Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Text Compression for Efficient Language Generation

Master's Thesis

David Gu

david.gu@inf.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Dr. Peter Belcák
Prof. Dr. Roger Wattenhofer

July 18, 2024

Acknowledgements

I would like to express my gratitude to Dr. Peter Belcák for introducing me to this fascinating area of research and for providing high-level guidance throughout the project with his expertise.

Moreover, I would like to thank my colleague Philippe for valuable discussions on this topic.

I am grateful to my friends Nathan and Giorgio for their invaluable discussions, proof-reading os this thesis, and non-stop support throughout the duration of this thesis.

Lastly, I thank the DISCO group led by Prof. Roger Wattenhofer for giving me the opportunity to do this project in their lab and for providing access to the TIK Arton cluster.

Abstract

We challenge the prevailing assumption that large language models must operate on sub-word tokens by introducing Thoughtformer (THF), a hierarchical transformer-based language model that efficiently operates on compressed sentence representations. Our main focus is on training a generative Thoughtformer model (GPTHF). We show that GPTHF can be trained with no architectural changes to GPT by instead employing sparse, dynamically computed triangular attention masks instead of full ones. Our experiments show that GPTHF models achieve an up to an order of magnitude improvement in FLOPs efficiency and a threefold increase in runtime efficiency compared to similarly sized GPT models in the low-size regime. This is achieved through an innovative generation method that caches and reuses sentence embeddings, allowing significant portions of the input to bypass large parts of the network. For completeness, we discuss also how THF can be applied also to bidirectional models.

Contents

Chapter 1

Introduction

In recent years, the development of large language models (LLMs) has garnered substantial interest from academic and business-related communities due to their far-reaching implications. The architecture underpinning the rise of LLMs is the Transformer model [Vaswani et al., 2017]. The dominant paradigm that has prevailed over the years has been to enhance these models by scaling. Early models like BERT [Devlin et al., 2018] and GPT-1 [Radford et al., 2018] had 110 million and 117 million parameters, respectively. This scaling trend continued with GPT-3 [Brown et al., 2020] with 175 billion parameters, PaLM with 540 billion [Chowdhery et al., 2023], and the Switch Transformer reaching 1.6 trillion parameters [Fedus et al., 2022], all within a span of just four years.

While these models have become remarkably capable in a variety of NLP tasks [Naveed et al., 2023], their massive scales come with substantial costs in hardware, energy and time to both train and deploy these models [Strubell et al., 2019, Patterson et al., 2021]. The high computational demands require the exploration of more efficient methodologies. Recent studies have focused on strategies such as pruning [Augasta and Kathirvalavakumar, 2013, Molchanov et al., 2016], quantization [Hubara et al., 2018], and knowledge distillation [Gou et al., 2021, Kim and Rush, 2016] to decrease model size without compromising performance. Additionally, advancement in architecture such as the mixture of experts models [Shazeer et al., 2017, Fedus et al., 2022] have been introduced to reduce effective model size during inference while maintaining overall model capacity. These advances show a growing trend towards making powerful language models more accessible and sustainable.

However, an area with great potential for improvement might have been overlooked: LLMs nowadays still operate on sub-word tokens. Each of these is represented by an embedding with a size of several kilo-bytes (3KB for BERT up to 37KB for PaLM). In contrast, an average English word can be represented by roughly 5 ASCII bytes. A question naturally arises of whether it is feasible for transformers to operate on more condensed text representations than sub-word token embeddings. Recent work such as the Funnel-Transformer [Dai et al., 2020] and hierarchical text transformers [Nawrot et al., 2021] have hinted that this might be possible: By compressing (and subsequently expanding) sequences of hidden states by small constant factors throughout the transformer, a significant saving in computational demands can be gained coming at little performance cost.

In this work, we go a step further. Instead of compressing fixed-size groups of sub-word tokens, we go beyond sub-word tokens and instead compress entire sentences into fixed-size embeddings. Specifically, we investigate a) whether transformer models can compress information-rich sentence representations such that operating on these representations alone can generate high-quality text, and b) whether a resulting performance-efficiency trade-off from applying this approach is worthwhile.

To answer these questions, we introduce Thoughtformer (THF), a **hierarchical transformer model** that uses transformer layers to individually compress sentences into fixed-size embeddings (later referred to as *sentence embeddings*). In a second step, these sentence embeddings are collected passed through a second set of transformer layers to pass the information between sentences. While we introduce a bidirectional model for completeness, our primary focus is on training a generative THF model (GPTHF).

We evaluate the perplexities of GPTHF and compare it with established baselines. Our results show that GPTHF follows scaling laws in the low-parameter regime, a promising sign that transformers can effectively perform sentence-level attention. Furthermore, we observe significant improvements in efficiency during generation, as evidenced by our experiments in both measuring floating-point operations (FLOPs) and runtime.

While computational efficiency is the immediate and measurable advantage of our approach, conceptually there is more to be gained from the idea of thinking hierarchically at different levels of abstraction. It mirrors human language understanding, where we comprehend and respond to information given by our peers at high levels of abstraction before articulating it into concrete words. If successful, our work could imply that such hierarchical thinking and generation is possible in AI. Though our focus is on language, this concept could be applicable to other domains such as vision, or further levels of abstraction (e.g., paragraphs, documents instead of sentences).

Our main contributions can be summarized as follows:

1. We propose a generative language model capable of generating text using only one fixed-size embedding per sentence. We demonstrate that this model can be trained with minimal modifications to the existing GPT training procedures.

2. We introduce a new generation method that caches and reuses embeddings of previous sentences. This approach achieves efficiency and speedups that

grow linearly with context size, resulting in up to an order of magnitude improvement in FLOPs and a threefold increase in runtime efficiency given sufficient context.

To the best of our knowledge, these contributions are novel and have not been explored previously.

Chapter 2

Related Work

2.1 Hierarchical Transformers

Traditionally, transformer models operate on a fixed size embedding across all layers [Vaswani et al., 2017, Devlin et al., 2018, Radford et al., 2018]. A line of research started exploring "hierarchical transformers", a transformer model that operates on variable-size embeddings representing meaning at various levels of abstraction within different layers of the network.

Early examples include the Hierarchical Attention Network for Document Classification [Yang et al., 2016]. An other example is "Sentence Bottleneck Autoencoders developed from Transformer Language Models" [Montero et al., 2021], where they propose AutoBot. This model takes the final embeddings from a pre-trained RoBERTa framework [Liu et al., 2019] and learns an attention pooling mechanism and a decoder. Fine-tuning on downstream tasks such as sentence similarity shows that their sentence embeddings outperform naive mean or first-token pooling methods. However, their approach focuses on downstream Natural Language Understanding (NLU) tasks and uses a pre-trained model as a backbone, which is frozen during fine-tuning due to computational constraints. In contrast, our work proposes a generative language model, where all parts of our model are trained jointly.

Dai et al. [2020] introduced the Funnel Transformer, a model that incrementally compresses tokens by pooling (mean pooling with filter size 2 and stride 2, effectively halving the context size at each pooling step) across multiple stages, for instance through a "8-8-8" layer configuration. Inter-layer skip connections are used in order for later layers to still be able to have access to information present at earlier layers. When re-investing the FLOPs that were saved due to the shorter context size, the Funnel Transformer achieves superior performance metrics compared to larger models such as ROBERTA-large or XLNet-Large with comparable computational resources.

Building on these foundations, Nawrot et al. [2021] extended these principles to autoregressive transformers with their "Hourglass" model. They compress a fixed number of tokens and then decompress them. To address the potential for information leakage, they shift the input by k tokens instead of the usual one-token shift, enhancing language modeling efficiency as demonstrated through improved perplexity scores on a Wikipedia dataset.

The landscape of hierarchical transformers also encompasses models like Sentence-BERT [Reimers and Gurevych, 2019] and Sentence-GPT [Muennighoff, 2022], which are tailored to generate sentence embeddings suitable to various downstream tasks. However, these models typically focus on sentence-level embeddings and do not fully explore the broader potential of text compression within transformer architectures.

Our work differs from all of the above in several ways. First, instead of compressing a fixed-size group of tokens, we compress a sentence – which represents a unit of higher semantic value in language – into one embedding. Moreover, our research does not prioritize the sentence embeddings themselves. Instead, it leverages compression techniques to enhance overall NLP efficiency. Lastly, our work proposes a generative model which can use compression to generate text 3 times faster than similar sized GPT models.

2.2 Sentence Embeddings

Several work leverage sentences embeddings, vector representations of sentences to improve performance on downstream task such as semantic sentence similarity scoring and semantic search. A first example of this is Sentence-BERT by Reimers and Gurevych [2019], which introduce a siamese networks and a triplet loss to explicitly encode semantic similarity into distances within the embedding space.

