Compressing code generation language models on CPUs

Using Group Lasso pruning and post-training quantization

Dan Sochirca

Supervisor(s): Prof. Dr. Arie van Deursen, Dr. Maliheh Izadi , ir. Ali Al-Kaswan

EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 25, 2023

Name of the student: Dan Sochirca
Final project course: CSE3000 Research Project
Thesis committee: Prof. Dr. Arie van Deursen, Dr. Maliheh Izadi, ir. Ali Al-Kaswan, Avishek Anand

Parts of the document were generated using the following AI tools: ChatGPT.

Abstract

Code generation models have become more popular recently, due to the fact that they assist developers in writing code in a more productive manner. While these large models deliver impressive performance, they require significant computational resources and memory, making them difficult to deploy and expensive to train. Additionally, their large carbon footprint raises environmental concerns. To address these challenges, there is a need to develop techniques for compressing these models while maintaining their performance.

In this work, we study the effectiveness of *Group lasso pruning and post-training quantization* techniques on CPUs, applied to the code generation model CodeGPT. We evaluate the performance of the compressed model using the Exact Match (EM) and Edit Similarity (ES) metrics and study the model size on disk, memory footprint, and CPU inference. In contrast with the original CodeGPT model, our solution offers a 48% relative reduction in disk size, with only a mild drop in the accuracy metrics: 8.51% absolute drop in ES and a 5.5% in EM. Using the ONNX runtime on a regular laptop, we are able to deliver a 2x inference speedup at a 32.6% reduction in size. Our code is publicly available at https://github.com/AISE-TUDelft/LLM4CodeCompression/tree/main/CodeGPT-on-Intel.

1 Introduction

Language models have witnessed significant advancements in various natural language processing tasks, including question-answering, translation, sentiment analysis, and text classification. Among these tasks, automatic code completion plays a crucial role in enabling software developers to write code more efficiently. However, deploying large language models (LLMs) poses challenges due to their computational complexity and high memory requirements. Their carbon footprint also raises concerns: the CO_2 emission of training the GPT-3 model amounts to three times that of a whole jet plane from San Francisco to New York [28]. Therefore, there is a need to explore techniques that can compress these models, making them use less energy and be more feasible for local deployment.

In recent years, tools such as Github Copilot[1] and Amazon CodeWhisperer[2] have experienced a surge in popularity due to their ability to improve developer productivity. These tools aim to provide developers with intelligent real-time code suggestions based on the user's coding context. The technology leverages the power of the GPT model architecture [30], which builds upon the advancements introduced by the original transformer [39]. As a result of these breakthroughs, modern GPT-based models have achieved remarkable progress in comprehending and generating human-like text. Furthermore, this paradigm shift in code generation has revolutionized the way developers interact with their integrated development environments (IDEs). Instead of relying solely on manual coding or limited code completion features, developers can now benefit from sophisticated AI-powered assistants that augment their coding experience.

Compression techniques such as pruning, quantization, and distillation have been proposed to make code-generation models more efficient. However, their application to GPT models for code completion remains limited, due to their novelty and the higher popularity of BERT models. While some studies have explored compression techniques for GPT models [21; 27; 38; 43], they have not focused on code completion tasks specifically.

Our work specifically focuses on CodeGPT, a transformer-based model fine-tuned by us for code completion. We target the compression and deployment of this model on CPU architectures, which is a more accessible resource to the everyday person rather than GPUs. We investigate pruning and quantization techniques applied post-training. In particular, we aim to answer the following *research questions*:

- **RQ1:** How does the performance of the CodeGPT model change after applying Group Lasso pruning?
- **RQ2:** How does the performance of the *pruned* model change after applying post training quantization?

Our study proposes a two-stage **approach** to tackle the research questions. Firstly, we use group lasso structured pruning at varying levels of sparsity to selectively remove unnecessary elements from the neural network. Next, we apply dynamic post-training quantization to further test the compression of the model. We experiment with different quantization mapping techniques and quantization scales to identify the best trade-off between model size, CPU inference, and performance.

