Efficient GPT Model Pre-training using Tensor Train Matrix Representation

Viktoria Chekalina[1] **Georgiy Novikov**[1] **Julia Gusak**[1*]
Alexander Panchenko[1,3] **Ivan Oseledets**[1,3]
[1]Skolkovo Institute of Science and Technology,
[2]HSE University, [3]Artificial Intelligence Research Institute
{v.chekalina, g.novikov, y.gusak, a.panchenko, i.oseledets}@skol.tech

Abstract

Large-scale transformer models have shown remarkable performance in language modelling tasks. However, such models feature billions of parameters, leading to difficulties in their deployment and prohibitive training costs from scratch. To reduce the number of parameters in the GPT-2 (Radford et al., 2019) architecture, we replace the matrices of fully-connected layers with the corresponding Tensor Train Matrix (TTM) (Oseledets, 2010) structure. Finally, we customize forward and backward operations through the TTM-based layer for simplicity and the stability of further training. The resulting GPT-2-based model stores up to 40% fewer parameters, showing the perplexity comparable to the original model. On the downstream tasks, including language understanding and text summarization, the model performs similarly to the original GPT-2 model. The proposed tensorized layers can be used to efficiently pre-train other Transformer models.

1 Introduction

Large language models such as GPT-2, GPT-3 (Radford et al., 2019; Brown et al., 2020) show outstanding results in all areas of natural language processing. However, training and employing models with a vast number of parameters requires memory, time, and electricity proportional to model size.

The goal of our study is to reduce the effective size of the model (number of parameters) and, as a result, to reduce the GPU memory used. We measured the aggregated memory in the different GPT-2 modules. We saw that the "thickest" modules are fully connected layers (see Table 1) and focused on making them more efficient.

To make GPT-2-based models easier to deploy, we replaced fully connected layers with sequential TTM (Oseledets, 2010) containers, based on Tensor Train (TT) (Oseledets, 2011) representation. We tested several approaches to forward and back propagations of a signal through containers and chose the most memory-stable and time-optimal pattern. We train the architecture with custom TTM layers from scratch and then study the behaviour of the pre-trained custom model on in-domain and out-off-domain language modelling tasks and several downstream tasks.

Table 1: Size of memory stored in different modules of Transformer-based architecutre GPT-2.

Module / GPT-2	Small	Medium	Large
Attention	9.01 MB	16.02 MB	25.02 MB
MLP	18.01 MB	32.02 MB	50.02 MB

The contribution of our paper is the following: (i) We develop a custom TTM-layer that, firstly, has fewer parameters and, secondly, uses less memory during forward and backward passes; (ii) We provide a GPT-based model with up to 40% fewer parameters showing performance close to the original GPT in in-domain and out-of-domain tasks language modelling, GLUE benchmark, and text summarization.

The sourse code are available online [1].

2 Related work

Several approaches explore ways to reduce the size of language models. The distillation mechanism (Hinton et al., 2015) was applied to BERT (Sanh et al., 2019) and GPT-2[2]. The Open Pre-trained Transformers (OPT) (Zhang et al., 2022) provide a smaller model that mimics the behaviour of GPT-3 (Brown et al., 2020). They employ more efficient training and use particular datasets to improve generalisation ability.

TT (Tensor Train) is an effective way to obtain low-rank representations of inner layers and is also

[*] Work has been done while at Skoltech. Now with INRIA, University of Bordeaux, France.

[1] https://github.com/sayankotor/GreenAI
[2] https://huggingface.co/distilgpt2

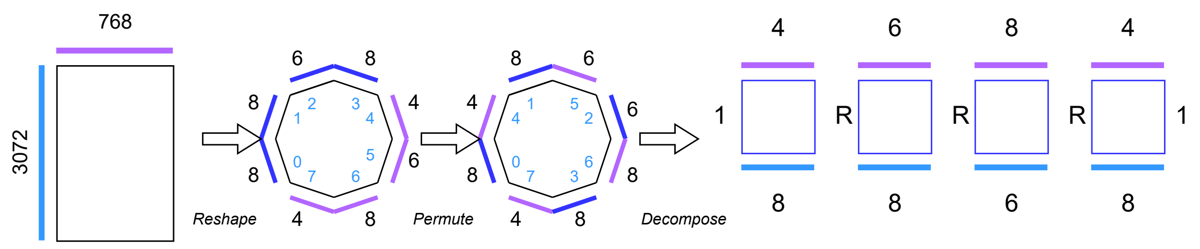

Figure 1: The scheme of 4-cores TTM representation of weight matrix in the GPT-2 small FC layer. The dimentions of the initial matrix are decomposed into 4 factors. The matrix ia reshaped to these factors. Than axis are permuted in a way that input and output dimensions are adjacent. Black digits indicate the size of the axes, and light blue - their number.

used to reduce parameter numbers. (Khrulkov et al., 2019) and (Yin et al., 2021) reduce the size of the embedding layer using TT. Novikov et al. (2015) uses the TT format of linear layers to compress the computer vision models; however, TT representations were not tested before for generative Transformers.

3 Singular Value decomposition (SVD) Layer

We compress the initial model by replacing fully-connected layers with their SVD analogues.