Wang et al. [2021] proposed TSDAE, a BERT-sized transformer autoencoder trained using the masked language modeling task, whose encoder can be fine-tuned using a contrastive loss to achieve strong downstream performance.

However, none of these works explores the possibility of contextualizing sentence embeddings using additional attention layers. Instead, they focus solely on applying the sentence embeddings to the specific downstream tasks. As such, their relevance to our work is rather limited.

CHAPTER 3

Method

We outline the general ideas and pipeline underlying the Thoughtformer (THF) model, focusing on preprocessing and optimization methods. In total, we present two variants: a bidirectional, BERT-like model tailored for natural language understanding (NLU) tasks, which we include for completeness and describe in detail in Chapter 4, and a GPT-like model designed for natural language generation (NLG) tasks, described further in Chapter 5, which constitutes the core part of this work.

3.1 Data Preprocessing

Given a corpus D consisting of documents $(X_1, X_2, \cdots, X_{|D|})$, each document is first split into sentences $s_1, \cdots, s_m$ using the NLTK Punkt sentence tokenizer [Bird et al., 2023]. The sentences are tokenized using a pre-trained tokenizer (DistilBert [Sanh et al., 2019] for bidirectional, GPT2 tokenizer [Radford et al., 2019] for generative). Documents are then divided into training samples such that the context size of the model and sentence boundaries are respected. Sentences that cannot be finished in the current context are carried over to the next sample. The samples then are considered independent, i.e. any information between them (e.g. if they belonged to the same document) is lost.

3.2 Architecture

The THF model has a modular architecture comprising a word-level encoder (`wlt_encoder`), a sentence-level transformer (`slt_body`), and an optional world-level decoder (`wlt_decoder`) in case the training objective requires the output to be a full sequence of hidden states. All modules consist of consecutive transformer layers. Implicitly, the encoder's objective is to compress the content of a sentence into a single embedding (also called *sentence embedding*) with as little information loss as possible. The idea is that if the sentence embeddings retain sufficient information for the body to effectively process them via attention mechanisms,

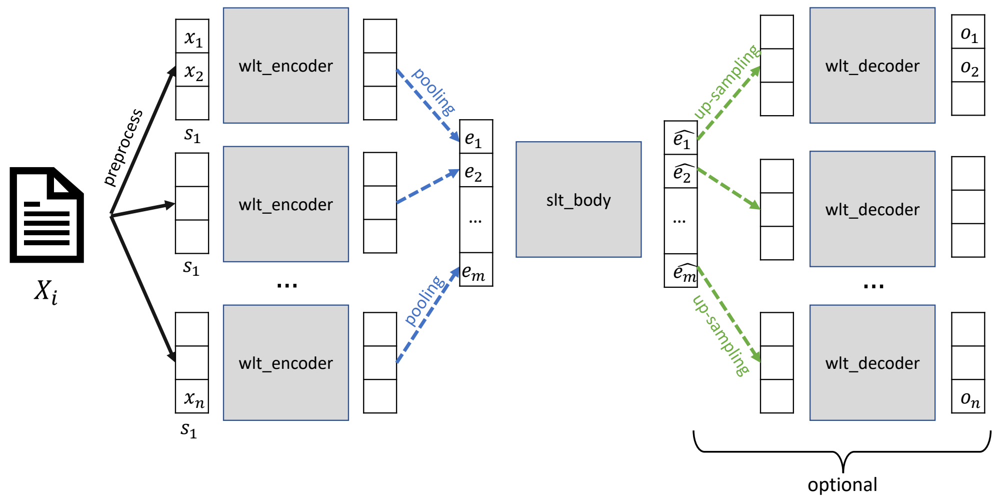


Figure 3.1: High-level overview of the THF Architecture.

a reasonable performance can be maintained. Due to the body's considerably shorter context size, this configuration will have a significant speedup in processing times compared to traditional equivalent models like BERT or GPT.

During the forward pass of a document X_i, the tokenized text $x_1, \cdots, x_n$ is first pre-processed into sentences $s_1, \cdots, s_m$ according to Section 3.1. After applying positional embeddings, the tokens of each sentence are first processed by the `wlt_encoder` to generate contextualized sub-word embeddings. A pooling method produces for each sentence $s_i, i \in [m]$, an embedding e_i.

The sentence embeddings $e_1, \cdots, e_m$ are fed directly, instead of using cross-attention, into the `slt_body`, where they are further contextualized by the surrounding sentence embeddings to form $\hat{e_1}, \cdots, \hat{e_m}$.

To recover a complete sequence of hidden states, the `wlt_decoder` restores the original length of the input sequence and produces $o_1, \cdots, o_n$, which can then be processed by a language modeling head.

More details on the training and inference of the model, including optimizations, are provided in Chapters 4 and 5.

3.3 Optimization 1: Efficient Sentence Packing

Challenge. In our initial model formulation, a naive implementation would handle input with the shape

$(B = \text{batch size},\ S = \text{number of sentences},\ T = \text{number of tokens per sentence})$, instead of the usual (B, T). This results from splitting documents into sentences and then into tokens in order to compress sentences individually **without allowing cross-sentence token attention**.

This approach, however, proves to be prohibitively expensive in terms of memory. The added dimension S requires dynamic padding to the maximum number of sentences (besides the usual padding to the maximum sentence length), leading to excessively large tensor dimensions. To see this, consider a batch containing two samples: One divided into 20 short sentences and one consisting of a single sentence spanning the whole context, 128 tokens (512 for the generative model). The resulting input is of shape $(B, 20, 128, 768)$, of which roughly 95% is occupied by padding tokens.

Key Idea. The key insight is that the extra dimension S is not really necessary. Instead, we can prevent cross-sentence token attention within the original input structure by employing a localized attention mechanism. The key is to track "sentence index" vector at tokenization time. This vector tracks which sentence each token belongs to and is crucial in applying the necessary adjustments in order to ensure that tokens from different sentences do not influence each other. All ideas presented in this section are also outlined in [Krell et al., 2021].

Adjust Attention Masking. The key to prevent cross-sentence token attention within the original input structure is to employ a localized attention mechanism. The mask is configured such that only tokens within the same sentence (as indicated by the sentence index vector) are allowed to attend to each other. Given the sentence index vector, we can modify the attention mask into a block diagonal matrix, where each block corresponds to a single sequence. For simplicity, let's assume the sentence index vector for a sample text "Hello there. How are you? I'm fine." to be $[0, 0, 1, 1, 1, 2, 2]$, the corresponding attention mask would be a matrix where interactions are allowed within the blocks of $[0, 0]$, $[1, 1, 1]$, and $[2, 2]$ (see Figure 3.2 for an illustration). This creates a slightly unusual situation, where the attention matrix is dependent on the content and hence dynamically computed for each input.

At this point, we would like to note that the localized attention matrix already offers a theoretical potential for computational speed-ups over a full attention matrix due to its increased sparsity. Consider a context of size T divided into k sentences of approximately equal length, the attention matrix would then comprise k blocks of size T/k each. This results in only T^2/k non-zero entries, compared to T^2 in a fully dense matrix. Moreover, these T^2/k nonzero entries are organized in a structured, block-wise way. A clever implementation could potentially bypass some operations on the zeroed entries, resulting in an efficiency gain. However, current practical implementations may not be able to take full advantage of this sparsity. Therefore, we acknowledge this only as an observation and will not further go into this point.

$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

(a) Block matrix

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

(b) Block lower triangular matrix

Figure 3.2: Visualization of adjusted attention masks for a text with sentence index vector $[0, 0, 1, 1, 1, 2, 2]$. (a) Standard block matrix allowing attention within sentences in a bidirectional model. (b) Block lower triangular matrix allowing attention to previous tokens within sentences in an autoregressive model.

Adjust Positional Embeddings (Optional). In order to align our model mathematically with the original THF formulation, it would be necessary to reset the positional embeddings at the beginning of each sentence. For instance, in the BERT model, each token at index i within a sequence is assigned a corresponding sinusoidal positional embedding that is added to the input. However, when packing sentences, the position index must be reset at the start of each new sentence. Following our earlier example with the sentence index vector $[0, 0, 1, 1, 1, 2, 2]$, the required positional embedding indices would be $[0, 1, 0, 1, 2, 0, 1]$ as opposed to the conventional sequence $[0, 1, 2, 3, 4, 5, 6]$. However, we observe no performance degradation without this adjustment, and dynamically adjusting positional embeddings slightly slows down training. Therefore, we decide against this adjustment in our final implementation.