Our **results** vary depending on what configuration is used when compressing. We identify a sweet spot at the 60% sparsity level in group lasso pruning, which results in a relative decrease in size by 48%, accompanied by an 8.51% absolute reduction in the ES score and 5.5% in EM. However, with the compression library we used, quantization did not to provide any speedups. Using the ONNX model format also provides double the inference speed when run on a regular laptop using an i9 11900H processor.

Our **contribution** can be summarized as follows:

1. The application of different compression techniques to our CodeGPT model: Group Lasso (post-training) pruning for different target sparsities (40-90%) and post-training quantization (PTQ) with channel-wise and tensor-wise quantization scales.
2. The evaluation of each technique and finding the optimal model.
3. Comparing our best approach to models part of other concurrent work: [5; 23; 37].
4. Publishing the used scripts on Github, for others to reproduce our work.

[1] GitHub Copilot: https://github.com/features/copilot

[2] Amazon CodeWhisperer: https://aws.amazon.com/codewhisperer/

By implementing these techniques, we aim to pave the way for more efficient and lower-carbon use of GPT-based language models in code completion tasks and to contribute to the advancement of this field.

2 Preliminaries

In this section, we provide an overview of the necessary concepts and techniques that support a better understanding of our study.

2.1 Transformers

Transformers [39] are the building block of many Large language models (LLMs) that we see today. A transformer takes in one sequence of words (like a sentence) and "transforms" it into another sequence (like a translation of that sentence). It's really good at this task because it can understand the context and meaning of the words in the sequence using what's called a "self-attention" mechanism.

A transformer is composed of two components: an **encoder** and a **decoder**. The encoder processes the input sequence and produces a sequence of hidden states, while the decoder uses the encoder's hidden states and the output sequence to generate a new sequence of output values. Both components consist of two sub-components: multi-head attention (**MHA**) and fully connected feed-forward network (**FFN**).

Currently, the two most popular LLM models in existing literature are BERT [7] and GPT [30]. BERT's architecture consists of bidirectional transformer encoders, while GPT uses unidirectional transformer decoders. "Bidirectional" means that the model considers the context of a word from both directions, i.e., it examines the words that come before and after a particular word in a sentence. This allows BERT to be better suited for tasks that require a deeper understanding of the relationships between different parts of a text, such as question-answering and sentiment analysis. GPT however processes words in a sentence sequentially (from left to right for example). This type of model is more suited for tasks that require generating coherent and fluent natural language text because it takes into account all the previously generated words when generating the next word. Because we study the code completion task, in this paper we focus on compressing a fine-tuned version of the **CodeGPT-small** model[3], based on the GPT-2-small architecture.

The GPT-2 [31] architecture is made of a stack of homogeneous Transformer decoder blocks, each of which consists of a masked MHA layer followed by a point-wise FFN layer. Specifically, an MHA layer consists of H independently parameterized attention heads [19]:

$$MHA(x) = \sum_{i=1}^{H} Att_i(x \cdot mask) \tag{1}$$

$$x_{MHA} = LayerNorm(x + MHA(x)) \tag{2}$$

[3]CodeGPT-small model: https://huggingface.co/microsoft/CodeGPT-small-py/tree/main

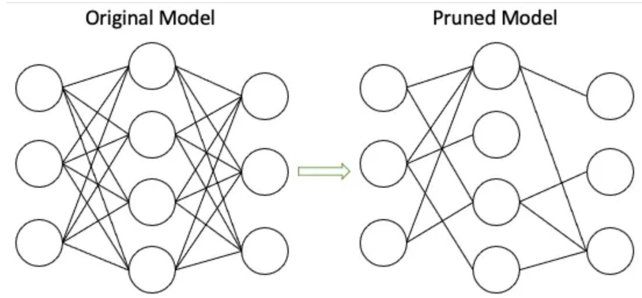


Figure 1: Pruning a neural network. The pruned model has fewer connections than the original model. Source: [16]

where Att is a masked dot product attention head, and x is the input sequence. The output of the MHA layer is then fed into the FFN layer, which consists of N filters:

$$FFN(x) = \sigma(XW^{(1)}) \cdot W^{(2)} \tag{3}$$

$$x_{out} = LayerNorm(x_{MHA} + FFN(x_{MHA})) \tag{4}$$

where $W^{(1)}$ and $W^{(2)}$ are the FFN parameters, and σ is the activation function, typically GELU. Both MHA and FFN layers are followed by a layer-normalization step, denoted by x_{MHA} and x_{out}. H is 12 for GPT-2.