More precisely, assuming that W is a layer weight matrix, we define SVD as follows: $W = U\Sigma V^T$. Then we use truncated products of it $U_r = U[:,:r], \Sigma_r = \Sigma[:r,:r], V_r = V[:,:r]$ to define weights for two sequential linear layers, with which we will replace the current:

$$W_2 = U[:,:r]\sqrt{\Sigma_r}, W_1 = \sqrt{\Sigma_r}V^T[:r,:] \quad (1)$$

As a result, we get an approximation of the linear matrix $W \approx W_2 W_1$ and an approximation of the initial layer $Y \approx XW_1^T W_2^T + b$.

If W have n_{in}, n_{out} shape, the number of parameters in the layer before compression is $n_{in} \times n_{out}$, after representation by truncated SVD, the number of parameters in the layer is $r \times (n_{in} + n_{out})$.

4 Math Background

Here we provide some mathematical notation which is used to represent matrix or tensor objects in the TTM format. This format is used in the TTM linear-like neural network layers.

We denote vectors as $\mathbf{v}$, matrices as $\mathbf{M}$ and tensors of 3-rd order and higher as $\mathcal{T}$.

Tensor contraction. Given two tensors $\mathcal{T}^1 \in \mathbb{R}^{I_1\times\cdots\times I_M\times S_1\times\cdots\times S_K}$ and $\mathcal{T}^2 \in \mathbb{R}^{S_1\times\cdots\times S_K\times J_1\times\cdots\times J_N}$ the result of *tensor contraction* along axis $s_1,\ldots,s_K$ is a tensor $\mathcal{T} \in \mathbb{R}^{I_1\times\cdots\times I_M\times J_1\times\cdots\times J_M}$, where one element is computed using formula

$$\mathcal{T}_{i_1,\ldots,i_M,j_1,\ldots,j_N} =$$
$$= \sum_{s_1,\ldots,s_K} \mathcal{T}^1_{i_1,\ldots,i_M,s_1,\ldots,s_K} \mathcal{T}^2_{s_1,\ldots,s_K,j_1,\ldots,j_N}$$

and requires $O(S_1S_2\ldots S_K) = O\left(\prod_{k=1}^{K} S_k\right)$ floating point operations (FLOP). Thus, number of FLOP to compute tensor $\mathcal{T}$ is $O\left(\prod_{m=1}^{M} I_m \prod_{n=1}^{N} J_n \prod_{k=1}^{K} S_k\right)$. For example, a multiplication of two matrices of shapes (I,S) and (S,J) can be calculated for $O(IJS)$ operations.

TTM format. We say that a tensor $\mathcal{T} \in \mathbb{R}^{I_1\times J_1\times\cdots\times I_M\times J_M}$ is represented in *Tensor Train Matrix (TTM) format* with rank $(R_0, R_1, \ldots, R_M)$ if each element is computed as

$$\mathcal{T}_{i_1,j_1,\ldots,i_M,j_M} =$$
$$\sum_{r_1,\ldots,r_{M-1}} \mathcal{G}^1_{r_0,i_1,j_1,r_1} \ldots \mathcal{G}^M_{r_{M-1},i_M,j_M,r_M},$$

where $\mathcal{G}^m \in \mathbb{R}^{R_{m-1}\times I_m\times J_m\times R_m}$, $m = \overline{1,M}$ are *core tensors (cores)* of TTM decomposition. Note that $R_0 = R_M = 1$.

Assume that $R_m = R$ for $m = \overline{1,M-1}$, then to represent tensor $\mathcal{T}$ with $\prod_{m=1}^{M} I_m J_m$ elements we need to store only $R(I_1J_1 + I_MJ_M) + R^2\sum_{m=2}^{M-1} I_mJ_m$ parameters of core tensors and compression rate is:

$$c\_rate = \frac{R(I_1J_1 + I_MJ_M) + R^2\sum_{m=2}^{M-1} I_mJ_m}{\prod_{m=1}^{M} I_mJ_m} \quad (2)$$

5 Efficient TTM Layer

In our paper we focus on replacing linear layers with TTM layers. In TTM layer the weight matrix $\mathbf{W}$ of shape $D_{in} \times D_{out}$ is represented as M-dimensional tensor $\mathcal{W}_{\mathcal{M}} \in \mathbb{R}^{I_1 \times J_1 \times \cdots \times I_M \times J_M}$, where I_m and J_m are such that $D_{in} = \prod_{m=1}^{M} I_m$ and $D_{out} = \prod_{m=1}^{M} J_m$. $\mathcal{W}_{\mathcal{M}}$, in turn, is represented as a set of M cores $\mathcal{G}$, so every element in $\mathcal{W}_{\mathcal{M}}$ is enumerated by a 2M-tuple of indices and is defined as:

$$\mathcal{W}_{\mathcal{M}}((i_1, j_1), \ldots, (i_M, j_M)) =$$
$$\mathcal{G}^1(:, i_1, j_1, :)\mathcal{G}^2(:, i_2, j_2, :)...\mathcal{G}^{\mathcal{M}}(:, i_M, j_M, :)$$

as it is described in Section 4. We call TTM layer with rank R a TTM-R layer and forward pass through this layer is described by the formula:

$$\mathcal{Y}(j_1, \ldots, j_M) = \sum_{i_1,\ldots,i_M} \mathcal{W}_{\mathcal{M}}((i_1, j_1), \ldots, (i_M, j_M))\mathcal{X}(i_1, \ldots, i_M), \quad (3)$$