3.4 Optimization 2: Training in Low-Compute Setting

In our pursuit to train language models despite being constrained on a limited computational budget, we adopt several modifications outlined in "Cramming: Training a Language Model on a GPU in One Day" [Geiping and Goldstein, 2023]. Here, we outline some key changes which allow us to efficiently train the THF architecture within a restricted timeframe and computational budget. More detailed optimizations specific to the bidirectional or generative models, as well as adjustments to their training procedures, are outlined in Chapters 4 and 5, respectively.

No Biases in Hidden Layers. For both models, we disable all QKV biases in the transformer attention layers, as well as biases in the linear layers. This modification reduces a slight computational overhead without significantly impacting model size, thereby increasing throughput at little performance cost.

Pre-normalization and Final Normalization Layer Previous studies by Popel and Bojar [2018] have shown that pre-normalization, which involves normalizing the input of each transformer sub-layer rather than the output (post-normalization), benefits the stability of the training process. Following this advice and in alignment with Geiping and Goldstein [2023], we use pre-normalization and add a final normalization layer at the end of the final transformer layer before the language modeling head.

No Dropout During Pre-training. We have decided to eliminate dropout during pre-training. Given that our models undergo only a single pass or less over the pre-training corpus, overfitting is not possible [Geiping and Goldstein, 2023], making dropout less necessary.

CHAPTER 4

Bidirectional Model

4.1 Data

We use the pre-training corpus proposed by the BERT model [Devlin et al., 2018], composed of English Wikipedia as of the 1st of March 2022 and BookCorpusOpen [1]. Our main motivation of choosing these sources is to facilitate direct comparisons to BERT. The documents are pre-processed according to Section 3.1, using the uncased DistilBertTokenizer [Sanh et al., 2019] as the pre-trained tokenizer.

4.2 Architecture

Name	Params	d	n_{heads}	$l_{encoder}$	l_{body}	$l_{decoder}$	lr
THF-6-4-2	94M	768	12	6	4	2	6e-4
THF-8-4-2	106M	768	12	8	4	2	6e-4
THF-8-6-4	192M	1024	16	8	6	4	5e-4
THF-10-4-2	174M	1024	16	10	4	2	5e-4

Table 4.1: Model sizes together architectural and optimization hyperparameters for bidirectional models.

The text $x_1, \cdots, x_n$ (where n is typically 128, our context size for this model) is first processed. We do not physically split into sentences as described in Section 3.1, but instead keep track of a sentence index vector (see Section 3.3). After applying positional embeddings, each sentence's tokens are first processed by the `wlt_encoder` to generate contextualized sub-word embeddings. The `wlt_encoder` uses the block attention matrix as outlined in Section 3.3, computed with the help of the sentence index vector.

[1]The original authors used BookCorpus, which is not available anymore due to licensing issues. BookCorpusOpen serves as an alternative, being compiled from books that are publicly accessible.

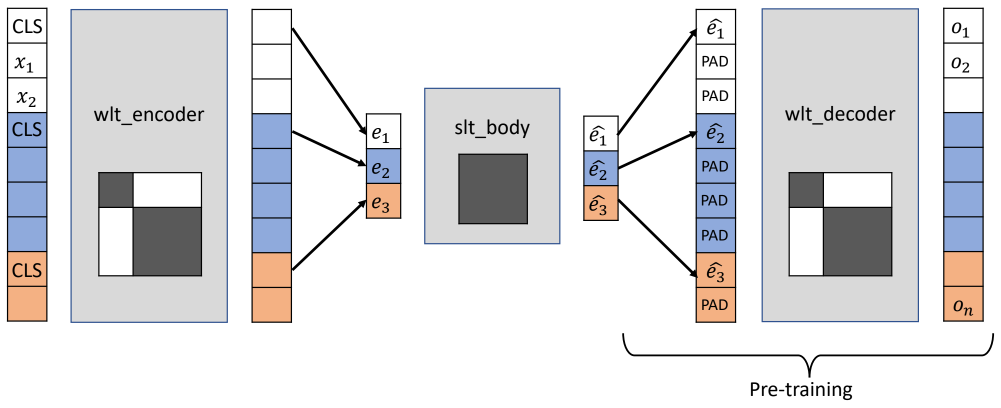


Figure 4.1: Overview of the Bidirectional THF Architecture without the classification head. The colors in the tokens indicate which sentence they belong to. The boxes in the modules indicate the attention masks being used in training and inference: Block attention masks as outlined in Section 3.3 for `wlt_encoder` and `wlt_decoder`, and a full attention matrix for the `slt_body`.

The pooling method of our choice to produce sentence embeddings e_i is the output token of the encoder corresponding to the `CLS` token of the sentence s_i. This is standard practice in BERT-like models, and no improvement was found when using average pooling. The fetching operation can be implemented efficiently by leveraging the sentence index vector. The sentence embeddings $e_1, \cdots, e_m$ are processed by the `slt_body`, where they are further contextualized at the sentence level to form $\hat{e_1}, \cdots, \hat{e_m}$. A fully dense attention matrix is used in this stage.

At inference time, $\hat{e_1}, \cdots, \hat{e_m}$ are averaged in order to obtain one final embedding, which is run through a final layer in order to produce the classification logits. No difference in performance was found when employing a different pooling method.

During pre-training, to recover a full sequence of hidden states, the `wlt_decoder` is used. Each $\hat{e_i}$ is concatenated with padding tokens until the original length of the sentence is restored. This input is then processed by the `wlt_decoder`, producing $o_1, \cdots, o_n$, which can be processed by a language modeling head.

For an illustration of the architecture, see Figure 4.1. A summary of the model sizes and other hyperparameters are provided in Table 4.1. As outlined in Section 3.4, we align our model with CrammedBERT [Geiping and Goldstein, 2023], both in order to train THF fast and to have a model that is easy to compare to it. In practice, we make the following design choices, many of which are taken directly from their work.

Embeddings. We implement scaled sinusoidal positional embeddings as described by Geiping and Goldstein [2023]. Moreover, we add a final layer normalization at the end of the embedding block.

Activation Function. We use GELU [Hendrycks and Gimpel, 2016] activation, where the block is re-ordered into a gated linear unit (GELUglu) without increasing the number of the parameters in the FF block to compensate for the halving of the hidden dimensionality due to gating.

Layer Structure. As outlined in Section 3.4, we use pre-normalization. Apart from that, the layer is equivalent to a PyTorch transformer encoder block.

In the subsequent section, we briefly describe the pre-training details for the bidirectional model. An evaluation of the bidirectional model can be found in Appendix A.

4.3 Pre-training.

Objective. In the pre-training phase, we employ the Masked Language Modeling (MLM) approach. Consistent with the standard BERT pre-training procedure, 15% of the tokens in the input sequence are masked, out of which 80% are replaced by a mask token, 10% are replaced by random tokens and 10% remain unchanged. Following insights from RoBERTa [Liu et al., 2019] and other related studies, we omit the Next Sentence Prediction task due to its limited impact on performance.

Optimizer. We use Adam [Kingma and Ba, 2014] as the optimizer of choice, with weight decay of 0.01 as described by Loshchilov and Hutter [2018], and similar to CrammedBERT use $\beta_1 = 0.9$, $\beta_2 = 0.98$ and $\epsilon = 10^{-12}$. Moreover, we include gradient clipping at a clip value of 0.5.

Learning Rate and Scheduler. Different to CrammedBERT's one-cycle learning rate scheduler, we found a slight improvement for our models by employing linear decay with 10000 warm-up steps. The peak learning rates, together with other hyperparameters, can be found in Table 4.1.

Batch Size Scheduler. The batch sizes are accumulated using gradient accumulation according to the batch size scheduler from CrammedBERT, starting at 256 and and linearly ramping up to 8192, reaching this peak at 60% of the training duration. This provides an early boost while performing more modest updates in the later stages of training.

Chapter 5

Generative Model

5.1 Data

Our training corpus incorporates OpenWebText, Wikipedia and ArXiv. OpenWebText, an open alternative to the Webtext dataset used by OpenAI for training GPT2 [Radford et al., 2019], forms the backbone of our corpus due to its larger size and diverse internet content. This allows our model to handle a broad spectrum of topics and writing styles. Wikipedia, known for its comprehensive coverage across various areas of general knowledge, ensures that the model develops a well-rounded understanding of general facts. Finally, ArXiv augments our corpus by adding scientific and highly technical texts, enabling our model to understand and generate high-quality content in these domains as well.

Moreover, we modify the tokenization process by adding a designated "end-of-sentence" token after each sentence. As we will see, this "end-of-sentence" token plays a crucial role in the design of a fast generation method, a cornerstone of this work.

5.2 Architecture

At this point, the reader has likely obtained a general understanding of the inner workings and the interplay between the `wlt_encoder` and the `slt_body`. To avoid repetition, we will directly describe the inference process of the Generative THF (GPTHF), highlighting the key changes from the bidirectional model.

