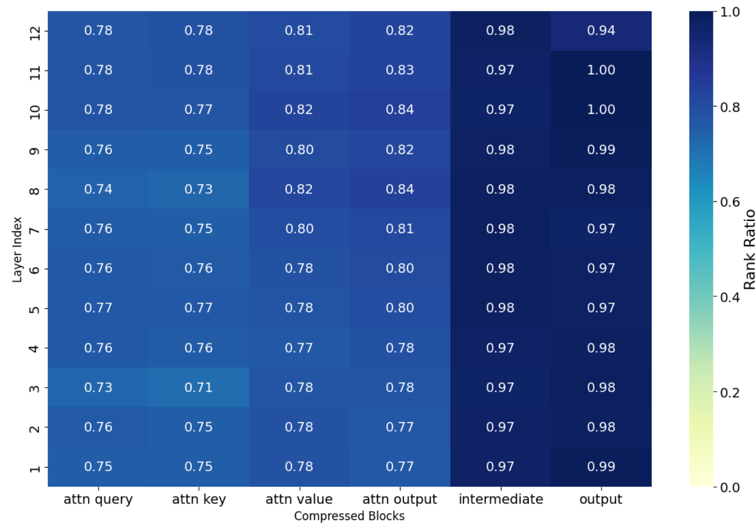



Figure 7: The normalized effective rank is shown for the BERT model with $K = 72$ weight matrices. The first three columns represent, respectively, the query, key, and value weight matrices, with their layer index indicated on the y-axis ranging from 1 to 12. The fourth column corresponds to the attention projection weight matrix, while the last two columns represent the feed forward layers. The maximal effective rank value is 726.6

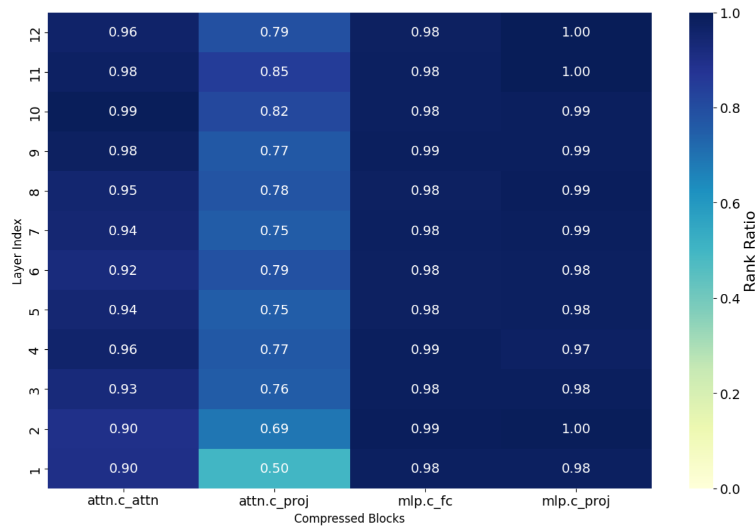



Figure 8: The normalized effective rank is shown for the GPT-2 model with $K = 48$ weight matrices. The first column represent the concatenated query, key, and value weight matrices, with their layer index indicated on the y-axis ranging from 1 to 12. The second column corresponds to the attention projection weight matrix, while the last two columns represent the feed forward layers. The maximal effective rank value is 713.1

The sorting criteria for the BERT and GPT-2 weight matrices is shown in Figures 9 and 10, respectively. Similar dynamics to those of the effective rank are observed, but the gap between the attention and feedforward layers for BERT, and between the projection and other attention matrices for GPT-2, is more pronounced. This is because the attention matrices in BERT and the projection matrices in GPT-2 are of size $768 \times 768$, while the feedforward layers are $3072 \times 768$. Weight matrices of equal size are then ranked by their effective rank.

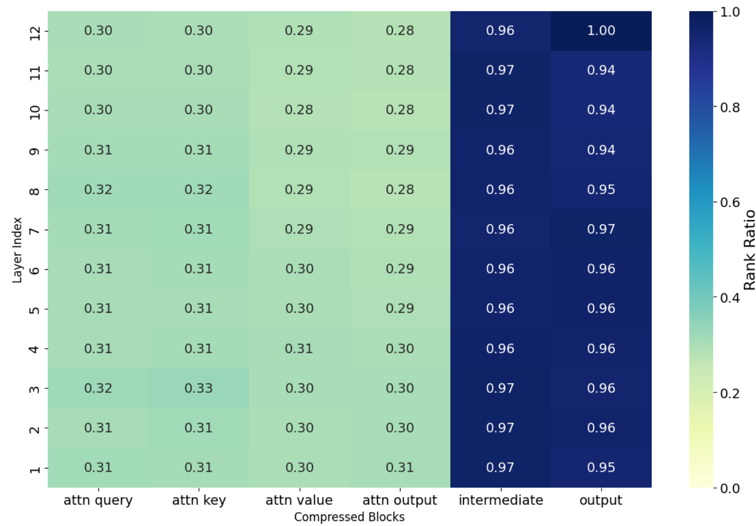



Figure 9: The normalized sorting criteria computed as the ratio between the maximal dimension of a given weight matrix and its effective rank, is shown for the BERT model with $K = 72$ weight matrices. The first three columns represent, respectively, the query, key, and value weight matrices, with their layer index indicated on the y-axis ranging from 1 to 12. The fourth column corresponds to the attention projection weight matrix, while the last two columns represent the feed forward layers.

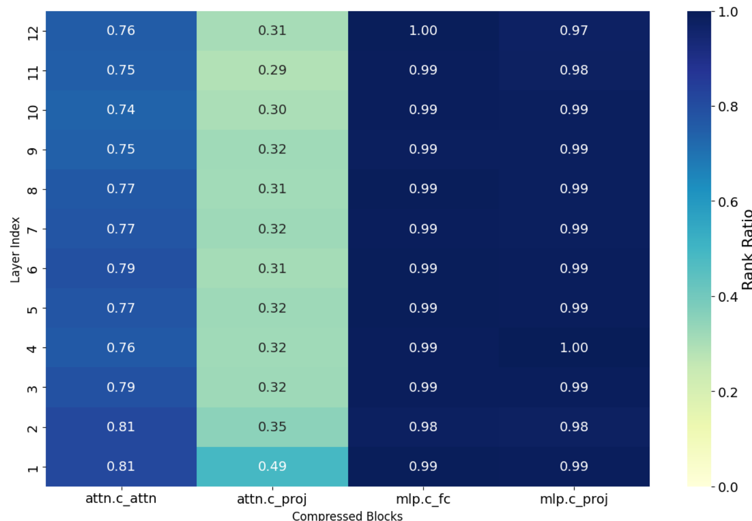



Figure 10: The normalized sorting criteria computed as the ratio between the maximal dimension of a given weight matrix and its effective rank is shown for the GPT-2 model with $K = 48$ weight matrices. The first column represent the concatenated query, key, and value weight matrices, with their layer index indicated on the y-axis ranging from 1 to 12. The second column corresponds to the attention projection weight matrix, while the last two columns represent the feed forward layers.