So, in Equation 3 we should contract activation $\mathcal{X}$ with sequence ($\mathcal{G}^M$, ..., $\mathcal{G}^1$) sequentially. Please note that we can start with the first core ($\mathcal{G}^1$, ...$\mathcal{G}^M$) or with the last ($\mathcal{G}^M$, ..., $\mathcal{G}^1$), in general it doesn't matter.

We contract $\mathcal{X}$ with size (B, D_{in}) to $\mathcal{G}_{\mathcal{M}}$ with size $(R_{M-1}, I_M, J_M, 1)$. As $D_{in} = \prod(I_1 \ldots I_M)$, we contract over I_M and have a tensor of shapes $(B, R_{M-1}, J_M, I_{M-1}, \ldots, I_1)$ as a result. Then, we should contract this tensor with a core $\mathcal{G}_{M-1}$ with shapes

$$(R_{M-2}, I_{M-1}, J_{M-1}, R_{M-1})$$

over dimensions $I_{M-1}R_{M-1}$. This operation produces the object of shapes

$$(B, R_{M-2}, J_M, J_{M-1}, I_{M-2}, \ldots, I_1)$$

By repeating such operation K times, we obtain a product with shapes

$$(B, I_1, ..., I_K, J_{K+1}, ..., J_M, R_K)$$

In the end, we gain the output of sizes $(B, J_1, \ldots J_M) = (B, D_{out})$. The computational complexity of this operation is estimated above.

We measure peak memory during one training iteration in the GPT-2 model with TTM layers with different ranks. Experiments depicted in 3 show that on a rank 16 the TTM layer can be more memory-consuming than the regular FC layer. The memory footprint for the FC and TTM layers for custom-defined and PyTorch signal propagation strategies (Table 3) confirms this claim.

Table 2: Peak memory footprints for signal propagation in full GPT-2 model with TTM layers with different ranks. At the rank 16 we have an increment in memory consumption.

Layer Type	TTM-16	TTM-32	TTM-64	Fully Connected
Memory, GB	75.07	48.7	48.31	48.37

Table 3: Memory footprints for signal propagation in TTM wiht rank 16 and Fully-Connected Layers. PyTorch strategy leads to memory costs for TTM.

Layer	TTM-16	TTM-16	Fully Connected
Backprop Strategy	PyTorch Autodiff	Einsum Full Matrix	PyTorch Autodiff
Single Layer, Batch 16	1100 MB	294 Mb	395 Mb

For a tensor contraction, the PyTorch framework uses Einstein summation notation. Optimized Einsum library (Smith and Gray, 2018) optimizes the expression's contraction order by looking for an optimal *path* - a set of strings of the form "ikl,lkj->ij". By default optimization, the obtained paths are time-optimized, not memory-optimized. We extend existing research by proposing memory-efficient techniques to compute forward and backwards through the TTM layer for a more comprehensive description of the proposed methods.

5.1 Forward Pass

Algorithm 1 Forward pass (FC layer). Number of layer parameters is $O(BD_{in}D_{out})$. Computational complexity is $O(BD_{in}D_{out})$. SavedActivations is $O(BD_{in})$.

```
Input: data $\mathbf{X} \in \mathbb{R}^{B \times D_{in}}$; parameters $\mathbf{W} \in \mathbb{R}^{D_{in} \times D_{out}}$,
  $\mathbf{b} \in \mathbb{R}^{D_{out}}$;
Output: $\mathbf{Y} \in \mathbb{R}^{B \times D_{out}}$
  $\mathbf{Y} = \mathbf{XW} + \mathbf{b}$
```

Fully-connected layer. Given an input batch $\mathbf{X} \in \mathbb{R}^{B \times D_{in}}$ a forward pass through a fully-connected layer with weight matrix $\mathbf{W} \in \mathbb{R}^{D_{in} \times D_{out}}$ and bias vector $\mathbf{b} \in \mathbb{R}^{D_{out}}$ results in the output $\mathbf{Y} = \mathbf{XW} + \mathbf{b} \in \mathbb{R}^{B \times D_{out}}$ and requires $O(BD_{in}D_{out})$ operations.

The schedule of contractions computed during the forward pass is optimized via *opt_einsum* function (Smith and Gray, 2018). This function optimizes the time of expression contraction in the BLAS library for common linear algebra operations. The default optimization strategy provides a recursive depth-first search over all possible paths, by pruning candidates that exceed the best time. Thus, due to some shared intermediate results memory for saved activations might be optimized.

TTM layer: Fixed Schedule The order of cores to contract with is fixed in advance, and we do not optimize it with *opt_einsum*. In this case, saved activations usually occupy the same amount of memory. The fixed scheduler approach is equivalent to sequential forward pass through M linear layers. The order of contraction in the schedule might be $(M, M-1, \dots, 1)$ or $(1, 2, \dots, M-1, M)$.