Inference. The model predicts the token j in sentence i by utilizing all compressed sentence embeddings from sentences 1 to $i-1$ and the compressed prefix of tokens from 0 to $j-1$ within sentence i. The compressed embeddings are generated by the word-level encoder, which compresses each sentence independently without cross-sentence contamination by using a block-attention mask. Instead of fetching the `CLS` embedding, we fetch the last embedding, consistent with the

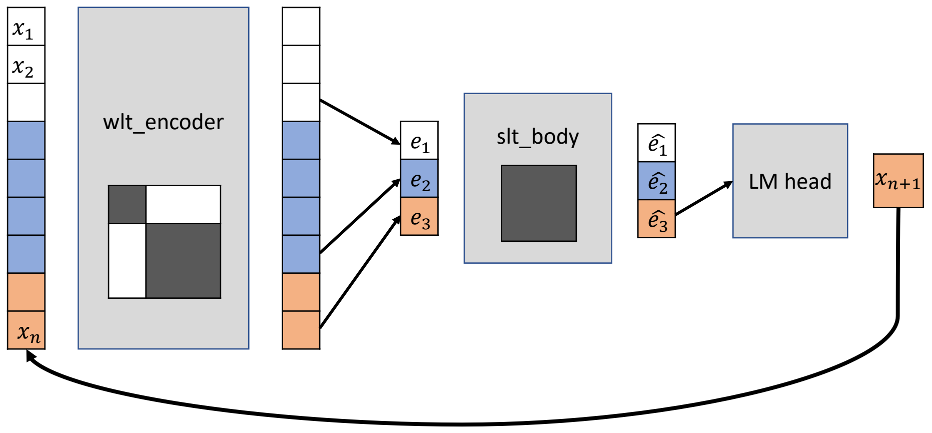


Figure 5.1: Overview of the Generative THF (GPTHF) Architecture during inference.

Name	Params	d	n_{heads}	$l_{encoder}$	l_{body}	lr
GPTHF-8-4	151M	768	12	8	4	6e-4
GPTHF-16-8	454M	1024	16	16	8	4e-4

Table 5.1: Model sizes together architectural and optimization hyperparameters for GPTHF models.

training process, as we will see in the subsequent section. These embeddings are then contextualized by the `slt_body`.

Intuitively, the inference behavior is almost identical to the inference behavior of the bidirectional model (treating the token as a classification label), with the only difference being the pooling method. First, the last embedding instead of the `CLS` embedding of each sentence is fetched as the sentence embedding. Second, the last contextualized sentence embedding instead of the mean of all sentence embedding is chosen as the input of the language modeling head.

For an illustration of the GPTHF at inference time, refer to Figure 5.1. A summary of the model sizes and other hyperparameters are provided in Table 5.1. Through empirical experimentation, we found that a relatively large encoder is beneficial, likely because it allows the model to perform extensive operations on sequences of normal length. Moreover, the architectural hyperparameters proposed by CrammedBERT were not as effective, which we believe are likely optimized for bidirectional models. Instead, we decide on these modifications, taken directly from Llama-1 [Touvron et al., 2023].

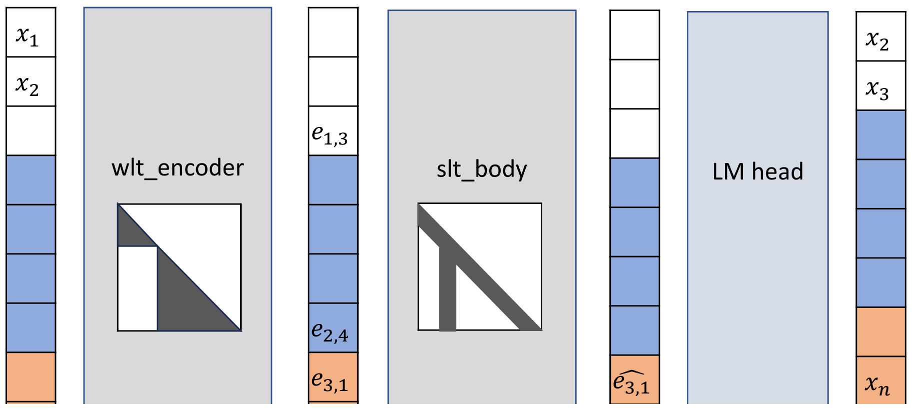


Figure 5.2: High-level overview of the parallel training of the generative THF on the next token prediction objective. The boxes in the models indicate the type of attention masks used. The attention masks are explained in Figure 5.3.
For instance, the embedding $e_{3,1}$ is allowed to attend to $e_{1,3}$ and $e_{2,4}$ while being processed by the `slt_body`, as ensured by the attention mask

Embeddings. Similar to Llama-1, we replace an absolute positional embedding layer with rotary positional embeddings (RoPE), introduced by Su et al. [2024], at each attention layer of the network.

Activation Function. We use the SwiGLU activation function, introduced by Shazeer [2020]. Following the recommendation of Llama-1, we use a dimension of 2/3 4d instead of 4d as in PaLM.

Layer structure We continue to use pre-normalization, but instead of LayerNorm, which was used in CrammedBERT, we use RMSNorm [Zhang and Sennrich, 2019].

5.3 Pre-training

Objective. For pre-training, we employ the next token prediction objective, common in autoregressive language models. The tokens are shifted by one position to the right to create the target sequence.

Procedure. To prepare our model for the task of predicting tokens with access only to embeddings of preceding sentences and the compressed prefix of the current sentence, while at the same time enabling efficient parallel training, we need a clever strategy. It turns out that we can achieve this with minimal change

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

(a) Encoder attention matrix

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$

(b) Body attention matrix

Figure 5.3: Attention masks during pre-training for an input with the sentence index vector [0,0,1,1,1,2,2]: The left matrix is the "block triangular mask" introduced in Section 3.3. Every token can attend to every preceding token of the sequence, but not to token of previous sequences. After going through the encoder, every token represents the compressed prefix of its sequence up to itself, and is only allowed to attend to itself and compressions of previous sequences (right).

to the usual pre-training procedure by using specialized attention masks as depicted in Figure 5.3. Following the application of the block-triangular mask in the encoder, the body attention mask ensures that each token can only attend to its position, which contains the compressed token sequence up to that point, and the last embedding of each preceding sentence. The target is the next token in the sequence. This setup trains the model for the inference scenario, where it can only access the compressed embeddings of each sentence.

Optimizer. We use Adam with weight decay of 0.01 from Section 4.3 as the optimizer. Moreover, we use $\beta_1 = 0.9$, $\beta_2 = 0.98$ and $\epsilon = 10^{-8}$. Additionally, we maintain gradient clipping at a clip value of 0.5.

Learning Rate and Scheduler. We again use linear decay with 10000 warmup steps as our learning rate scheduler. The peak learning rates are provided in Table 5.1.

Batch Size Scheduler. We keep the batch size scheduler from Section 4.3. To account for the increased context size, the batch size starts at 64 and linearly ramps up to 4096, reaching this peak at 60% of the training duration.

Remark. It may seem surprising to the reader that the essential difference between training a conventional GPT and training a GPTHF, which at first sight

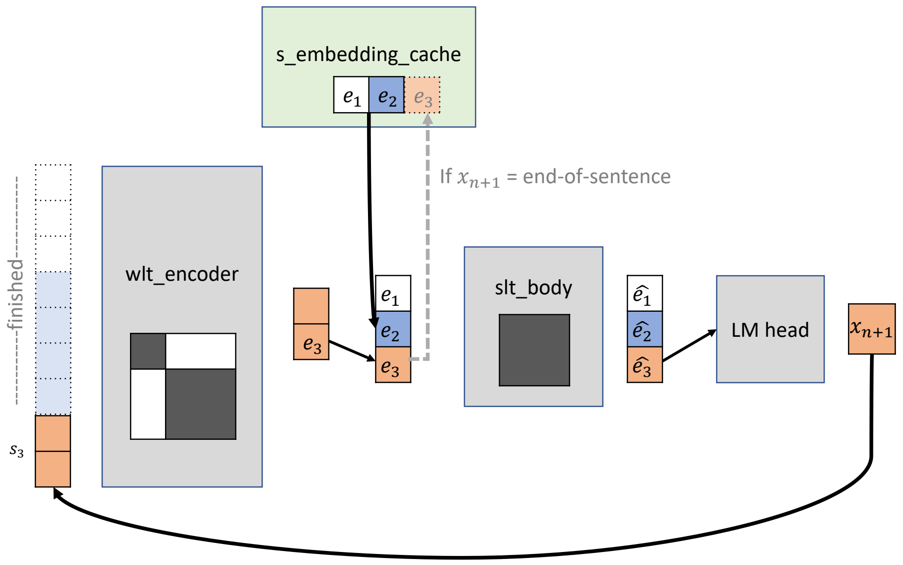


Figure 5.4: Illustration of an iteration of the Fast Generation Algorithm. After having finished two sentences s_1 and s_2 in the context, any subsequent token mathematically cannot influence e_1, e_2. The Fast Generation Algorithm caches them and feeds them directly to the `slt_body`, together with e_3, a compressed representation of the tokens in s_3, the current sentence.

appear very different, lies in just substituting full triangular attention matrices with sparser ones dynamically computed for each input, with no architectural changes. Theoretically, this results in a strictly weaker generative power when comparing to a conventional GPT. We will experimentally verify the extent of the performance drop in the following sections. For now, let's explore how our method allows a faster generation process, with speedup scaling proportionally to the number of sentences in the context.