## 5.2 Fine-tuned model compression

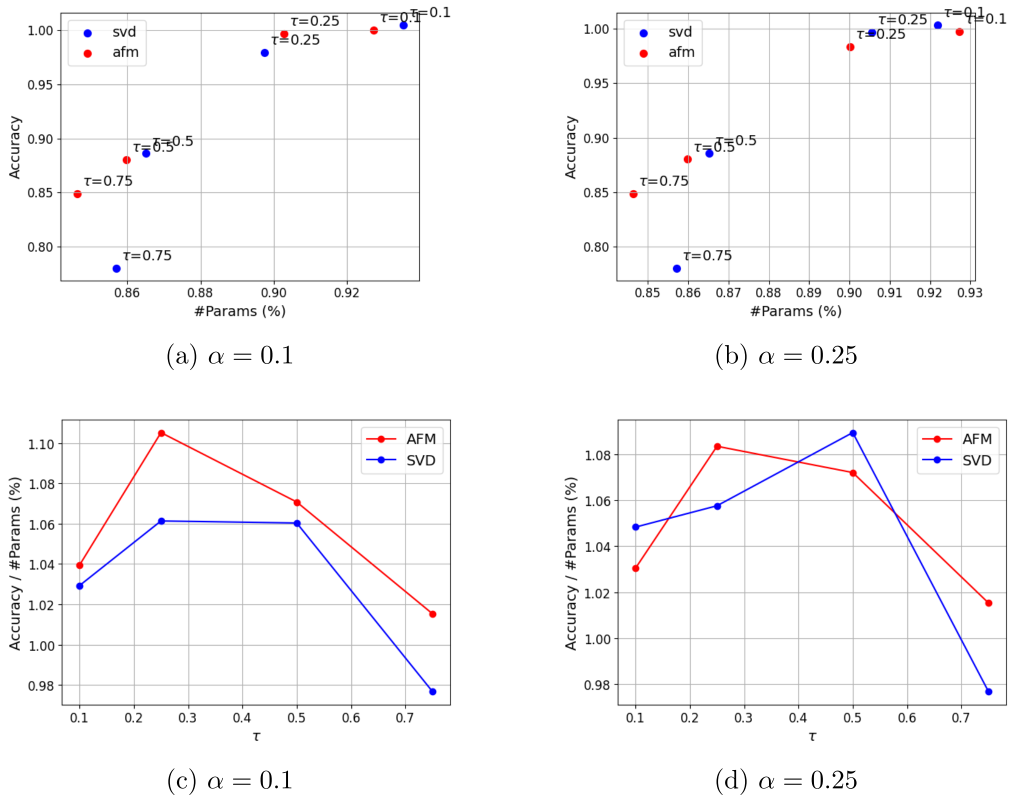
The compression framework is evaluated on the BERT, GPT-2, and RoBERTa models fine-tuned on the CoLA, MRPC, RTE, QNLI, and WNLI datasets from the GLUE benchmark. As described in Section 2.3, the fine-tuned models include a classification head and are denoted as $M^L_{W,Cls}$. The 5 datasets were selected for practical reasons, such as having the same number of classes and reasonable sizes. Evaluating three models across all 9 GLUE tasks was not feasible due to limited computational resources. It should also be noted that results on WNLI show high variance because the dataset only contains a few hundred samples. All models were trained on the training split and evaluated on the validation split of each dataset and using 3 epochs to keep the computational cost reasonable. **Note that no post-compression fine-tuning is performed here.**

To assess the effectiveness of the compression framework, its results are compared to a simple SVD-based compression where all layers are compressed using the same fixed truncation rank. This comparison helps to show whether the proposed method finds a better compression configuration. State of the art results will not be mentioned in this section since they are generally obtained after a fine-tuning step following the compression stage. We will consider 2 types of proxy datasets : task-specific and random proxy datasets as they were described in section 4.4. The framework hyper-parameters used in each cases will be discussed in the next sections.

### 5.2.1 Task-specific proxy

When an unlabeled task-specific dataset $\mathbf{X} \in \mathbb{R}^{n \times n_{\text{vocab}}}$ is used within the compression framework, the models are compressed using this dataset. The dataset $\mathbf{X}$ is provided as input to the embedding layer of the classification models $M^L_{W,Cls}$. The distillation output $Y = (1-\alpha)Y_{\text{soft}} + \alpha Y_{\text{hard}} \in \mathbb{R}^{n \times c}$ is collected from the output of the classification head. The distillation parameter $\alpha$, along with the compression method (either SVD or AFM), must still be selected.

To determine the best configuration, several values of $\tau$ and $\alpha$ are tested, and results are compared between SVD and AFM. The most effective models are those that retain few parameters while maintaining high accuracy. As shown in Figure 11, the best results are achieved using AFM with $\alpha = 0.1$.




(a) $\alpha = 0.1$ (b) $\alpha = 0.25$

(c) $\alpha = 0.1$ (d) $\alpha = 0.25$

Figure 11: A comparison between $\alpha = 0.1$ and $\alpha = 0.25$ is shown for the SVD and AFM compression methods, across different values of $\tau$, using the BERT model evaluated on the CoLA dataset.