Algorithm 2 Forward pass (TTM layer, Fixed Scheduler). The number of layer parameters is $O(\sum_{m=1}^{M} R_{m-1} I_m J_m R_m)$.

```
Input: data $\mathbf{X} \in \mathbb{R}^{B \times D_{in}}$; $D_{in} = \prod_{m=1}^{M} I_m, D_{out} = \prod_{m=1}^{M} J_m$;
  parameters $\mathcal{G}^m \in \mathbb{R}^{R_{m-1} \times I_m \times J_m \times R_m}, m = \overline{1, M}$,
  $R_0 = R_M = 1$;
Output: $\mathcal{Y} \in \mathbb{R}^{B \times J_1 \times \dots \times J_M}$
  $\mathcal{X} = Reshape(\mathbf{X}) \in \mathbb{R}^{B \times I_1 \times \dots \times I_M}$
  $\mathcal{Y}_0 := \mathcal{X}$
  $ContractionSchedule := (1, 2, \dots, M)$
  for $k$ in $ContractionSchedule$ do
    $\mathcal{Y}_k := einsum(\mathcal{G}^k, \mathcal{Y}_{k-1})$
        ▷ $FLOP_{\mathcal{Y}} = O(B \prod_{m=1}^{k+1} J_m \prod_{m=k+1}^{M} I_m R_k R_{k+1})$
    $\mathcal{Y} = \mathcal{Y}_k$
        ▷ $Memory_{\mathcal{Y}} = O(B \prod_{m=1}^{k} J_m \prod_{m=k+1}^{M} I_m R_k)$
  end for
```

The forward pass with a Fixed Scheduler approach is equivalent to a sequential forward pass through M linear layers. The order of contraction in the schedule might be either $(M, M-1, \dots, 1)$ or $(1, 2, \dots, M-1, M)$.

5.2 Backward Pass

While training neural networks, intermediate activations are saved during the forward pass to compute gradients during the backward pass.

Fully-connected layer. For the layer $\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$ a derivatives w.r.t. weight is computed as

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}} \mathbf{x}^T.$$

Algorithm 3 Forward pass (TTM layer, Einsum). Number of layer parameters is $O(\sum_{m=1}^{M} R_{m-1} I_m J_m R_m)$.

```
Input: data $\mathbf{X} \in \mathbb{R}^{B \times D_{in}}$; $D_{in} = \prod_{m=1}^{M} I_m, D_{out} = \prod_{m=1}^{M} J_m$;
  parameters $\mathcal{G}^m \in \mathbb{R}^{R_{m-1} \times I_m \times J_m \times R_m}, m = \overline{1, M}$,
  $R_0 = R_M = 1$;
Output: $\mathcal{Y} \in \mathbb{R}^{B \times J_1 \times \dots \times J_M}$
  $\mathcal{X} = Reshape(\mathbf{X}) \in \mathbb{R}^{B \times I_1 \times \dots \times I_M}$
  $\mathcal{Y} := einsum(\mathcal{G}^1, \dots \mathcal{G}^M, \mathcal{Y})$
```

TTM layer: Automatic Pytorch differentiation (Autodiff). Automatic Pytorch differentiation during backpropagation through the TTM layer results in storing many intermediate activations, as the TTM layer is considered as a sequence of linear layers (where the number of layers corresponds to the number of core tensors).

We propose several ways to perform a backward pass that require smaller memory consumption.

TTM layer: Full Einsum. In the first approach for each core tensor $\mathcal{G}_m$ we compute gradient of loss with respect to its parameters:

$$\frac{\partial \mathcal{L}}{\partial \mathcal{G}_m} = \frac{\partial \mathcal{L}}{\partial \mathbf{W}} \frac{\partial \mathbf{W}}{\partial \mathcal{G}_m} = \mathbf{X}^T \frac{\partial \mathcal{L}}{\partial \mathcal{Y}} \frac{\partial \mathbf{W}}{\partial \mathcal{G}_m}. \quad (4)$$

As a gradient computation might be considered as a tensor contraction along the specified axis, the process includes three main steps.

Firstly, we generate a string-type expression, which specifies the shapes of input and resulting tensors (e.g. "ikl,lkj->ij" for performing tensor contraction along two axes). Second, the schedule of contraction is defined (e.g., firstly along axis 'l' and then along axis 'k'). And thirdly, einsum computation is performed.

In Full Einsum approaches, the first two steps (expression generation, contraction scheduling) are performed independently for all $\frac{\partial \mathcal{L}}{\partial \mathcal{G}_m}$. The third step, in turns, tracks which contractions are computed for different derivatives and allows sharing of intermediate results. Due to this sharing, we get memory savings compared to the Autodiff approach.

TTM layer: Full Matrix. In the Full Matrix approach, we perform the same three steps as in the Full Einsum approach. The difference is that as a first contraction we convolve the tensors $\mathcal{X}$ and $\frac{\partial \mathcal{L}}{\partial \mathcal{Y}}$ along the batch axis, and the schedule of the other contractions is further optimized. This provides complexity improvements when batch size is large (which is the case in Transformer-like models, where batch dimension is the product of batch size by sequence length).