5.4 Fast Generation

The key insight that enables our fast generation algorithm to be mathematically equivalent to regular generation is the design of our block-wise attention matrix. During the generation loop, when generating a token in sentence j, the block-attention matrix ensures that this process only affects the tokens in sentence j and never any token in previous sentences. Since the feedforward layers operate element-wise, there is no operation within the transformer layer that alters the compressed embeddings $e_1, e_2, \cdots, e_{j-1}$. Thus, these embeddings remain fixed. The core idea is to cache these embeddings, allowing the encoder to process only the current sentence j to compute e_j and then feed the body with the concatenation of the cached embeddings $e_1, e_2, \cdots, e_{j-1}$ and the newly computed e_j. Below, we outline the algorithm in pseudo-code. An illustration can be found in Figure 5.4. Intuitively, since the fast generation algorithm processes only the current sentence after the first iteration, and these sentences are (statistically) bounded by a constant length, we can expect to observe a speedup that increases asymptotically with the number of sentences. We will conduct experiments later to verify this intuition.

Algorithm 1 Fast generation

```
Require: input_ids, attention_mask, sentence_ids, max_new_tokens, temperature, top_k

  emb_cache ← [ ]
  for $i$ ← 1 to max_new_tokens do
      word_emb ← self.encoder(input_ids, attention_mask, sentence_ids)
      sent_emb ← fetch_sent_emb(word_emb, sentence_ids)
      body_input ← concat(emb_cache, sent_emb)
      final_emb ← self.body(body_input, attention_mask, sentence_ids)
      next_token ← get_next_token(final_emb, temperature, top_k)
      emb_cache ← sent_emb of finished sentences (marked by end-of-sentence token)
      Truncate input_ids, attention_mask, sentence_ids by finished sentences
  end for
```

Chapter 6

Experiments

In this chapter, we present the details of our experimental setup for our generative model. We include model configurations, training procedures, and the results of our evaluations.

6.1 Experiment Setup

We start by reporting pre-training perplexity, a measure of how well our predicted probability distribution matches the distribution of the training data. We do not include evaluations on downstream generation tasks, such as zero-shot or few-shot tasks (e.g., Question-Answering (BoolQ) or Reading Comprehension (RACE)), as our model and training were likely too small for these evaluations to be meaningful. Preliminary trials indicated that a certain threshold in model scale is required to perform significantly better than random guessing. Finally, we evaluate the inference speed of our models, focusing on FLOPs and runtime.

Baselines. To ensure a fair comparison and avoid disadvantaging our baselines in the low-compute setting, we trained a 12-layer baseline named "Baseline-12" and a 24-layer "Baseline-24" with the exact same architecture and same number of parameters as their GPTHF counterparts. The only difference was that they were trained using full triangular masks for both the encoder and body, as opposed to the masks in Figure 5.3). As remarked in Section 5.3, the baselines can be regarded as equivalent to conventional GPTs.

In order to compare with more established architectures, we also include the vanilla Huggingface implementations of OPT [Zhang et al., 2022] and Llama-1 [Touvron et al., 2023], open-source alternatives to GPT. We chose OPT because the authors provided detailed hyperparameter configurations for training models of comparable size, and Llama-1 because we incorporated many design choices from it into our architecture, facilitating a more direct comparison. However, we ultimately replaced the training configurations proposed in the OPT paper with ours, as their configurations resulted in slightly higher perplexities for our setup.

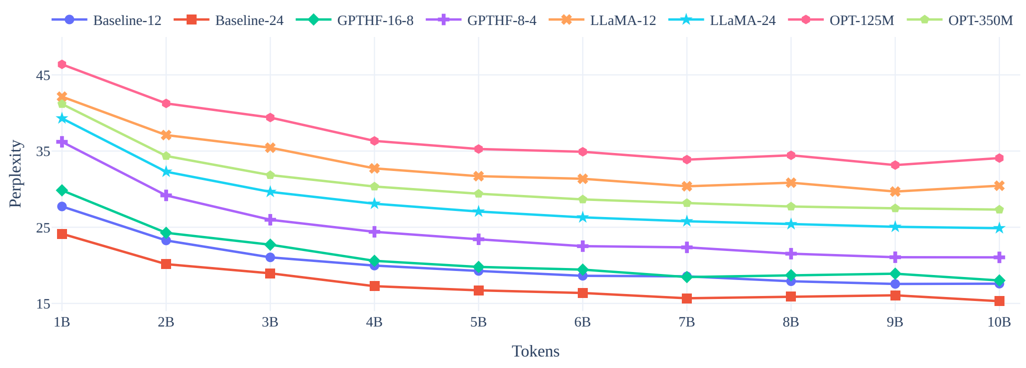


Figure 6.1: Validation perplexity of pre-trained models and baselines. Lower values indicate better performance.

Training Configurations. Due to computational constraints, we limited the pre-training of each model to 10 billion tokens. This means that the models were trained for 320,000 (micro-batch) steps at a context size of 512 with an effective micro-batch size of 64. The 12-layer models were trained on either 2 NVIDIA A6000 GPUs for 320,000 steps each or 4 NVIDIA RTX 3090 GPUs for 160,000 steps each. The 24-layer models were trained on 2 NVIDIA A100 GPUs for 320,000 steps each. Using this setup, training can be completed in 2-3 days.

6.2 Perplexity.

The pre-training perplexity after processing 10 billion tokens are presented in Figure 6.1. The perplexity scores are computed on a hold-out validation dataset comprising 1000 micro-batches, which totals 16 million tokens (using a micro-batch size of 32 at a context size of times 512). The perplexity scores are calculated on a per-subsequence basis, meaning that a document split into multiple sequences of 512 tokens is reset at the beginning of each subsequence, rather than using a more accurate but cumbersome "context sliding window" approach [Huggingface, 2020]. This method has the drawback of slightly overestimating the actual perplexity.

Llama-1 and OPT Have High Perplexity Scores. Unfortunately, our Llama-1 and OPT baselines perform subpar. This is likely due to the architectures not being optimized for a low-compute setting or sub-optimal training configurations. The configuration proposed by the OPT authors (no specific configuration were provided by Llama-1 for our model sizes) resulted in even higher perplexity scores. Due to time constraints, we did not optimize the configurations for these baselines. Consequently, for the remainder of this chapter, we focus on comparing GPTHF with "Baseline-12" and "Baseline-24".

Scaling Law Holds for GPTHF in the Low-Compute Setting. Our reference baselines yield significantly lower perplexity scores than the GPTHF models, as expected. Interestingly, scaling law appears to hold for GPTHF (in the low-parameter regime) as well. Morever, the same shift in perplexity, approximately 5 points from a 12-layer model to a 24-layer model of the same architecture after 10B tokens, is observed for both the baseline and GPTHF architecture. This indicates that it may be possible to compensate for performance loss due to compression simply by increasing model size. Perhaps even at an equally reliable rate as for conventional GPTs, as suggested by the same shift of 5 points, but clearly it is difficult to extrapolate this result to large-scale models.

Tripling the Model Size Brings Performance Back. We observe that the perplexity plots of GPTHF-16-8 and the 12-layer baseline are very similar. This sets up a basis for further comparisons: If GPTHF-16-8 can achieve faster generation efficiency and/or speed than the baseline of 12 layers, it might be worthwhile to invest in training a larger model capable of compression.

6.3 Experiments on Generation Speed

The primary motivation for this study is to leverage sentence compression to gain efficiency in inference, in the hope to justify an expected drop in predictive performance compared to a standard GPT. To achieve this, we measure FLOPs (floating-point operations) as a robust indicator of the time and energy cost of our model. In addition, we provide an analysis of the actual runtime.