The results from the uniform truncation rank in Table 8 can be compared with those obtained using the compression algorithm in Table 9. Since the resulting parameter proportion is indirectly determined by the output threshold $\tau$ in the algorithm, and directly by the truncation rank in the uniform method, a direct comparison between the two is not straightforward. To address this, results with nearly equivalent compression ratios have been highlighted in bold. From these values, it can be observed that the compression algorithm outperforms the

uniform SVD approach. This indicates that selecting a rank configuration based on the threshold $\tau$ leads to better performance than applying a uniform rank across all layers, confirming the effectiveness of the framework's strategy on this task.

| **Model** | **Dataset** | **Accuracy** | **#Params (%)** |
|---|---|---|---|
| BERT | CoLA | **0.4075** | **92.12** |
| | MRPC | **0.7574** | **92.12** |
| | RTE | **0.6029** | **92.12** |
| | QNLI | **0.7664** | **92.12** |
| | WNLI | 0.5634 | 92.12 |
| GPT-2 | CoLA | 0.3106 | 86.25 |
| | MRPC | 0.3456 | 86.25 |
| | RTE | 0.4946 | 86.25 |
| | QNLI | 0.7930 | 86.25 |
| | WNLI | 0.5634 | 86.25 |
| RoBERTa | CoLA | 0.3538 | 93.08 |
| | MRPC | 0.4853 | 93.08 |
| | RTE | 0.4729 | 93.08 |
| | QNLI | **0.6678** | **93.08** |
| | WNLI | 0.5634 | 93.08 |

Table 8: Accuracy and resulting parameters proportion using uniform SVD compression, with uniform truncation rank of 0.6 and without post fine-tuning.

| Model | Dataset | Base Benchmark | Benchmark | #Params (%) | $\tau$ |
|---|---|---|---|---|---|
| BERT | CoLA | 0.8245 / 0.5711 | **0.8006** / 0.5485 | **92.45** | 0.25 |
| | MRPC | 0.8407 / 0.8889 | **0.8260** / 0.8834 | **92.45** | 0.25 |
| | RTE | 0.6318 | **0.6606** | **90.56** | 0.50 |
| | QNLI | 0.8960 | **0.8345** | **90.29** | 0.50 |
| | WNLI | 0.2676 | 0.3099 | 96.76 | 0.25 |
| GPT-2 | CoLA | 0.7785 / 0.4383 | 0.7593 / 0.3976 | 95.26 | 0.25 |
| | MRPC | 0.8039 / 0.8684 | 0.8039 / 0.8718 | 96.68 | 0.25 |
| | RTE | 0.6318 | 0.6137 | 95.02 | 0.25 |
| | QNLI | 0.8849 | 0.8757 | 92.41 | 0.25 |
| | WNLI | 0.3380 | 0.5634 | 40.76 | 0.50 |
| RoBERTa | CoLA | 0.8313 / 0.5855 | 0.8207 / 0.5699 | 95.97 | 0.25 |
| | MRPC | 0.8775 / 0.9117 | 0.8701 / 0.9042 | 95.74 | 0.25 |
| | RTE | 0.4729 | 0.4729 | 41.98 | 0.25 |
| | QNLI | 0.8891 | **0.8666** | **93.84** | 0.25 |
| | WNLI | 0.5634 | 0.5634 | 96.21 | 0.25 |

Table 9: Benchmark results selected from $\tau = 0.25$ and $\tau = 0.5$ configurations based on highest accuracy-to-parameters ratio. For CoLA : Accuracy/MCC, for MRPC : Accuracy/F1, for the others : Accuracy. The $\tau$ column indicates which configuration showed the highest accuracy-to-parameters ratio. No post fine-tuning.

An effective compression case is illustrated in Figure 12, where the feed-forward layers were prioritized based on their high sorting criteria values and compressed using different truncation ranks, while the attention weight matrices remained unchanged. In this example, the model was compressed to 10% of its original size without any loss in accuracy and without requiring post fine-tuning.

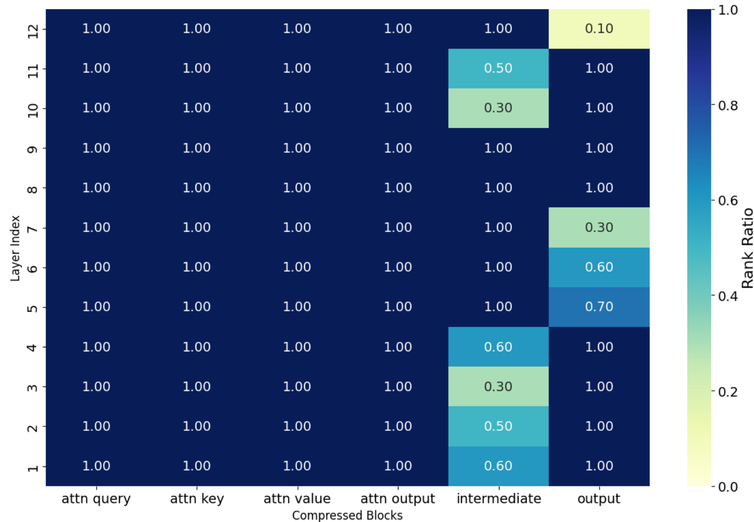



Figure 12: BERT evaluated on the RTE dataset with $\tau = 0.5$, 66.1% accuracy and 90.6% resulting parameters.

### 5.2.2 Random proxy

When the fine-tuning dataset is unavailable or too small, compression can be performed using a random proxy dataset $\mathbf{X} \in \mathbb{R}^{n \times d_{\text{model}}}$, as described in section 4.4. In this case, the proxy dataset is provided at the input of the backbone of the classification model $M^{L}_{W,Cls}$. Since no task-specific dataset is used, AFM doesn't make a lot of sense, so SVD is used instead. The results from SVD with a uniform truncation rank from Table 10 are compared with those from the compression algorithm from Table 11. Values that can be compared are highlighted in bold. The compression framework still outperforms the uniform method for comparable cases highlighted in bold (with the exception of WNLI).

| **Model** | **Dataset** | **Accuracy** | **#Params (%)** |
|---|---|---|---|
| BERT | CoLA | 0.4372 | 98.03 |
| | MRPC | 0.7475 | 98.03 |
| | RTE | **0.6209** | **98.03** |
| | QNLI | **0.7954** | **98.03** |
| | WNLI | **0.5211** | **98.03** |
| GPT-2 | CoLA | 0.3279 | 90.88 |
| | MRPC | 0.4534 | 90.88 |
| | RTE | **0.4729** | **90.88** |
| | QNLI | 0.8213 | 90.88 |
| | WNLI | 0.5634 | 90.88 |
| RoBERTa | CoLA | 0.3912 | 98.27 |
| | MRPC | 0.4853 | 98.27 |
| | RTE | 0.4729 | 98.27 |
| | QNLI | **0.6782** | **98.27** |
| | WNLI | **0.5634** | **98.27** |

Table 10: Accuracy and resulting parameters proportion using uniform SVD compression, with truncation rank of 0.65 and without post fine-tuning.