2.2 Distillation

Knowledge distillation [10] is the process of training a student model to reproduce the behavior of a teacher model (training against the teacher's outputs). It leverages the logits (the input of softmax in the classification tasks) of teacher and student models to minimize the difference between their predicted class distributions, this can be done by minimizing the loss function $L_{KD} = D(z_t, z_s)$, where D is a distance measurement (for example the Euclidean distance and Kullback–Leibler divergence), and z_t and z_s are the logits of teacher and student model.

Distillation is not the main focus of this work. We do consider it however in our discussion section.

2.3 Pruning

Pruning removes redundant parameters/weights from a model by forcing them to zero. A nice visual representation is displayed in Figure 1. Pruning can be either **unstructured**, where individual weights are pruned, or **structured** where structured groups of weights are pruned, e.g. entire blocks, channels, layers. Pruning results in sparse neural networks that reduce the computation and the memory footprint of the trained model [9; 46].

Pruning can be applied to transformer LLMs in several ways: by pruning layers (a pair of MHA and FFN layers) [8; 33], attention heads [40; 25], FFN layers [29; 4; 24; 12]. In this work, however, we are going to prune all components.

A common way to prune is in iterative steps, each of which removes weights until a desired **sparsity level** is reached, where sparsity is computed as the number of pruned parameters divided by the full model size [18]. The motivation behind iterative pruning is that removing one weight can cause the remaining weights to become more important and change their values [18]. However, in this paper, we used one-shot pruning, which is less effective but computationally cheaper [3], because it prunes the entire model at once.

Additionally, pruning requires a fine-tuning step, in order to reduce the loss it induced. Therefore, the model needs to be retrained during/after it was pruned in order to preserve its accuracy.

2.4 Quantization

Quantization uses fewer bits to represent weights and activations. It converts a floating-point tensor to tensors with integer values, operations on which are more efficient [11]. The (linear) quantization formula is as follows:

$$X_q = round(\frac{X_{fp32}}{s} + z),\ where\ \ s = \frac{2^{B-1}}{\alpha} \quad (5)$$

Here s is the quantization scale, z is the zero point, B is the bitwidth (8 in case of int8) and α is the quantization range. Int8 is the most popular int range in literature, so we are going to use it as well. Two common range mapping techniques are **affine/asymmetric** quantization, where:

$$s = \frac{127}{|X_{f_{max}} - X_{f_{min}}|}, \quad z = -128 - X_{f_{min}} \cdot s \quad (6)$$

which maps the min/max range in the float tensor to the integer range (int8), and **scale/symmetric** quantization:

$$s = \frac{127}{max(abs(X_{f_{max}}), abs(X_{f_{min}}))}, \quad z = 0 \quad (7)$$

which uses the maximum absolute value in the float tensor as the float range.

We distinguish 2 methods to obtain a quantized network:

- *Quantization Aware Training (QAT).* QAT requires training the model from scratch with additional simulated quantization operations during training. This method is expensive due to the high cost of training but can lead to higher accuracy compared to PTQ because the model better adapts to the introduced quantization noise.
- *Post Training Quantization (PTQ).* PTQ quantizes an existing network without the need for the original training pipeline [17]. Models quantized by PTQ are highly susceptible to quantization noise since the model is not originally trained with quantized parameters. However, the low cost of applying this method makes it a very popular choice, which is why we decided to use it.

There are also different methods to choose the activation scale, which can be categorized as follows:

- *Dynamic quantization:* The quantization range α and scale s are determined on the fly and differ for each tensor. Additional scanning cost is incurred because the max value needs to be found for each tensor.
- *Static quantization:* The quantization range and scale are determined using a *calibration dataset* and are the same for all tensors. This method however can be susceptible to more quantization noise, though lower computational cost during inference [42].

The main problem with quantization is that it introduces additional noise in the network that can lead to a drop in the model's performance. This is mainly caused by outliers present in the tensor. To reduce the quantization noise and avoid outliers, column-wise (or channel-wise) quantization scales can be used [42], where the quantization scales are determined based on the max value in each tensor column/row/channel instead of the entire tensor. Outliers can also be avoided with data clipping before quantization is used [42].