6.3.1 FLOPs

Our fast generation algorithm is designed to achieve speedups; however, the extent of this speedup depends heavily on the distribution of tokens into sentences. Intuitively, having many sentences is beneficial because completed sentences can be cached and hence bypass the encoder, which comprises two-thirds of the model. Therefore, analyzing the FLOP count theoretically is impractical. Instead, we analyze it by running a number of samples from OpenWebText with varying lengths as prompts, and count the FLOPs empirically using a specialized tool [Li]. The results are presented in Table 6.1. Note that KV-caching [Pope et al., 2023] is disabled for our experiments, significantly impacting our results, as it requires a considerable effort to make our approach compatible with KV-caching, which was not pursued due to time constraints.

Efficiency Gain Increases With Prompt Length. The results indicate that the longer the prompt, the higher the efficiency factor, as expected, confirming

Batch size 1

Model	$n \leq 100$				$n \leq 250$			
	$k = 100$	efficiency	$k = 250$	efficiency	$k = 100$	efficiency	$k = 250$	efficiency
Baseline-12	2.38T	1.00x	9.1T	1.00x	4.88T	1.00x	15.7T	1.00x
GPTHF-8-4	0.95T	2.51x	4.16T	2.19x	0.80T	**6.10x**	4.31T	3.64x
Baseline-24	8.30T	1.00x	31.4T	1.00x	17.0T	1.00x	53.9T	1.00x
GPTHF-16-8	2.99T	2.78x	17.4T	1.81x	2.97T	**5.72x**	17.5T	3.08x

Batch size 32

Model	$n \leq 100$				$n \leq 250$			
	$k = 100$	efficiency	$k = 250$	efficiency	$k = 100$	efficiency	$k = 250$	efficiency
Baseline-12	2.46T	1.00x	9.62T	1.00x	4.96T	1.00x	16.0T	1.00x
GPTHF-8-4	1.90T	1.29x	7.72T	1.25x	2.53T	**1.96x**	9.32T	1.72x
Baseline-24	8.52T	1.00x	32.69T	1.00x	17.2T	1.00x	54.9T	1.00x
GPTHF-16-8	6.11T	1.39x	25.6T	1.28x	8.39T	**2.05x**	31.3T	1.75x

Table 6.1: Empirical count of Tera-FLOPs consumed per sample for different prompt lengths n and number of tokens generated k. Lower values are better. Bold values indicate the configuration with the highest efficiency gain for each batch size. The mean over 50 batches is reported. For GPTHF models, the number corresponding to the fast generation algorithm is reported. Efficiency is calculated as the inverse FLOPs reduction of the GPTHF model compared to its respective baseline.

our intuition at the beginning of this section. However, that would entail that more tokens would magnify our speedup, as the fast generation method bypasses sentence generation once a sentence is completed. However, the numbers do not fully support this hypothesis. An explanation is provided below.

Inability to Generate End-of-Sentence Embeddings. The trends in our results show that the efficiency gain is negatively correlated with the number of tokens generated, contrary to our initial intuition. A close examination of the generated text reveals the answer: Our models are only able to generate a few relevant tokens, but seem to repeat those tokens indefinitely without ever generating an end-of-sentence token. This problem occurs both in GPTHF models and baselines, indicating that it is likely due to insufficient scale or training rather than a limitation of the compression procedure.

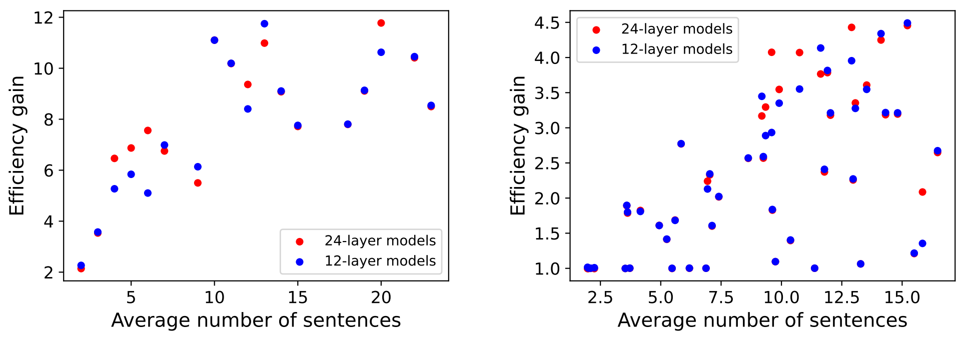
While quality of text normally is not coupled to efficiency, it is tightly coupled in the case of the GPTHF, as generating end-of-sentence embeddings is crucial for the fast generation algorithm to detect a finished sentence embedding. To underline this issue, we augment our initial experiment with two additional settings: a) when prompting 20 tokens and generating 500 tokens, and b) When prompting 500 tokens and generating 20 tokens. The results are shown in Table 6.2. In the former setting, we observe only tiny efficiency gains, as the model is unable to

Model	Batch size 1				Batch size 32			
	$n, k = 20, 500$	efficiency	500, 20	efficiency	$n, k = 20, 500$	efficiency	500, 20	efficiency
Baseline-12	21.8T	1.00x	1.56T	1.00x	27.1T	1.00x	1.7T	1.00x
GPTHF-8-4	21.2T	1.03x	0.17T	**9.18x**	21.7T	1.25x	0.58T	**2.93x**
Baseline-24	78.7T	1.00x	5.45T	1.00x	83.1T	1.00x	5.95T	1.00x
GPTHF-16-8	69.2T	1.14x	0.56T	**9.73x**	76.4T	1.10x	2.04T	**2.92x**

Table 6.2: Empirical number of FLOPs per sample for extreme values of prompt length n and number of tokens generated k. The setup is the same as in Table 6.1. This table illustrates: a) the issue that our models struggle to predict sentence boundaries, hindering efficiency gains, and b) the extent of efficiency gains that can be obtained when processing large text segments with regular sentence length distributions.

terminate sentences by predicting end-of-sentence embeddings. However, when given 500 tokens of prompt (packed in well-formed sentences), we observe efficiency gains up to an order of magnitude, and almost triple when comparing GPTHF-16-8 to the 12-layer baseline.

We hypothesize that a model capable of correctly terminating sentences achieves much greater efficiency gains than reported in Table 6.1. They would be closer to the efficiency gain observed in the 500 prompt, 20 generated tokens scenario, and would likely magnify even further with more tokens.



(a) Batch size = 1 (b) Batch size = 32

Figure 6.2: Scatter plots showing the average number of sentences (x-axis) versus the efficiency gain (y-axis) of the GPTHF model over its equal-sized baseline when generating 20 tokens. The left plot represents a batch size of 1, the right plot a batch size of 32.

Sentences vs Efficiency. To better understand the influence of the number of sentences present in the prompt on efficiency, we provide scatter plots in Figure 6.2 that display the average number of sentences on the x-axis and the speedup gain of the GPTHF model over the baseline on the y-axis. The figure demonstrates that the efficiency gain increases asymptotically with the average number of sentences, in particular that the relationship appears to be linear. For batched data, the efficiency gain is smaller compared to unbatched data, likely due to the increased variety (which can be observed from the increased variance) in tokens leading to more padding tokens being processed, which inhibits the gain from the fast generation algorithm.

6.3.2 Inference Time

A skeptical reader might argue that while we save many FLOPs, not all of these savings can be converted into reduced actual running time. The skepticism is valid because our algorithm includes considerable overhead and tracking, conditional executions etc. In other words, code that goes beyond straightforward matrix multiplications may not run as efficiently on a GPU. Therefore, we measure actual inference times, with the entire setup identical to the experiment for FLOPs.

	Batch size 1							
Model	$n \leq 100$				$n \leq 250$			
	$k = 100$	speedup	$k = 250$	speedup	$k = 100$	speedup	$k = 250$	speedup
Baseline-12	1.73s	1.00x	4.44s	1.00x	1.82s	1.00x	4.77s	1.00x
GPTHF-8-4	1.77s	0.98x	4.46s	1.00x	1.77s	1.03x	4.48s	**1.06x**
Baseline-24	3.40s	1.00x	8.88s	1.00x	3.73s	1.00x	9.85s	1.00x
GPTHF-16-8	3.32s	1.02x	8.43s	1.05x	3.32s	1.12x	8.44s	**1.17x**
	Batch size 32							
Model	$n \leq 100$				$n \leq 250$			
	$k = 100$	speedup	$k = 250$	speedup	$k = 100$	speedup	$k = 250$	speedup
Baseline-12	0.17s	1.00x	0.57s	1.00x	0.28s	1.00x	0.88s	1.00x
GPTHF-8-4	0.15s	1.13x	0.50s	1.14x	0.18s	1.56x	0.56s	**1.57x**
Baseline-24	0.40s	1.00x	1.42s	1.00x	0.73s	1.00x	2.34s	1.00x
GPTHF-16-8	0.35s	1.14x	1.24s	1.15x	0.37s	**1.97x**	1.29s	1.81x

Table 6.3: Empirical number of Generation time in seconds per sample for different prompt lengths (denoted by n) and number of tokens generated (denoted by k). Lower values are better. Bold values indicate the configuration with the highest speedup gain for each batch size. The mean over 50 batches executed on a single NVIDIA RTX A6000 is reported. For GPTHF models, the measurement corresponding to the fast generation algorithm is reported. Speedup is calculated as the inverse time reduction of our model in comparison to the baseline.