| Model | Dataset | Base Benchmark | Benchmark | #Params (%) | $\tau$ |
|---|---|---|---|---|---|
| BERT | CoLA | 0.8245 / 0.5711 | 0.8291 / 0.5814 | 95.95 | 0.50 |
| | MRPC | 0.8407 / 0.8889 | 0.8407 / 0.8889 | 100.00 | 0.10 |
| | RTE | 0.6318 | **0.6534** | **97.03** | 0.50 |
| | QNLI | 0.8960 | **0.8960** | **99.46** | 0.10 |
| | WNLI | 0.2676 | **0.3380** | **96.76** | 0.25 |
| GPT-2 | CoLA | 0.7785 / 0.4383 | 0.7785 / 0.4222 | 96.92 | 0.10 |
| | MRPC | 0.8039 / 0.8684 | 0.7819 / 0.8594 | 96.68 | 0.10 |
| | RTE | 0.6318 | **0.5812** | **89.56** | 0.25 |
| | QNLI | 0.8849 | 0.8757 | 96.92 | 0.10 |
| | WNLI | 0.3380 | 0.5634 | 76.01 | 0.50 |
| RoBERTa | CoLA | 0.8313 / 0.5855 | 0.8313 / 0.5857 | 99.05 | 0.25 |
| | MRPC | 0.8775 / 0.9117 | 0.8775 / 0.9117 | 100.00 | 0.10 |
| | RTE | 0.4729 | 0.4729 | 41.98 | 0.10 |
| | QNLI | 0.8891 | **0.8896** | **98.58** | 0.25 |
| | WNLI | 0.5634 | **0.5634** | **97.63** | 0.50 |

Table 11: Benchmark results selected from $\tau = 0.1$, $\tau = 0.25$ and $\tau = 0.5$ configurations based on highest accuracy-to-parameters ratio. For CoLA : Accuracy/MCC, for MRPC : Accuracy/F1, for the others : Accuracy. The $\tau$ column indicates which configuration showed the highest accuracy-to-parameters ratio. No post fine-tuning.

A similar outcome to the one of Figure 12 is illustrated in Figure 13, where compression was effective but involved fewer weight matrices due to the general nature of the input. This example shows that a fine-tuned model can be compressed to approximately 5% of its original size without using fine-tuning data and while preserving its full accuracy. Although the compression ratio is not very high, it serves as a proof of concept for this use case.

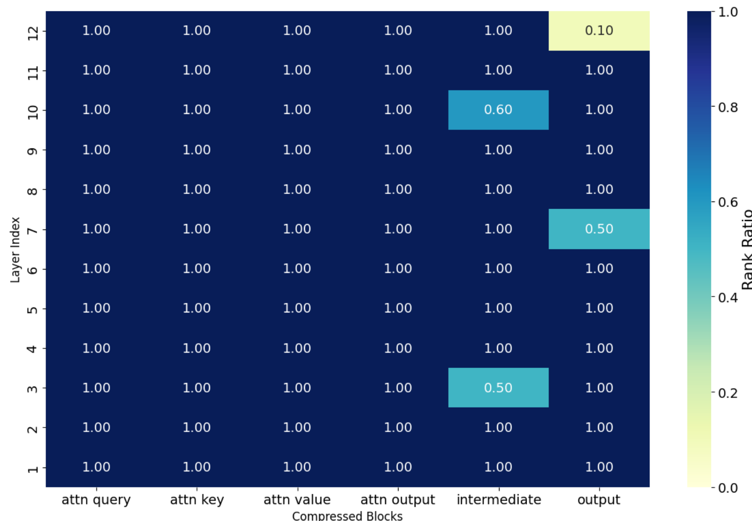



Figure 13: BERT evaluated on the CoLA dataset with $\tau = 0.5$, 82.9% accuracy and 95.9% resulting parameters.

## 5.3 Pre-trained model compression

The compression algorithm framework can also be applied to pre-trained models $M^L_{W,Pre}$, which have been trained on general tasks but not fine-tuned on specific datasets. Compressing such models is useful, as the resulting smaller models can be fine-tuned with lower computational cost and may still achieve good performance with less data.

The compression algorithm is compared with SVD compression using a uniform truncation rank. In this setting, a fine-tuning step is performed after compression to evaluate whether the compressed pre-trained model can still achieve good performance. To fine-tune the pre-trained model $M^L_{W,Pre}$, a classification head is added after the backbone, resulting in the model $M^L_{W,Cls}$. State-of-the-art results are included for reference, although a direct comparison is not appropriate. Indeed, as mentioned earlier, the word embedding layer was excluded from compression for practical reasons, even though it accounts for 20–25% of the model size, and fine-tuning was limited to 3 epochs, which may not always be sufficient for full convergence. Other limitations will be discussed later.

### 5.3.1 Task-specific proxy

Pre-trained models $M^L_{W,Pre}$ can be compressed for specific tasks by applying the compression framework. The input $X \in \mathbb{R}^{n \times n_{\text{vocab}}}$ is fed into the embedding layer, and the distillation output $Y \in \mathbb{R}^{n \times d_{\text{model}}}$ is taken from the backbone (even though distillation is typically applied at the output of a classification head). The same hyper-parameter settings and value of $\alpha$ as in Section 5.2.1 are used. After compression, a classification head is added to the backbone, and the resulting model $M^L_{W,Cls}$ is fine-tuned on specific datasets. Results for

uniform SVD compression are shown in Table 12, and those for the compression framework are shown in Figure 13. Cases with nearly equivalent compression ratios are highlighted in bold.

Unlike the case of fine-tuned models, the performance of the uniform compression method and the algorithm framework appears almost identical here. This suggests that, during fine-tuning, performance recovery depends more on the fine-tuning process than on the specific choice of truncation ranks. It is also observed that models can be strongly compressed and still recover much of their performance through fine-tuning. This supports the idea that pretrained models have a low *intrinsic dimension*, as described in [1]. According to this study, pretrained models such as BERT can recover up to 90% of their full performance on GLUE benchmark tasks by optimizing only a small fraction of their parameters. In the paper, the intrinsic dimension is defined as the smallest number of parameters needed to reach 90% of the full level of performance.

| **Model** | **Dataset** | **Post Accuracy** | **#Params (%)** |
|---|---|---|---|
| BERT | CoLA | 0.6913 | 33.94 |
| | MRPC | 0.7083 | 33.94 |
| | RTE | **0.5415** | **33.94** |
| | QNLI | **0.7756** | **33.94** |
| | WNLI | **0.5352** | **33.94** |
| GPT-2 | CoLA | **0.6913** | **40.75** |
| | MRPC | **0.6814** | **40.75** |
| | RTE | **0.4296** | **40.75** |
| | QNLI | **0.8052** | **40.75** |
| | WNLI | **0.5634** | **40.75** |
| RoBERTa | CoLA | **0.6913** | **41.97** |
| | MRPC | **0.6838** | **41.97** |
| | RTE | **0.4729** | **41.97** |
| | QNLI | **0.4946** | **41.97** |
| | WNLI | **0.5634** | **41.97** |