Algorithm 4 Backward pass (TTM layer, Autodiff).

```
Input: $\frac{\partial L}{\partial \mathcal{Y}}$; saved activations from forward $\mathcal{Y}_1, \dots \mathcal{Y}_M$
Output: $\frac{\partial L}{\partial \mathcal{X}}, \frac{\partial L}{\partial \mathcal{G}^1}, \dots, \frac{\partial L}{\partial \mathcal{G}^M}$
  $\frac{\partial L}{\partial \mathcal{Y}_M} = \frac{\partial L}{\partial \mathcal{Y}}$
  for $k$ in $\{M, \dots, 1\}$ do
    $\frac{\partial L}{\partial \mathcal{G}^k} = einsum(\mathcal{Y}_{k-1}, \frac{\partial L}{\partial \mathcal{Y}^k})$
    $\frac{\partial L}{\partial \mathcal{Y}_{k-1}} = einsum(\frac{\partial L}{\partial \mathcal{Y}_k}, \mathcal{G}^k)$
  end for
  $\frac{\partial L}{\partial \mathcal{X}} = \frac{\partial L}{\partial \mathcal{Y}_0}$
```

Algorithm 5 Backward pass (TTM layer, Full Einsum).

```
Input: $\frac{\partial L}{\partial \mathcal{Y}}$; $\mathcal{X}$
Output: $\frac{\partial L}{\partial \mathcal{X}}, \frac{\partial L}{\partial \mathcal{G}^1}, \dots, \frac{\partial L}{\partial \mathcal{G}^M}$
  ▷ Results in the below for-cycle are computed only for the first batch during training and reused for others.
  for $k$ in $\{1, \dots, M\}$ do
    Compose $einsum_k$ expression for $\frac{\partial L}{\partial \mathcal{G}^k}$
    Optimize contraction schedule for composed $einsum_k$
  end for
  for $k$ in $\{1, \dots, M\}$ do
    $\frac{\partial L}{\partial \mathcal{G}^k} = einsum_k(\frac{\partial L}{\partial \mathcal{Y}}, \mathcal{G}^1, \dots, \mathcal{G}^M)$
  end for
```

Backward with Full Einsum approach in the worst case has the same complexity as the backward Autodiff.

Algorithm 6 Backward pass (TTM layer, Full Matrix).

```
Input: $\frac{\partial L}{\partial \mathcal{Y}}$; $\mathcal{X}$
Output: $\frac{\partial L}{\partial \mathcal{X}}, \frac{\partial L}{\partial \mathcal{G}^1}, \dots, \frac{\partial L}{\partial \mathcal{G}^M}$
  $\frac{\partial L}{\partial \mathbf{W}} = einsum(\frac{\partial L}{\partial \mathcal{Y}}, \mathcal{X})$   ▷ $einsum$ here contracts only along batch dimension
                                   ▷ $FLOP_{\frac{\partial L}{\partial \mathbf{W}}} = O(BD_{in}D_{out})$
                                   ▷ $Memory_{\frac{\partial L}{\partial \mathbf{W}}} = O(D_{in}D_{out})$
  ▷ Results in the below for-cycle are computed only for the first batch during training and reused for others.
  for $k$ in $\{1, \dots, M\}$ do
    Compose $einsum_k$ expression for $\frac{\partial L}{\partial \mathcal{G}^k}$
    Optimize contraction schedule for composed $einsum_k$
  end for
  for $k$ in $\{1, \dots, M\}$ do
    $\frac{\partial L}{\partial \mathcal{G}^k} = einsum_k(\frac{\partial L}{\partial \mathbf{W}}, \mathcal{G}^1, \dots, \mathcal{G}^M)$
      ▷ $FLOP = O(D_{in}D_{out} \max_m(I_m, J_m)(\max_m R^m)^2)$
  end for
```

6 Experiments: end-to-end training of GPT-2 with custom layers

We conducted experiments with a GPT-2 generative model.We replaced the fully connected layers with the sequence of corresponding TTM containers and trained the resulting models from scratch on the task of language modelling (LM). In this section, we compare the performance of the original model with our model and a GPT-2 with a fully-connected layer, replaced with SVD structure (with the same parameter budget as our model).

The general intuition of TTM layers superiority w.r.t. SVD is as follows: TTM is proved to be full-rank (Khrulkov et al., 2019), since the truncated SVD is a low-rank method. Training the layers from scratch, we find a structure that defines weight matrices. The matrix $\mathcal{M} \in \mathbf{R}^{IJ}$ being restored from the TTM containers has rank $R_{TTM} = min(I, J)$. On the contrary, the matrix assembled from SVD factors has a truncated rank $R_{SVD} < min(I, J)$

We can suggest that for matrices with a certain dimension:

- TTM is seeking a proper weight in a more comprehensive space by utilizing a set of full-rank matrices, which are more effective than a set of matrices with truncated ranks;
- Higher rank matrix can store more information than a matrix with the same dimensions but a lower rank.