In this work, we are using dynamic PTQ because of its low cost. We also used the popular int8 as the quantization range, due to its popularity and increased support.

3 Related Work

There has been significant research on compressing and quantizing large language models such as BERT and GPT, and several methods have been proposed for this purpose. However, none of them study the application of compression techniques on LLMs for the code completion task, therefore research in this specific area is limited. In this section, we provide a brief overview of some relevant works that are related to our research question.

Distillation-based methods have been widely used for compressing language models. For example, Sanh et al. introduced DistilBERT and showed that it's possible to reduce the size of a BERT model by 40% while retaining 97% of its language understanding capabilities and being 60% faster [34]. Shleifer and Rush proposed a pre-trained summarization distillation technique that can be used to compress summarization models [36].

Another way of compressing language models is pruning. Wang et al. proposed structured pruning of large language models, which uses a structured pruning technique based on adaptive low-rank factorization, to remove a certain percentage of weights from a model, resulting in a smaller model [41]. Similarly, Zafrir et al. proposed Prune Once For All (POFA), a technique that uses Gradual Magnitude Pruning (GMP) and Learning Rate Rewinding (LRR) to prune a pretrained model only once and then fine-tune it to obtain a smaller model with minimum accuracy loss [45].

Quantization is another technique that has been used to compress language models. Hu et al. evaluated several post-training quantization methods for language tasks and proposed an empirical evaluation framework for comparing these methods [13]. Bondarenko et al. proposed three solutions to overcome the challenges of efficient transformer quantization [2], while Yao et al. introduced ZeroQuant, an efficient and affordable post-training quantization method for large-scale transformers [44].

There are also works that combine pruning and quantization techniques for compression. Kwon et al. proposed a fast post-training pruning framework for transformers [20], while Jiao et al. introduced TinyBERT, a distilled BERT model that combines both pruning and distillation [15]. Kurtic et al. proposed an optimal BERT surgeon that uses second-order pruning for large language models [18].

Perhaps the most significant research for this paper would be the work of Shen et al., who developed a pipeline for accelerating Transformer models on CPUs [35]. By making use of several software acceleration techniques, their pipeline outperformed the state-of-the-art Neural Magic's DeepSparse runtime performance by up to 50%. Their work is critical, as we used their toolkit[4] to apply compression techniques in this paper.

4 Methodology

In this section, we introduce our chosen technique for compressing the CodeGPT model. We employ a two-step process: pruning and quantizing. We consider group lasso pruning for different percentages of sparsities and quantization for different layers and quantization scales. Each of these methods comes with its own trade-offs, which is why there can be more than one appropriate solution.

4.1 Group lasso structured pruning

In this study, we implement Group Lasso pruning, a method that extends the conventional Lasso technique by considering the group structure of the input variables. This approach allows us to remove whole groups of neurons or layers rather than individual nodes, resulting in a smaller dense network. Using this technique, we aim to reduce the complexity and size of our model.

Given a matrix w split into non-overlapping groups of parameters G, the Group Lasso pruning problem is formally defined as [1]:

$$R(w) = \lambda \sum_{g=1}^{G} \sqrt{s_g}\ ||w_g||_1 \tag{8}$$

where the 1-norm takes the square root of the squared sum of all weights in a group, to force them to go towards 0 together. The s_g scalar applies scaling and represents the group size (i.e. number of elements in a group). λ is the regularization parameter, controlling the degree of sparsity in the pruned model. In our experiments, λ is controlled through the sparsity level as a parameter in the intel-extension-for-transformers library [14]. Finally, the regularization term $R(w)$ is added to the original training loss function, and fine-tuning the model using this new function will successfully prune it.

We apply the pruning method for all layers in our model, at different levels of sparsity: 0.4, 0.5, 0.6, 0.7, 0.8, and 0.9. We prune the model in one step (one-shot pruning), instead of doing it iteratively, to speed up the compression. While the model is pruned it is also fine-tuned on 128 samples (we chose a relatively small number since this takes place on CPUs) in the dataset in order to reduce some of the loss incurred. Based on the evaluation results, we further compress the best-performing model using quantization.