Table 12: Accuracy and resulting parameters proportion using uniform SVD compression, with truncation rank of 0.1 and with post fine-tuning.

| Model | Dataset | Base Benchmark | Benchmark | #Params (%) | $\tau$ |
|---|---|---|---|---|---|
| BERT | CoLA | 0.8245 / 0.5711 | 0.7940 / 0.4904 | 81.93 | 1.25 |
| | MRPC | 0.8407 / 0.8889 | 0.7304 / 0.8318 | 54.68 | 1.25 |
| | RTE | 0.6318 | **0.5776** | **33.94** | 1.25 |
| | QNLI | 0.8960 | **0.5997** | **29.80** | 1.25 |
| | WNLI | 0.2676 | **0.6056** | **28.18** | 0.75 |
| GPT-2 | CoLA | 0.7785 / 0.4383 | **0.6912** / 0.0000 | **39.29** | 1.25 |
| | MRPC | 0.8039 / 0.8684 | **0.7083** / 0.8120 | **40.76** | 1.25 |
| | RTE | 0.6318 | **0.4729** | **36.25** | 1.25 |
| | QNLI | 0.8849 | **0.8173** | **44.18** | 1.25 |
| | WNLI | 0.3380 | **0.4366** | **36.25** | 0.75 |
| RoBERTa | CoLA | 0.8313 / 0.5855 | **0.6912** / 0.0000 | **36.92** | 1.0 |
| | MRPC | 0.8775 / 0.9117 | **0.6887** / 0.8025 | **36.92** | 1.25 |
| | RTE | 0.4729 | **0.4729** | **36.92** | 0.5 |
| | QNLI | 0.8891 | **0.4946** | **36.92** | 0.5 |
| | WNLI | 0.5634 | **0.5634** | **36.92** | 0.5 |

Table 13: Benchmark results selected from $\tau = 0.5$, $\tau = 0.75$, $\tau = 1$ and $\tau = 1.25$ configurations based on highest accuracy-to-parameters ratio. For CoLA : Accuracy/MCC, for MRPC : Accuracy/F1, for the others : Accuracy. The $\tau$ column indicates which configuration showed the highest accuracy-to-parameters ratio. With post fine-tuning.

Figure 14 shows a case where the compression was effective. The model was strongly compressed by the algorithm. In this example, the attention mechanism (first column) was also compressed, as its size matches the size of the feed-forward layers (last two columns). As a result, the compression order of the weight matrices from these 3 columns was determined based on their effective rank. As reported in [1], the model was able to recover its performance by fine-tuning only a portion of its original parameters.

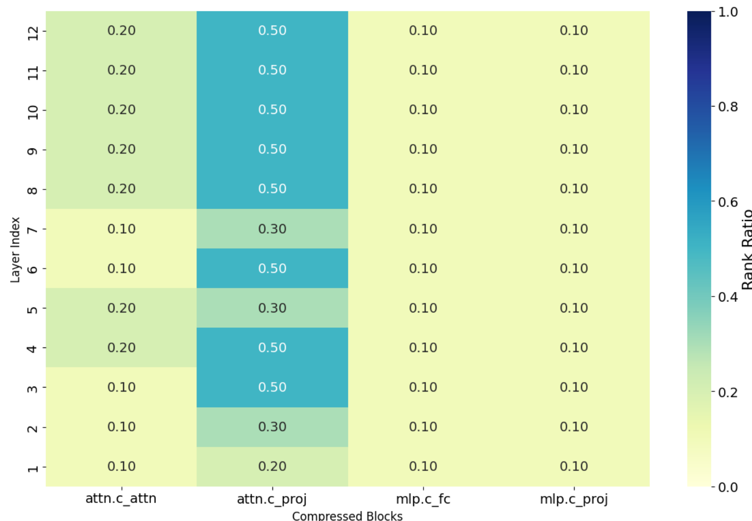



Figure 14: GPT-2 evaluated on the QNLI dataset with $\tau = 1.25$, 81.73% accuracy and 44.18% resulting parameters.

As mentioned at the beginning of this section, these results cannot be directly compared with state-of-the-art methods. The embedding layer was not compressed, and fine-tuning was limited to 3 epochs, which may not always ensure convergence. In addition, the fine-tuning process could be improved by using knowledge distillation at intermediate layers or weight matrices. For example, in [31], distillation-based improvements led to performance gains of up to 10% on some GLUE benchmark datasets compared to standard SVD compression followed by basic fine-tuning. In that case, initial performance was recovered while keeping only about 60% of the parameters. This suggests that refining the fine-tuning strategy would likely improve the results of the compression framework on pre-trained models.

Other transformer compression methods can also achieve strong results. In [25], most of the performance on the GLUE benchmark was recovered by retaining only 10 to 20% of the parameters through a method that combines low-rank and sparse approximations. Other pruning methods referenced in the same work also maintained performance while preserving a similar fraction of parameters. Quantization techniques, such as those in [3], reduced model size from about 400MB to a few dozen MB while preserving performance. In [20], INT8 quantization was applied to BERT on the GLUE benchmark, resulting in an average inference speed-up of a 3.5 factor.

### 5.3.2 Random proxy

As described in Section 5.2.2, when the fine-tuning dataset is unavailable or too small, compression can be applied using a random proxy dataset $\mathbf{X} \in \mathbb{R}^{n \times d_{\text{model}}}$. In this case, the proxy dataset is used as input to the backbone of the pretrained model $M^L_{W,Pre}$. Since no task-specific data is available, AFM is not suitable and

SVD is used instead. The results from SVD with a uniform truncation rank in Table 12 are compared with those from the compression algorithm shown in Table 14.