Model	Batch size 1				Batch size 32			
	$n,k = 20,500$	speedup	$500,20$	speedup	$n,k = 20,500$	speedup	$500,20$	speedup
Baseline-12	1.73s	1.00x	4.44s	1.00x	1.54s	1.00x	0.093s	1.00x
GPTHF-8-4	1.77s	0.98x	4.46s	1.00x	1.33s	1.16x	0.041s	**2.27x**
Baseline-24	3.40s	1.00x	8.88s	1.00x	3.65s	1.00x	0.26s	1.00x
GPTHF-16-8	3.32s	1.02x	8.43s	1.05x	3.4s	1.07x	0.087s	**2.99x**

Table 6.4: Empirical number of generation time in seconds per sample for different prompt lengths (denoted by n) and number of tokens generated (denoted by k). The mean and standard deviation over 100 samples on a single NVIDIA RTX A6000 is reported. For our models, the fast generation number is reported. Speedup is calculated as the inverse time reduction of our model in comparison to the baseline.

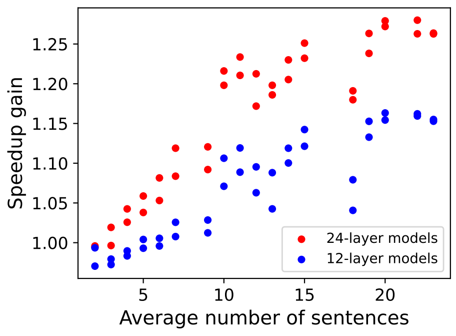
Speedup Increases With Context. Similar to the FLOP experiment, we observe that the longer the prompt, the greater the speedup. For unbatched input, speedup seems to increase with the number of generated tokens as well, but the gains are limited, up to 17%. For batched data, a larger context yields higher speedup, but more tokens do not, which we attribute to the same problem of the model failing to terminate sentences correctly.

Latency vs. Throughput. We observe significant differences in speedup gains between unbatched and batched data. We attribute this observation to be a classical latency vs. throughput issue. For unbatched data with small contexts, fast generation is slightly slower, likely due to the overhead of caching and additional operations. If too little data is available, the gains in FLOPs result in parts of the GPU simply staying idle. Consequently the runtime is bound by latency, primarily dependent on the model size, thus failing to translate into actual runtime gains.

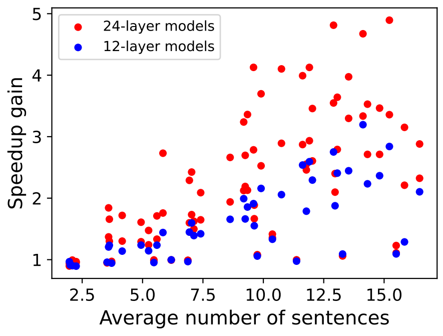
When batched, there is enough data to keep the GPU busy, converting efficiency gains in FLOPs into higher throughput. Moreover, the speedup seems to increase with model size as well [1]. This results in speedups up to triple (for the $n = 500, k = 20$ setting, in Table 6.4) when comparing GPTHF with their equal-sized baselines and slightly faster when comparing the GPTHF 16-8 with the 12-layer baseline. We conclude that runtime improvements are primarily about how effectively we can keep the GPU busy.

Sentences vs. Speedup. Similar to the FLOPs analysis, we provide scatter plots (cf. Figure 6.3 that display the average number of sentences on the x-axis

[1]Contrary to the higher speedups with longer context, it is reasonable to assume that this will not continue increasing indefinitely, but will rather saturate at a certain threshold in model size.



(a) Batch size = 1



(b) Batch size = 32

Figure 6.3: Scatter plots showing the average number of sentences (x-axis) versus the speedup gain (y-axis) of the GPTHF model over its equal-sized baseline when generating 20 tokens. The left plot represents unbatched input, while the right plot represents a batch size of 32.

and the speedup gain in runtime of the GPTHF model over the baseline on the y-axis. The figure highlights a linear relationship between the number of sentences and the speedup. As noted earlier, for unbatched data the speedup is limited, while the speedup also seems to increase with model size, likely due to our latency versus throughput considerations.

6.4 Discussion

Compression Leads to a Performance Drop. We conclude that compression does lead to a notable performance drop. Specifically, transitioning from a baseline/GPT model to a GPTHF model (for small sizes) results in a shift of approximately 5 points higher in perplexity after 10B tokens of training. This performance drop is comparable to the decrease observed when reducing a 24-layer GPT to a 12-layer GPT. Moreover, the perplexities of a GPTHF-16-8 and a 12-layer baseline are equivalent, as shown in Figure 6.1.

Scaling Potential of GPTHF Models. A promising observation from our results is that the GPTHF models could scale. The shift of 5 points in perplexity is observed when moving from a GPTHF-16-8 to a GPTHF-8-4. The important question is whether this relationship holds in the high-scale regime. A positive answer would entail that sentence embeddings can adequately replace sub-word embeddings if a larger model is used to compensate for the compression. This tradeoff is not only understandable, but is worthwhile if a compressed model comes with a corresponding speedup. On the other hand, there is a possibility that the quality of text generated by a compression model hits a natural ceiling, possibly due to information-theoretic limits when compressing. If this is the case, sub-word embeddings would remain superior to sentence embeddings for high-quality text generation.

Substantial Gains in Efficiency and Speed. Our approach achieves substantial speed-ups, up to an order of magnitude in FLOPs and up to triple in runtime when comparing equally sized models. Moreover, we observed empirically that these speedup factors increase asymptotically (linearly) with context size. It is important to keep in mind that our results are obtained with KV-caching [Pope et al., 2023] disabled. KV-caching could exhibit similar asymptotic properties, potentially making our fast generation method an alternative, but not necessarily an improvement over KV-caching. The exact factors would have to be carefully studied to make a definitive statement.

Is It Worth It? To evaluate the overall tradeoff, we compare the GPTHF-16-8 and the 12-layer baseline, which have similar perplexities. When processing 500 tokens of context, the GPTHF-16-8 uses roughly a third of the FLOPs for unbatched data and is slightly faster (7%) for batched data. While we have not found simultaneous improvements for both FLOPs and runtime in our configurations, such improvements are to be expected when increasing the prompt length and batch size due to the asymptotical result. Therefore, in the low-compute scale, this tradeoff appears worthwhile, although this conclusion again disregards the potential impact when including KV-caching.

Chapter 7

Limitations and Future Work

We outline several ways to build on and improve this project.

Scaling Up. The central question remains: Can a transformer generate high-quality text using only compressed sentence embeddings if provided with sufficient size and training? Informally speaking, the existence proof has yet to be done. For GPTHF models of our size, this does not seem to be the case, as proved in their inability to finish sentences. It is difficult to entangle if this limitation stems from insufficient scale or inherent challenges of text generation using sentence embeddings. Our perplexity plots suggest that small GPTHF models follow scaling laws similar to conventional GPTs. It is unclear if this relationship will hold for larger models or if there is a performance ceiling for GPTHF. We consider this the most significant limitation. A larger budget for trainingcould allow for a more thorough investigation.

Downstream Evaluation. Once sufficiently large models are obtained, we should not rely solely on perplexity as an evaluation metric. Future work should include downstream tasks to assess the practical effectiveness of the models.

Compatibility or Competition with Existing Optimizations. In order to obtain a true speedup for existing language models in practice, it would have to be compatible with existing optimizations, most importantly KV-caching, and re-evaluated for speed gains with the optimizations. A "diminishing-in-returns" phenomenon is possible, where the gains would be much more limited, and the growing of efficiency factors would not grow linearly in the number of sentences. However, our approach also does not contradict KV-caching, leaving this as a possible task.

Ablation Studies. Future studies could explore the impact of various individual factors such as the number of layers, hidden size, and other parameters on THF performance. The most interesting parameter is likely the hidden size, as it determines the size of the bottleneck. With more computational resources, comprehensive ablation studies could provide deeper insights into what contributes most to model performance.

Alternative Approaches. Our current model generates tokens from sentence embeddings, which can be thought of as a preliminary compression step compared to directly reducing a sequence to one embedding. We did not explore predicting a sentence embedding and training a decoder to decode those sentences due to the project's scope and additional challenges this approach would entail. This approach could be interesting to investigate in future work.