4.2 Dynamic Post Training Quantization

After pruning, the model undergoes post-training quantization (PTQ) to further compress it. Specifically, we utilize dynamic post-training quantization (PTQ) with the int8 quantization range, as explained in section 2.4. An example of how the technique works is provided in figure 2.

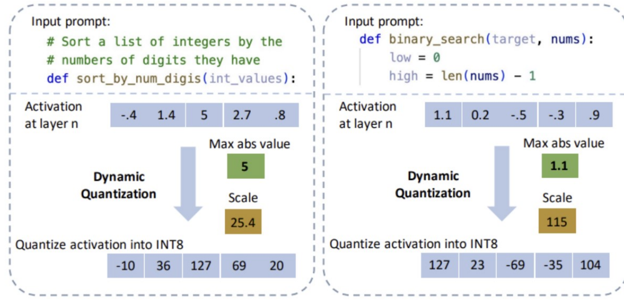


Figure 2: PTQ visualized, where the weight clipping range is determined dynamically. Each input prompt has a different max abs value and therefore gets a different scale. Source: [42]

We apply dynamic PTQ with different mapping techniques: symmetric and asymmetric, and at different scales: channel-wise and tensor-wise. We also vary the layers to which quantization is applied: in one scenario, we quantize all layers of the model, and in another, we only quantize the weights (excluding the activations). As we did in pruning, we also fine-tune the model after it was quantized on 100 samples to reduce some of the quantization noise. The impact of the differing scopes of quantization is evaluated in our results.

5 Experimental setup

Model. The model[5] we use in this work is a version of the CodeGPT-small model[6], fine-tuned on the code completion PY150 dataset[7] with one training epoch. The model contains 124.24 million parameters and its disk size is 486MB. Its score (evaluated on the test partition of the dataset) is ES of 39.05 and EM of 14.50%. We proceeded to use this model for our compression experiments.

Hardware. For running the experiments we use the DelftBlue [6] high-performance computer (HPC). We utilize the standard compute nodes, made of Intel XEON E5-6248R 24C 3.0GHz CPUs. Our setup is made of 8 CPUs, each with 4GB of memory. This amount of processors however is only used for parallel tokenization of data and model evaluation. Compressing the models was sadly not possible to parallelize on CPUs, and due to that fact, it can only run serially. Pruning a model takes almost 5 hours to complete, quantizing - around 30 minutes, and evaluating it takes ~10 minutes. An example Unix shell executable script as well as outputs of our python programs can be found in our repository.

Frameworks and libraries. Our deep learning library of choice is PyTorch[8]. To prune and quantize the models

[4]Intel Extension For Transformers: https://github.com/intel/intel-extension-for-transformers

[5]CodeGPT fine-tuned on the Py150 dataset: https://huggingface.co/AISE-TUDelft/CodeGPT-Py150/tree/main

[6]Original CodeGPT created by Microsoft: https://huggingface.co/microsoft/CodeGPT-small-py

[7]PY150 code completion dataset: https://huggingface.co/datasets/0n1xus/codexglue

[8]PyTorch: https://pytorch.org/

we make use of the intel-extension-for-transformers toolkit[9], which provides APIs for various model compression techniques. We also use the ONNX runtime[10] to convert the Hugging Face models into the ONNX format, which delivers significantly better inference results.

Evaluation. The resultant compressed models are evaluated on 1000 samples taken from the test partition of the same PY150 dataset, using the EM and ES metrics. For each approach, we study the model size on disk, memory footprint, and CPU inference time. The Exact Match metric measures the percentage of instances where the model's output exactly matches the expected output. The Edit Similarity metric however is based on the Levenshtein distance, which measures the minimum number of single-character edits (insertions, deletions, or substitutions) needed to change one string into another. Inference times are measured in samples/sec, while memory usage and disk size are in MB. Both the inference time and memory footprint will be computed by inferring the same samples from the test set.

The Exact Match metric. Our goal is to generate code that is functionally equivalent to a reference implementation rather than identical to it. Therefore the ES metric is to be considered more important than the EM metric, even though we are not getting a character-for-character match.