The performance remains similar to that obtained with uniform truncation rank when the proportion of remaining parameters is comparable (see rows highlighted in bold). As observed with fine-tuned models, the compression algorithm tends to apply less compression when a random proxy is used instead of a task-specific proxy. The main advantage of the algorithm in the case of pretrained models is that it provides a way to select compression levels that allow much of the original performance to be recovered after fine-tuning, unlike uniform compression which does not suggest how to choose the truncation rank.

| **Model** | **Dataset** | **Base Benchmark** | **Benchmark** | **#Params (%)** | $\tau$ |
|---|---|---|---|---|---|
| BERT | CoLA | 0.8245 / 0.5711 | 0.8233 / 0.5658 | 95.95 | 0.5 |
| | MRPC | 0.8407 / 0.8889 | 0.7549 / 0.8423 | 83.28 | 0.5 |
| | RTE | 0.6318 | 0.5776 | 97.03 | 0.5 |
| | QNLI | 0.8960 | 0.8940 | 97.57 | 0.5 |
| | WNLI | 0.2676 | 0.5634 | 64.83 | 0.75 |
| GPT-2 | CoLA | 0.7785 / 0.4383 | 0.6901 / -0.0207 | 78.76 | 0.5 |
| | MRPC | 0.8039 / 0.8684 | 0.7206 / 0.8213 | 78.67 | 0.5 |
| | RTE | 0.6318 | 0.5307 | 78.67 | 0.5 |
| | QNLI | 0.8849 | 0.8700 | 83.87 | 0.5 |
| | WNLI | 0.3380 | **0.5634** | **40.75** | 1.0 |
| RoBERTa | CoLA | 0.8313 / 0.5855 | 0.8301 / 0.5830 | 98.34 | 0.5 |
| | MRPC | 0.8775 / 0.9117 | 0.8775 / 0.9144 | 96.21 | 0.5 |
| | RTE | 0.4729 | **0.4729** | **41.97** | 1.0 |
| | QNLI | 0.8891 | 0.8896 | 98.58 | 0.25 |
| | WNLI | 0.5634 | **0.5634** | **41.97** | 1.0 |

Table 14: Benchmark results selected from $\tau = 0.5$, $\tau = 0.75$, $\tau = 1$ and $\tau = 1.25$ configurations based on highest accuracy-to-parameters ratio. For CoLA : Accuracy/MCC, for MRPC : Accuracy/F1, for the others : Accuracy. The $\tau$ column indicates which configuration showed the highest accuracy-to-parameters ratio. With post fine-tuning

The architecture of a successfully compressed model is shown in Figure 15, where the feed-forward layers were once again compressed in priority.

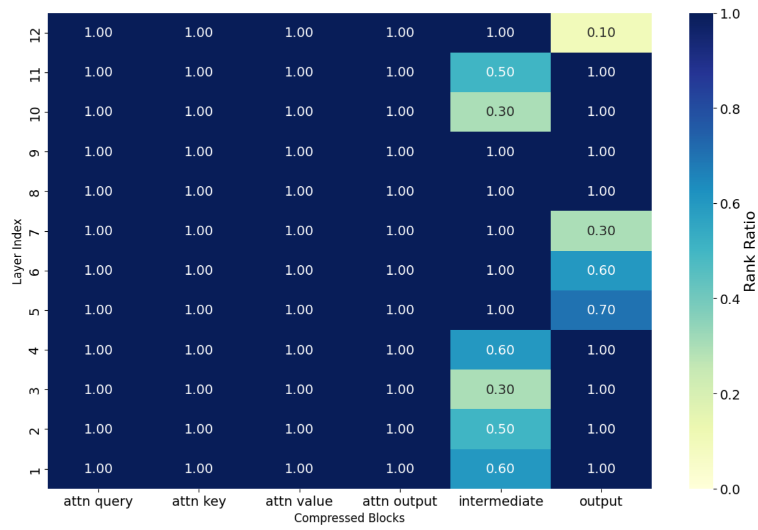



Figure 15: GPT-2 evaluated on the QNLI dataset with $\tau = 0.5$, 87.0% accuracy and 83.9% resulting parameters.

# 6 Conclusion

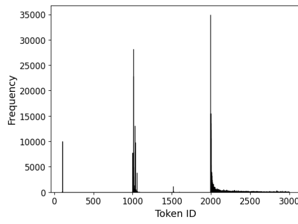
In this work, a compression algorithm framework for neural networks was developed and its performance on Transformer models was evaluated. Since the framework is highly modular, much effort was devoted to selecting among the various options it provides. This process enabled the exploration of rank metrics, with effective rank ultimately chosen as the most suitable, as well as output features, error measures, and proxy datasets. Method were designed to compress neural network models (specifically Transformers) using some task-specific proxy or without using any actual data (i.e. by employing a random proxy to model the distribution of the embedding layer's outputs). Although these methods outperformed the baseline approach for compressing fine-tuned models (demonstrating that the chosen truncation rank configuration affects compression quality) they performed similarly to the baseline when compressing pretrained models followed by fine-tuning. In that case, the fine-tuning step alone was sufficient to recover the original performance. Therefore, potential improvements may arise from enhancing the fine-tuning process (for example, through feature distillation) and from compressing the embedding layer, which was not addressed here for practical reasons despite its substantial size.

In conclusion, the low-rank compression framework developed in this study has shown promising results in certain scenarios but could be improved through further optimization.

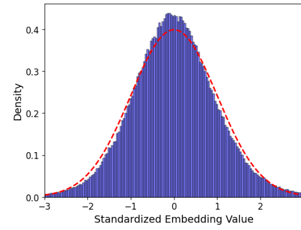
# 7 Appendix A

As described in Section 4.4, the output of the embedding layer was computed for the entire WikiText dataset [29]. These outputs were then flattened and standardized, resulting in a distribution that closely resembles a Gaussian. To verify this resemblance, a QQ-plot was generated after removing a small number of extreme outliers. These outliers correspond to feature values lying outside the interval $[-3, 3]$ and account for only a very small fraction of the data. For example, in the case of BERT, they represent approximately 0.1% of the total points. However, given the large number of data points (26.88M), this small percentage still translates into a large absolute number of outliers, which can distort the appearance of the QQ-plot. For this reason, they were excluded from the plot purely for visualization purposes. For the same reasons, quantitative statistical tests for normality were not applied to this data. Tests such as the Kolmogorov–Smirnov test [28] are known to be sensitive to outliers, while the Shapiro–Wilk test [35] is designed for small sample sizes and becomes impractical when dealing with large datasets.

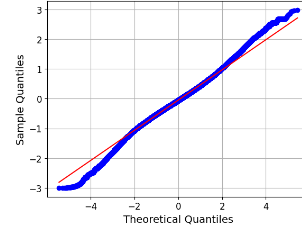
The same figure as in Figure 5 is extended in Figure 16 to include results for GPT-2 and RoBERTa. It is observed that GPT-2's output distribution remains approximately Gaussian, though slightly less so than BERT. In the case of RoBERTa, the distribution appears to consist of two overlapping Gaussian-like components, although its QQ-plot remains nearly linear. These observations further support the idea that the output of the word embedding layer can be reasonably modeled by a Gaussian distribution.