7 Inference time, memory requirements and energy assumption of training loops

Table 5: Memory, needed to provide one forward-backward operation on an NVIDIA A40 for a GPT-2 model with regular fully-connected, SVD-40 and TTM-32 layer. The batch size is equal to 1.

Memory / Layers	Regular	TTM-32	SVD-40
Model + input	487.4 MB	280.7 MB	285.6 MB
After Forward	3020.9 MB	2814.1 MB	2822.7 MB
After Backward	1293.0 MB	878.5 MB	881.3 MB
Peak usage	3376.9 MB	3170.4 MB	3178.14 MB

Table 6: Avaraged inference time and power consumption for a BERT model with regular fully-connected, SVD-40 and TTM-32 layer. The batch size equals to 1.

FC Layers	Regular	TTM-32	SVD-40
Inference time, ms	9,7	11.7	8.6
Power, kWh$*10^{-5}$	1.1	1.2	0.9

In this section we measure memory required for one training loop of the GPT-2 model with TTM

Table 4: In-domain perplexities for GPT-2 small model, pre-training from scratch.

Model	Training	Validation	Number of parameters	% of classic GPT-2 size	Perplexity
GPT-2 small	Wikitext-103 train	Wikitext-103 test	124 439 808	100	17.55
GPT-2 small TTM-16	Wikitext-103 train	Wikitext-103 test	68 085 504	54	21.33
GPT-2 small TTM-32	Wikitext-103 train	Wikitext-103 test	71 756 544	57	21.06
GPT-2 small SVD-40	Wikitext-103 train	Wikitext-103 test	71 503 104	57	22.19
GPT-2 small TTM-64	Wikitext-103 train	Wikitext-103 test	83 606 784	67	18.08
GPT-2 small SVD-170	Wikitext-103 train	Wikitext-103 test	83 483 904	67	20.98

and SVD-based linear layers with ranks 32 and 40, respectively (see Table 4). The results, which are shown in the Table 5, indicates reduction of required memory during the training loop in the case of SVD and TTM-based layers. Moreover, TTM-based layers require less memory than SVD.

We measure electricity consumption using Eco2AI library (Budennyy et al., 2023). As it is depicted in the Table 6, the SVD layers provide a reduction in inference time and energy consumption. On the contrary, TTM increases energy consumption and inference time. It may be due to the sequential multiplication of several cores within the forward function.

7.1 Hyperparameter selection

The proposed layer structure assumes two sets of hyperparameters - *TTM cores shapes* and *TTM ranks*. The matrix of sizes (I, J) is represented in cores $\mathcal{G} \in \mathbb{R}^{1,j_1,i_1,R_1}, \mathcal{G} \in \mathbb{R}^{R_1,j_2,i_2,R_2}, \dots, \mathcal{G} \in \mathbb{R}^{R_{M-1},j_M,i_M,1}$, where $I = \prod_{k=1}^{M} i_k, J = \prod_{k=1}^{M} j_k$, M - number of cores. Assuming the formula 2 for the compression rate in a TTM layer, we state that for the maximum compression rate, shapes should be as close to each other as possible. We choose $i_k * j_k$ in a way that they are equal to each other and approximately equal to $(I * J)^{1/M}$. Shapes selection is implemented with a custom algorithm which will be presented in the source code. In our case, GPT-2 small fully-connected layers [I, J] is [768, 3072]; $768 = 4 * 6 * 8 * 4$ and $3072 = 8 * 8 * 6 * 8$; $i_k * j_k$ are 8*4, 8*6, 6*8, 8*4. Figure 1 presents the scheme of TTM-based layers and the appropriate matrix of the FC layer. Purple and blue marks the dimensions of cores corresponding to the input and output sides of the initial weight matrix, respectively.

As for the choice of ranks, we choose them based on the desired compression of the entire model. For a small GPT, these are from 50% to 90%. For a medium GPT, the reduction is 40%.

7.2 In-domain language modelling task

To evaluate in-domain performance on the LM task, we provide training and evaluation on the train and test partition of the same dataset, respectively. We replace the fully connected layers of GPT-2-small with TTM of ranks 16, 32 and 64 as well as SVD with ranks 40 and 170. We train and validate the model with block size 512 in the Wikitext-103 dataset (Merity et al., 2016) for 40 epochs using the AdamW optimizer and the Cosine warm-up scheduler, increasing the training step from 0 to $2.5e^{-4}$. In this and subsequent experiments, we established the maximum learning rate point relative to the total number of training steps. Our goal was to ensure that the model reached its highest point and underwent approximately 1/10 of the entire learning process. Table 4 shows that the resulting perplexity is comparable to the original model. However, model compression has a negligible impact on quality within this domain. For example, a reduction in parameters of more than 30% only results in a half-percent decrease in perplexity, while a reduction of more than 40% leads to a drop in 3%. TTM-based model shows better results at the same compression ratio than SVD: so, TTM-64 obtains perplexity 18.08 while SVD-170 with the same size obtains 20.98. similarly, GPT-2 small models with 71 mln parameters - TTM-32 and SVD-40 - have performance of 21.06 and 22.19 respectively.

7.3 Out-domain language modelling task

In this setup, we perform validation on the Wikitext-103 test section while training the model on other datasets for the same language modelling task.