Measuring disk size. Regarding the disk size measure, we would like to point out that we report the size of the model after compression as a zip archive, instead of the actual file. For example, pruning sets the pruned weights to zero but does not actually remove them from the binary file, thus keeping the model's architecture and size the same. Zipping the model however does reduce its size, and this is reflected in our results. As a reference, zipping the baseline model reduces its size from 510 MB down to 462 MB.

6 Results

In this section, we present our evaluation results and examine how our model performs when converted to the more efficient ONNX format.

Pruning (RQ1). In Figure 3, we compare the metrics of the pruned model at different sparsity levels. As expected, the sparser the model - the more compressed it is and the lesser its disk size. The memory usage seems to slightly go up, but the CPU inference does not present any significant change. Both the ES and EM scores decrease as the model is pruned more, but we see a sweet spot at the 60% sparse model, which achieves a reduction in size by almost a factor of 2 while having an absolute ES decrease of 8.51%. The model has achieved pruning of a total of 74.43 million parameters out of a total of 124.2 million (hence where 60% sparsity comes from), leaving it with only 49.77 million. The disk size reduction in particular makes it an attractive choice, which is why we went forward with it.

Quantization (RQ2). We proceed to quantize the 60% sparse model. As explained in section 4.2, we apply quantization with various configuration options, resulting in a total of eight quantized models. Results are plotted in Figure 4. Memory usage remains relatively constant, while other measures vary depending on the quantization scheme. One thing that remains consistent is that the performance continues to decline, even more drastically for the EM metric. Another observation that can be made is that quantizing the weights alone, rather than the entire layer, results in better overall scores. Here, the best solution in terms of preserving accuracy seems to be the one where weights are quantized symmetrically per tensor, with an ES of 20.76 and an EM of 0.5.

Model name	Disk size	Mem usage	CPU inf	Edit sim	EM
original	462.26	2184.49	0.80	39.05	14.5
compressed	240.52	2189.63	0.80	30.54	9.0
ONNX inference	311.75	4166.37	**1.59**	30.54	9.0

Table 1: Measuring the impact on inference of using the ONNX model format. The inference improvement is highlighted in bold.

Local ONNX inference. We convert the optimal model (which in our opinion is the 60% sparse model) to the ONNX runtime format and we examine if there are any inference improvements. ONNX inference is supposed to boost the numbers thanks to hardware optimizations, but we were unable to test this on the DelftBlue cluster due to constant missing module errors we could not resolve. This might indicate that the library is not supported on Linux. Despite that, ONNX inference worked successfully on our local machine, which uses an i9 11900H processor. Results are shown in table 1. We report a significant increase in CPU inference by a factor of 2, at a cost of higher disk size and memory usage.

Summary. From our results, pruning at a sparsity level of 60% provides a good balance between model size and performance. Our solution offers a 48% reduction in disk size (relative to the original size) with a mild drop in accuracy: 8.51% absolute drop in ES and a 5.5% in EM. Running the model locally in the ONNX format (on an i9 11900H processor) delivers a **2x speedup** at a 32.6% reduction in size.

7 Discussion

In this section, we reflect on our results, compare them with those of similar work, and discuss the limitations, implications and possible threats to validity.

7.1 Reflection

Disk size. When pruning, the disk size consistently decreases, which is expected. However, we observe no pattern in quantization, and in some cases, we notice the number going up. We believe that the reason for this is that the quantization process has added additional metadata to the model file, storing additional information such as scale and zero-point parameters for each quantized layer. The parameters are necessary to dequantize the values back to the original range during inference.

[9] See footnote 4.

[10] The ONNX runtime library: https://huggingface.co/docs/optimum/main/en/onnxruntime/overview