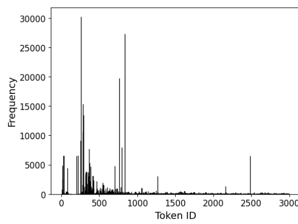

BERT token ID histogram



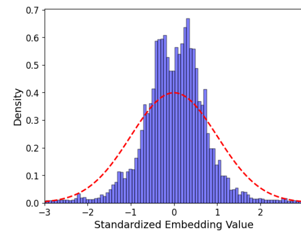

BERT word embedding output's standardized distribution (blue) with $\mathcal{N}(0,1)$ (red)



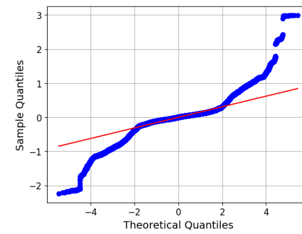

BERT QQ-plot



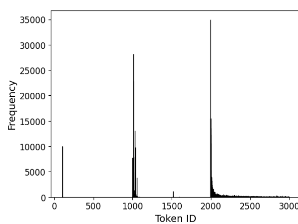

GPT-2 token ID histogram



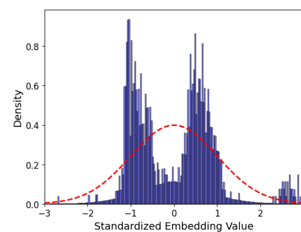

GPT-2 word embedding output's standardized distribution (blue) with $\mathcal{N}(0,1)$ (red)



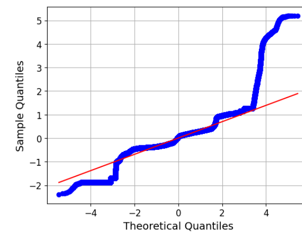

GPT-2 QQ-plot




RoBERTa token ID histogram




RoBERTa word embedding output's standardized distribution (blue) with $\mathcal{N}(0,1)$ (red)




RoBERTa QQ-plot

Figure 16: Comparison between the distribution of raw token IDs and the distribution of embedding outputs from transformer models on the WikiText dataset [24, 29], using BERT, GPT-2, and RoBERTa. While token IDs are discrete and non-Gaussian, the continuous embedding outputs approximate a Gaussian distribution after standardization.

# 8 Appendix B

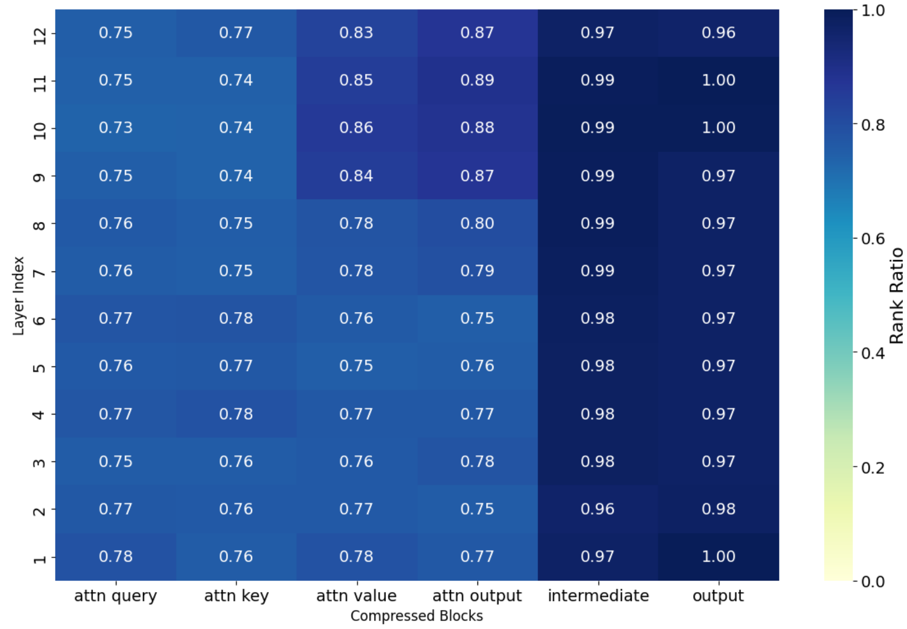



Figure 17: The normalized effective rank is shown for the RoBERTa model with $K = 72$ weight matrices. The first three columns represent, respectively, the query, key, and value weight matrices, with their layer index indicated on the y-axis ranging from 1 to 12. The fourth column corresponds to the attention projection weight matrix, while the last two columns represent the feed forward layers. The maximal effective rank value is 708.6

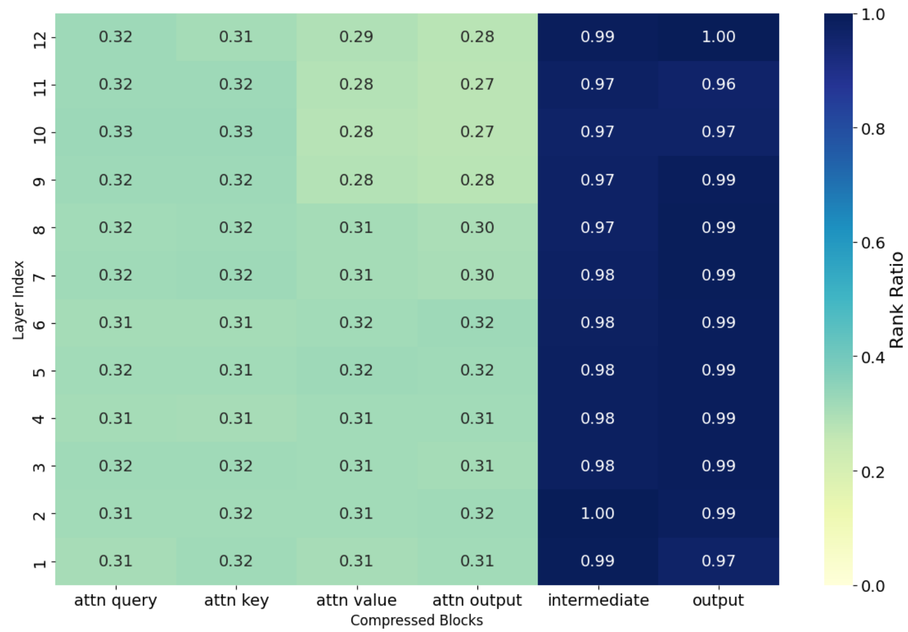



Figure 18: The normalized value metric computed as the ratio between the maximal dimension of a given weight matrix and its effective rank is shown for the RoBERTa model with $K = 72$ weight matrices. The first three columns represent, respectively, the query, key, and value weight matrices, with their layer index indicated on the y-axis ranging from 1 to 12. The fourth column corresponds to the attention projection weight matrix, while the last two columns represent the feed forward layers.