We train the GPT-TTM architecture on a sufficiently large dataset OpenWebText[3], which im-

[3]http://Skylion007.github.io/OpenWebTextCorpus

Table 7: Out-domain perplexities for GPT-2 Medium, GPT TTM-72 and SVD-50 models, pre-training from scratch.

Model	Training	Validation	Number of parameters	% of classic GPT-2 size	Perplexity
GPT-2 med	Webtext	Wikitext-103	354 823 168	100	20.56
GPT-2 TTM-72	Openwebtext	Wikitext-103	218 303 488	61	30.85
GPT-2 SVD-50	Openwebtext	Wikitext-103	220 920 832	62	55.46
Distill GPT-2	Openwebtext	Wikitext-103	81 912 576	23	51.45
OPT 350m	Openwebtext + BookCorpus + Pile (Gao et al., 2021)	Wikitext-103	331 196 416	93	24.75

itates the WebText dataset and is publicly available. We train the model for 10 epochs with a similar optimizer scheduler with a maximum learning rate $2.95 \exp -5$ and global batch size 340. When reaching the perplexity value of 50, we halved the batch size. We use an optimizer and scheduler as in the previous section, sequence length 1024. The optimal parameters were chosen based on the perplexity in the validation part of the Wikitext-103 dataset of a small GPT-2 model with classical fully connected layers. After obtaining the optimal parameters for the classical model, the learning settings were fixed. The training process continued for approximately 20 days on 4 GPUs 3090ti. To receive a GPT-based model with a compatible size, we train from scratch under the same conditions the GPT-2 medium with linear layer replaced with SVD-structure layers with rank 50. As shown in Table 7, the best perplexity among the compressed models is related to OPT (Zhang et al., 2022) with 350 million parameters. Herewith, OPT saves 7% of the full GPT-2, while TTM-72 saves 40%, and the perplexity decreases to 31. At the same time, an SVD-50 of a size similar to TTM-72 has perplexity 55, which is even worse than Distill GPT, the architecture with the smallest number of parameters.

7.4 Natural Languge Understanding - GLUE

We take a pre-trained GPT TTM-72 model from the previous section (without fine-tuning) and validate it on a General Language Understanding Evaluation (GLUE) benchmark. It is a collection of nine natural language tasks, including language acceptability, sentiment analysis, paraphrasing and natural language inference. The evaluation script is based on the original Transformer repository (Wolf et al., 2020). We add a top head compatible with the given task and run one training epoch. We choose just one epoch to avoid a situation where several models, all "large" concerning the number of tokens in the dataset but of different sizes relative to each other, converge to approximately the same loss during the entire training cycle (Kaplan et al., 2020). We repeated these experiments 5 times with different random seeds, Table 8 shows the averaged obtained results with a standard deviation of no more than 0.0008. The classical models and models with TTM layers show approximately equal results, periodically overtaking each other. GPT-2 TTM-72 has a performance decrease in Acceptability and several Question-Answering datasets (QNLI, MNLI). The result of SVD-50 is close to TTM-72.

7.5 Text Summarization

We also compare the behaviour of the proposed models in the text summarization task when tuning on a small amount of data. Based on the pipeline from (Khandelwal et al., 2019), we trained both models on 3000 objects from the CNN/Daily Mail datasets (Hermann et al., 2015; Nallapati et al., 2016). The obtained ROUGEs are not high (Table 9) but match the result from the paper cited and highlight the similar behaviour of the classical GPT-2 and TTM-72. SVD-50 shows a bit worse outcome, except for the ROUGE-L metric. The DistillGPT gives the expected lowest score; unexpectedly, but on the summarization task the OPT-350m loses to TTM-72 while remaining high only on the ROUGE-L.

8 Conclusion

We introduced an approach to obtain the compressed version of the GPT-2 model by representing its layer with its compressed analogue of a smaller number of parameters. We incorporate custom TTM layers as Fully Connected layers in a transformer-based GPT-2 architecture and evaluate it on Language modelling and Language Understanding tasks on English language. This modification results in a 40% reduction in model size, while maintaining performance on in-domain tasks without significant loss in quality. Furthermore, in

Table 8: Performance for GPT-2-based model on GLUE benchmark after one epoch fine-tining.

Model	STSB	CoLA	MNLI	MRCP	QNLI	QQP	RTE	SST2	WNLI	AVG
GPT-2 med	0.76	0.45	0.82	0.78	0.87	0.87	0.53	0.92	0.43	0.74
OPT 350m	0.73	0.32	0.81	0.78	0.88	0.86	0.56	0.92	0.39	0.69
GPT-2 TTM-72	0.77	0.23	0.79	0.80	0.61	0.86	0.47	0.82	0.56	0.66
GPT-2 SVD-50	0.73	0.08	0.78	0.68	0.84	0.84	0.57	0.89	0.43	0.64
DistilGPT	0.18	0.00	0.73	0.70	0.79	0.52	0.57	0.88	0.43	0.64

Table 9: Text summarization results.