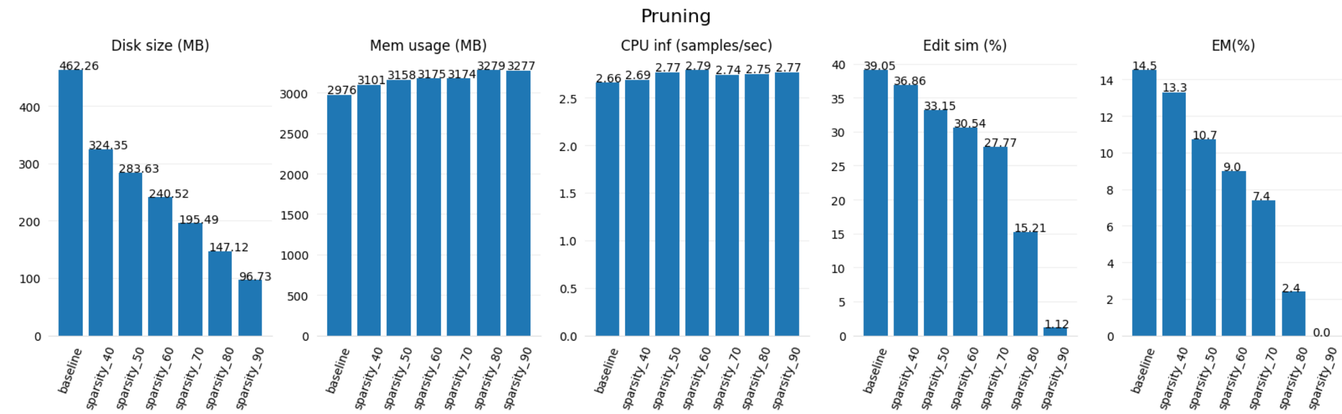


Figure 3: Evaluation results of pruning at different sparsities.