# Use of AI

In this work, AI was used to design the functions used to generate figures, correct some bugs that needed a deep dive into technical documentation and reformulating some sentences from the written report.

# References

[1] Armen Aghajanyan, Luke Zettlemoyer, and Sonal Gupta. Intrinsic dimensionality explains the effectiveness of language model fine-tuning. *arXiv preprint arXiv:2012.13255*, 2020.

[2] Michael Artin. Row rank = column rank, 2010. 18.701 Algebra I, Fall 2010, MIT OpenCourseWare.

[3] Haoli Bai, Wei Zhang, Lu Hou, Lifeng Shang, Jing Jin, Xin Jiang, Qun Liu, Michael Lyu, and Irwin King. Binarybert: Pushing the limit of bert quantization. *arXiv preprint arXiv:2012.15701*, 2020.

[4] Lokesh Borawar and Ravinder Kaur. Resnet: Solving vanishing gradient in deep networks. In *Proceedings of International Conference on Recent Trends in Computing: ICRTC 2022*, pages 235–247. Springer, 2023.

[5] Emmanuel J Candès and Terence Tao. Robust uncertainty principles: Exact signal reconstruction from highly incomplete frequency information. *IEEE Transactions on Information Theory*, 52(2):489–509, 2006.

[6] Guangliang Chen. Lecture 5: Singular value decomposition (svd). https://www.sjsu.edu/faculty/guangliang.chen/Math253S20/lec5svd.pdf, 2020. San José State University, Department of Mathematics & Statistics.

[7] Tianlong Chen, Jonathan Frankle, Shiyu Chang, Sijia Liu, Yang Zhang, Zhangyang Wang, and Michael Carbin. The lottery ticket hypothesis for pre-trained bert networks. *Advances in neural information processing systems*, 33:15834–15846, 2020.

[8] Davide Chicco and Giuseppe Jurman. The advantages of the matthews correlation coefficient (mcc) over $f_1$ score and accuracy in binary classification evaluation. *BMC Genomics*, 21:6, 2020.

[9] Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, et al. Rethinking attention with performers. *arXiv preprint arXiv:2009.14794*, 2020.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, pages 4171–4186, 2019.

[11] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635*, 2018.

[12] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, MD, 4 edition, 2013.

[13] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy, 2020.

[14] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[15] Yerlan Idelbayev and Miguel A Carreira-Perpinán. Low-rank compression of neural nets: Learning the rank of each layer. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8049–8059, 2020.

[16] Ilse C.F. Ipsen and Arvind K. Saibaba. Stable rank and intrinsic dimension of real and complex matrices. *arXiv preprint arXiv:2407.21594*, 2024.

[17] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2704–2713, 2018.

[18] Hyeji Kim, Muhammad Umar Karim Khan, and Chong-Min Kyung. Efficient neural network compression. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 12569–12577, 2019.

[19] Hyeji Kim and Chong-Min Kyung. Automatic rank selection for high-speed convolutional neural network. *arXiv preprint arXiv:1806.10821*, 2018.

[20] Sehoon Kim, Amir Gholami, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. I-bert: Integer-only bert quantization. In *International conference on machine learning*, pages 5506–5518. PMLR, 2021.

[21] N Kishore Kumar and Jan Schneider. Literature survey on low rank approximation of matrices. *Linear and Multilinear Algebra*, 65(11):2212–2244, 2017.

[22] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451*, 2020.

[23] Solomon Kullback and Richard A. Leibler. On information and sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86, 1951.

[24] Quentin Lhoest, Albert Villanova del Moral, Yacine Jernite, Numa Thain, Suraj Patil, Patrick von Platen, Julien Chaumond, Mariama Drame, Julien Plu, Victor Sanh, Thomas Wolf, Teven Le Scao, Sylvain Gugger, Francisco

Lagunas, Alexander M. Rush, and Thomas Wolf. Datasets: A community library for natural language processing. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 175–184. Association for Computational Linguistics, 2021.

[25] Yixiao Li, Yifan Yu, Qingru Zhang, Chen Liang, Pengcheng He, Weizhu Chen, and Tuo Zhao. Losparse: Structured compression of large language models based on low-rank and sparse approximation. In *International Conference on Machine Learning*, pages 20336–20350. PMLR, 2023.

[26] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[27] Ivan Markovsky and Florian Dörfler. Behavioral systems theory in data-driven analysis, signal processing, and control. *Annual Reviews in Control*, 52:42–64, 2021.

[28] Frank J Massey Jr. The kolmogorov-smirnov test for goodness of fit. *Journal of the American statistical Association*, 46(253):68–78, 1951.

[29] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

[30] Paul Michel, Omer Levy, and Graham Neubig. Are sixteen heads really better than one? *Advances in neural information processing systems*, 32, 2019.

[31] Matan Ben Noach and Yoav Goldberg. Compressing pre-trained language models by matrix decomposition. In *Proceedings of the 1st Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics and the 10th International Joint Conference on Natural Language Processing*, pages 884–889, 2020.

[32] David M W Powers. Evaluation: From precision, recall and f-measure to roc, informedness, markedness and correlation. *Journal of Machine Learning Technologies*, 2(1):37–63, 2011.

[33] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8), 2019.

[34] Olivier Roy and Martin Vetterli. The effective rank: A measure of effective dimensionality. In *2007 15th European signal processing conference*, pages 606–610. IEEE, 2007.

[35] Samuel S Shapiro and Martin B Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3-4):591–611, 1965.

[36] Marina Sokolova and Guy Lapalme. A systematic analysis of performance measures for classification tasks. *Information Processing Management*, 45(4):427–437, 2009.

[37] Vikas Sindhwani Ebru Arisoy Bhuvana Ramabhadran Tara N. Sainath, Brian Kingsbury. Low-rank matrix factorization for deep neural network training with high-dimensional output targets. *ICASSP*, 2013.

[38] A Vaswani. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

[39] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*, 2018.

[40] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019.

[41] Hao Yu and Jianxin Wu. Compressing transformers: features are low-rank, but weights are not! In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 11007–11015, 2023.

[42] Xiangyu Zhang, Jianhua Zou, Kaiming He, and Jian Sun. Accelerating very deep convolutional networks for classification and detection. *IEEE transactions on pattern analysis and machine intelligence*, 38(10):1943–1955, 2015.

**UNIVERSITÉ CATHOLIQUE DE LOUVAIN**
**École polytechnique de Louvain**
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**