Model	ROUGE-1	ROUGE-2	ROUGE-L
GPT-2 med	20.5	4.6	10.2
OPT-350m	15.9	3.7	11.5
GPT-2 TTM-72	20.1	4.1	9.9
GPT-2 SVD-50	18.1	2.3	11.3
DistilGPT	12.7	2.3	7.5

out-of-domain tasks, our proposed model outperforms similar architectures that use SVD instead of Fully-Connected layers and training from scratch under the same conditions.

We compare our method with other approaches to compressing the effective model size: distillation (DistillGPT) and training cut version of the architechture from scratch (OPT model). Our model significantly surpasses DistillGPT, but the OPT model provides the best score. This trend continues in downstream tasks such as Language Understanding and Text Summarization, where the quality of our resulting model is lower than the original but superior to baseline compressed models. These results show that in case of training a compressed or cut version of the given model from scratch, the dataset and training setup play a more significant role than the choice of method. DistillGPT2 has a week setup; OPT has a strong setup and well-prepared dataset; our model employs a regular dataset and mimics the training setup of the original model.

9 Limitations

The main limitation of this work is that a model with custom layers must be trained from scratch. It requires the operation of several industrial GPUs for several weeks and the necessary equipment, at least a load-bearing power supply. Such resources may be limited in the academy. The proposed model also was not validated on few-shot tasks, which defines a good generalization ability of the pre-trained model. It is important to recognize that training a large model from scratch is a skill that requires a certain level of expertise. As a result, the performance of two identical architectures can vary significantly depending on the specific training pipeline utilized.

10 Potential Risks and Ethical Statements

On the one hand, this work involves training large enough models from scratch, which can negatively affect the environment in terms of wasting resources. On the other hand, the proposed model has significantly fewer parameters and needs fewer floating point operations to learn and fine-tune it up to a given performance. On the other hand, in the future, this will help reduce the cost of training models to the desired scores.

11 Acknowledgements

The work was supported by the Center in the field of Artificial Intelligence in the direction of optimizing management decisions to reduce the carbon footprint on the basis of the Skolkovo Institute of Science and Technology under Contract No. 70-2021-00145/10841 dated 02.11.2021.

References

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Semen A. Budennyy, Vladimir D. Lazarev, Nikita Zakharenko, Alexey N. Korovin, Olga Plosskaya, Denis Dimitrov, Vladimir Arkhipkin, Ivan V. Oseledets, Ivan Barsola, Ilya Egorov, Aleksandra Kosterina, and Leonid Zhukov. 2023. Eco2ai: carbon emissions tracking of machine learning models as the first step towards sustainable ai. In *Doklady Mathematics*, pages 1–11. Springer.

Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. 2021. The pile: An 800gb dataset of diverse text for language modeling. *CoRR*, abs/2101.00027.

Karl Moritz Hermann, Tomas Kocisky, Edward Grefenstette, Lasse Espeholt, Will Kay, Mustafa Suleyman, and Phil Blunsom. 2015. Teaching machines to read and comprehend. In *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc.

Geoffrey E. Hinton, Oriol Vinyals, and Jeffrey Dean. 2015. Distilling the knowledge in a neural network. *CoRR*, abs/1503.02531.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *CoRR*, abs/2001.08361.

Urvashi Khandelwal, Kevin Clark, Dan Jurafsky, and Lukasz Kaiser. 2019. Sample efficient text summarization using a single pre-trained transformer. *CoRR*, abs/1905.08836.

Valentin Khrulkov, Oleksii Hrinchuk, Leyla Mirvakhabova, and Ivan V. Oseledets. 2019. Tensorized embedding layers for efficient model compression. *CoRR*, abs/1901.10787.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer sentinel mixture models. *CoRR*, abs/1609.07843.

Ramesh Nallapati, Bowen Zhou, Cicero dos Santos, Çağlar Gulçehre, and Bing Xiang. 2016. Abstractive text summarization using sequence-to-sequence RNNs and beyond. In *Proceedings of the 20th SIGNLL Conference on Computational Natural Language Learning*, pages 280–290, Berlin, Germany. Association for Computational Linguistics.

Alexander Novikov, Dmitrii Podoprikhin, Anton Osokin, and Dmitry P Vetrov. 2015. Tensorizing neural networks. In *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc.

Ivan V. Oseledets. 2010. Approximation of $2^d \times 2^d$ matrices using tensor decomposition. *SIAM Journal on Matrix Analysis and Applications*, 31(4):2130–2145.

Ivan V. Oseledets. 2011. Tensor-train decomposition. *SIAM J. Sci. Comput.*, 33(5):2295–2317.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. Distilbert, a distilled version of BERT: smaller, faster, cheaper and lighter. *CoRR*, abs/1910.01108.

Daniel G. A. Smith and Johnnie Gray. 2018. Opt_einsum - A python package for optimizing contraction order for einsum-like expressions. *J. Open Source Softw.*, 3(26):753.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online. Association for Computational Linguistics.

Chunxing Yin, Bilge Acun, Xing Liu, and Carole-Jean Wu. 2021. Tt-rec: Tensor train compression for deep learning recommendation models. *CoRR*, abs/2101.11714.

Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona T. Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. 2022. OPT: open pre-trained transformer language models. *CoRR*, abs/2205.01068.