Chapter 8

Conclusion

In this thesis, we have questioned the established notion of large language models (LLMs) to operate on sub-words by exploring the idea of compressing entire sentences into single embeddings. Our goal was to leverage this idea to obtain speedups in language models, in addition to other known methods such as pruning, quantization, and knowledge distillation.

We demonstrated how to implement a compression bottleneck into existing architecture by building the THF, a hierarchical transformer, with a bidirectional and a generative variant. The surprising (and theoretically pleasing) takeaway is that we can achieve this with minimal changes to existing training procedures by using sparse attention matrices dynamically computed for each input.

Our evaluations of the generative model on perplexity and generation speed reveal several important findings. First, GPTHF models follow scaling laws, at least in the low-compute setting, with roughly the same perplexity shifts (5 points) as for conventional GPTs. The answer to the question of whether this finding holds true in a large-scale setting would lead us to the answer to a central research question: whether transformers are capable of compressing sentences into individual embeddings, and whether high-quality text generation is possible using only these embeddings.

Second, we have seen that our generative models exhibit impressive efficient gains: When given enough context, up to ten times more efficient in terms of FLOPs and approximately three times faster in terms of generation speed compared to their counterparts. This is possible due to a new fast generation method that caches sentence embeddings, which mathematically remain fixed once completed. The speedup factors increase linearly with the number of sentences present in the context.

When considering FLOPs, the GPTHF-16-8 model is significantly more efficient than a 12-layer GPT model. Given their equal perplexities, this makes for a worthwhile trade-off. Achieving the same level of efficiency in actual runtime is more challenging. The speedup factor depends significantly on many factors, such as batch size, context size and model size. This is due to latency vs. throughput considerations. Despite this, given sufficient context and batch size, we found that GPTHF-16-8 is faster than a 12-layer baseline, making this a worthwhile trade-off when handling large data. Our results were obtained with KV-caching disabled.

In conclusion, our exploration of compressing sentences into single embeddings for LLMs offers potential for promising speedup gains, particularly when processing large amounts of data. More research with more rigorous evaluation, including downstream tasks, in the high-scale regime and with KV-caching enabled is needed to fully understand the potential and limitations of this approach.

Bibliography

M. Augasta and T. Kathirvalavakumar. Pruning algorithms of neural networks—a comparative study. *Open Computer Science*, 3(3):105–115, 2013.

S. Bird, E. Klein, and E. Loper. Natural language toolkit (nltk). `https://www.nltk.org/`, 2023. Version 3.6.5.

T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, et al. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24 (240):1–113, 2023.

Z. Dai, G. Lai, Y. Yang, and Q. Le. Funnel-transformer: Filtering out sequential redundancy for efficient language processing. *Advances in neural information processing systems*, 33:4271–4282, 2020.

J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

W. Fedus, B. Zoph, and N. Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022.

J. Geiping and T. Goldstein. Cramming: Training a language model on a single gpu in one day. In *International Conference on Machine Learning*, pages 11117–11143. PMLR, 2023.

J. Gou, B. Yu, S. J. Maybank, and D. Tao. Knowledge distillation: A survey. *International Journal of Computer Vision*, 129(6):1789–1819, 2021.

D. Hendrycks and K. Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.

I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *Journal of Machine Learning Research*, 18(187):1–30, 2018.

Huggingface. Perplexity of fixed-length models. https://huggingface.co/transformers/v3.2.0/perplexity.html, 2020. Accessed: 2024-06-15.

Y. Kim and A. M. Rush. Sequence-level knowledge distillation. *arXiv preprint arXiv:1606.07947*, 2016.

D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

M. M. Krell, M. Kosec, S. P. Perez, and A. Fitzgibbon. Efficient sequence packing without cross-contamination: Accelerating large language models without impacting performance. *arXiv preprint arXiv:2107.02027*, 2021.

C. Li. flops-profiler. https://pypi.org/project/flops-profiler/.

Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

I. Loshchilov and F. Hutter. Fixing weight decay regularization in adam. 2018.

P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz. Pruning convolutional neural networks for resource efficient inference. *arXiv preprint arXiv:1611.06440*, 2016.

I. Montero, N. Pappas, and N. A. Smith. Sentence bottleneck autoencoders from transformer language models. *arXiv preprint arXiv:2109.00055*, 2021.

N. Muennighoff. Sgpt: Gpt sentence embeddings for semantic search. *arXiv preprint arXiv:2202.08904*, 2022.

H. Naveed, A. U. Khan, S. Qiu, M. Saqib, S. Anwar, M. Usman, N. Barnes, and A. Mian. A comprehensive overview of large language models. *arXiv preprint arXiv:2307.06435*, 2023.

P. Nawrot, S. Tworkowski, M. Tyrolski, Ł. Kaiser, Y. Wu, C. Szegedy, and H. Michalewski. Hierarchical transformers are more efficient language models. *arXiv preprint arXiv:2110.13711*, 2021.

D. Patterson, J. Gonzalez, Q. Le, C. Liang, L.-M. Munguia, D. Rothchild, D. So, M. Texier, and J. Dean. Carbon emissions and large neural network training. *arXiv preprint arXiv:2104.10350*, 2021.

R. Pope, S. Douglas, A. Chowdhery, J. Devlin, J. Bradbury, J. Heek, K. Xiao, S. Agrawal, and J. Dean. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems*, 5, 2023.

M. Popel and O. Bojar. Training tips for the transformer model. *arXiv preprint arXiv:1804.00247*, 2018.

A. Radford, K. Narasimhan, T. Salimans, I. Sutskever, et al. Improving language understanding by generative pre-training. 2018.

A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

N. Reimers and I. Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*, 2019.

V. Sanh, L. Debut, J. Chaumond, and T. Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.

N. Shazeer. Glu variants improve transformer. *arXiv preprint arXiv:2002.05202*, 2020.

N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, and J. Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

E. Strubell, A. Ganesh, and A. McCallum. Energy and policy considerations for deep learning in nlp. *arXiv preprint arXiv:1906.02243*, 2019.

J. Su, M. Ahmed, Y. Lu, S. Pan, W. Bo, and Y. Liu. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.

H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

K. Wang, N. Reimers, and I. Gurevych. Tsdae: Using transformer-based sequential denoising auto-encoder for unsupervised sentence embedding learning. *arXiv preprint arXiv:2104.06979*, 2021.

Z. Yang, D. Yang, C. Dyer, X. He, A. Smola, and E. Hovy. Hierarchical attention networks for document classification. In *Proceedings of the 2016 conference of the North American chapter of the association for computational linguistics: human language technologies*, pages 1480–1489, 2016.

B. Zhang and R. Sennrich. Root mean square layer normalization. *Advances in Neural Information Processing Systems*, 32, 2019.

S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

Appendix A

Experiments on Bidirectional Model

A.1 Bidirectional Model

A.1.1 Fine-Tuning on GLUE.

The General Language Understanding Evaluation (GLUE) benchmark is a collection of diverse natural language understanding tasks designed to evaluate the general language understanding a model has obtained after pre-training.

We report the median GLUE scores over five runs for each task, ensuring robustness of the results. The results can be found in Table A.1. The same hyperparameters were used consistently across all GLUE tasks, as follows:

- Learning Rate: 4e-5
- Batch Size: 16
- Epochs: 3
- Optimizer: AdamW with weight decay
- Dropout is re-enabled with a value of 0.1

During fine-tuning for sequence classification tasks, we discarded the decoder component, as it did not contribute to performance improvements. This procedure was consistently applied across various GLUE tasks.

We see that the bidirectional THF models are outperformed equal-sized and sometimes smaller baselines. The reasons for this were not further investigated due to time constraints.

Model	CoLA	MNLI	MRPC	QNLI	QQP	RTE	SST-2	STSB	Avg.
CrammedBERT-12	42.6	78.1	87.9	85.6	85.5	57.4	89.2	84.6	
CrammedBERT-10	48.3	79.0	87.4	86.3	86.3	57.0	88.5	84.8	
CrammedBERT-8	45.3	78.2	86.6	85.8	85.8	56.7	89.7	85.0	
THF-every2-12	35.1	76.1	86.0	84.5	84.6	56.0	87.5	83.7	
THF-every4-12	42.2	76.7	88.6	84.2	84.4	58.5	88.4	84.0	
THF-8-4-2	45.4	76.4	83.2	84.2	84.8	54.5	86.6	82.4	
GPTHF-16-8	46.6	79.1	1.07x	99.2	91.8	84.8	99.2	79.6	

Table A.1: Results of our models on the GLUE dev set. For each value, the median over 5 runs is reported. THF-everyN denote baselines that instead of a sentence bottleneck have a bottleneck by compressing every group of N embeddings into one.