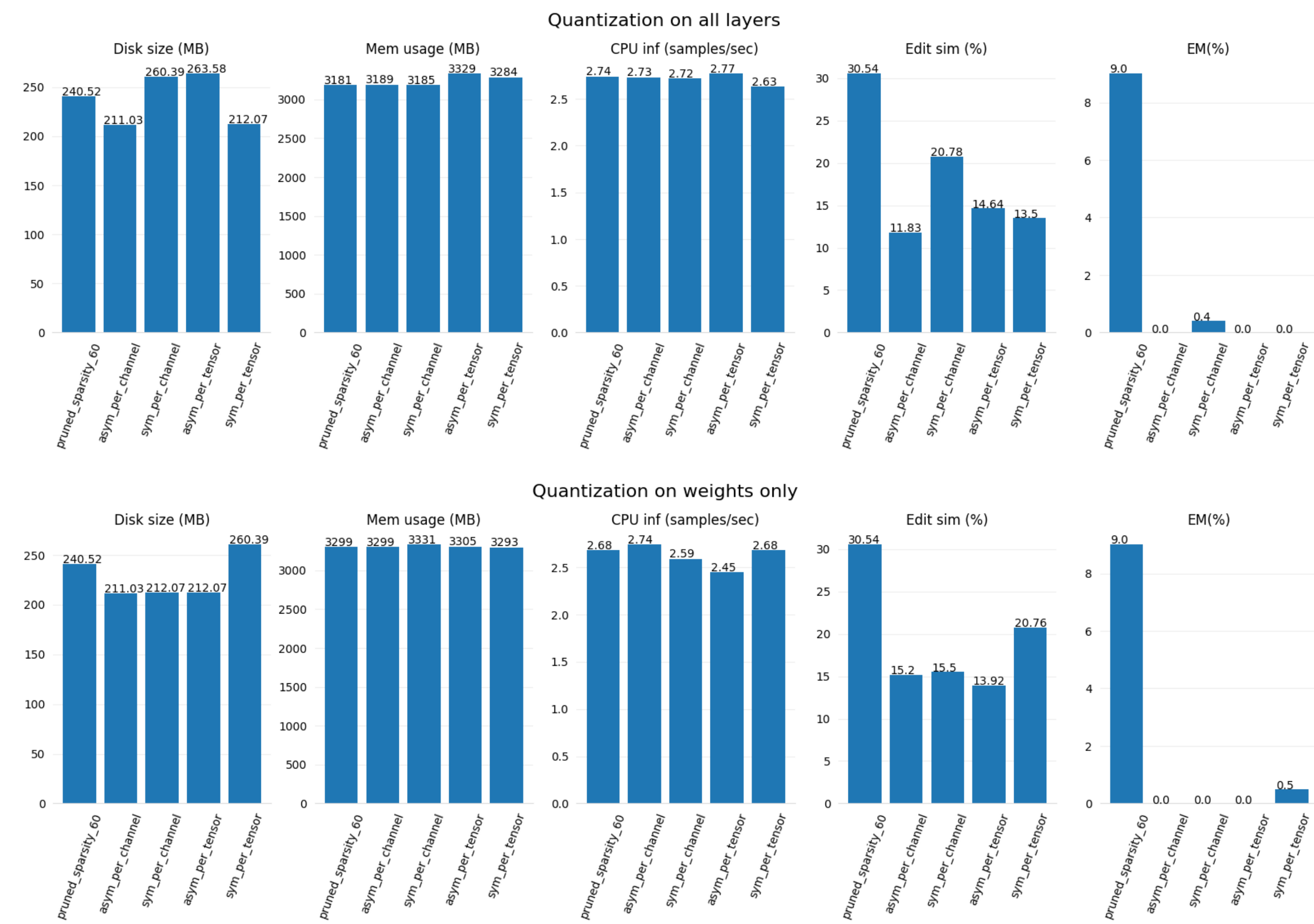


Figure 4: Evaluation results of quantizing with different configurations.

Variability in Memory usage and CPU inference. There is variability present in our memory usage and CPU inference measurements. We noticed the effect over multiple runs too. We have made all efforts to delete unused variables and induce manual garbage collection, so our assumption is that this can be attributed to the variable system load due to background processes over which we do not have control. Therefore, we can regard approaches that have relatively close memory usage/inference times as similar in performance. This leads to us concluding that we do not see any memory or inference improvement in our measurements.

CPU inference. We report no CPU inference improvement in the results. This is because even with fewer parameters, the model's architecture remains the same and the operations present on each layer still have to be performed. We also could not use an Inference Engine optimized for low-precision computations (such as int8), due to their missing support of our GPT-2 architecture and the text-generation task. This approach would potentially lead to better outcomes for inference in our quantized models, and we think this is something that needs to be further investigated in future research.

Quantization. Compared to pruning, quantization delivered significantly less accuracy. In our opinion, quantizing, in this case, is only worth the effort if an optimized Inference Engine is used. We also notice that, on average, using per-channel scales delivers slightly higher accuracy. This aligns with the findings of Wei et. al. [42], which claimed that using per-column scales reduces the quantization noise. In our case, however, the difference is not that pronounced, which might be because we are using a different model.

7.2 Comparison with concurrent work

In table 2 we compare our method with several other compression techniques applied to the CodeGPT model. In particular, the compressed models are:

- **CodeGPT on XTC** [5] - A hybrid model compressed using a 6-layer knowledge distillation reduction, then quantized on 1-bit precision weights and 8-bit precision activations.

  Their method demonstrates the highest reduction in disk size and the most impressive CPU and GPU inference speed increases. However, this achievement does not come without an accuracy compromise, which is very acceptable considering the low disk size.
- **Distill-CodeGPT** [23] - Compressed using in-training knwoledge distillation, and made up of 4 attention heads.

  The distilled model delivers quite a low GPU model size, and considerable memory usage and inference speed improvements. It has an ES score of 34.6, a slightly higher score than our solution and CodeGPT on XTC.
- **MP and PEG PTQ on CodeGPT** [37] - This work presents 2 equally performing models, that use post-training quantization with mixed precision and PTQ applied per embedding group. The paper simulates the quantization without changing the data type of the PyTorch tensor, and because of that, it does not provide any evaluation results other than the model size and the accuracy metrics.

  This model achieves the highest accuracy metrics but lacks empirical results for memory usage and inference speedups. It does not provide a complete picture due to its theoretical approach, so we cannot examine it in more detail.

It seems that each method offers unique advantages, and the results suggest that the MP and PEG PTQ methods work best in terms of accuracy. However, this method only provides simulated results, and if we consider both efficiency and performance, CodeGPT on XTC appears to be the best. In comparison, our method still achieves a significant reduction in model size and contains the lowest number of parameters. This result comes with its own trade-offs, but we think the model maintains a solid level of accuracy and performance.

7.3 Limitations

Compression library. Our work uses the intel-extension-for-transformers toolkit, which does not officially support the GPT-2 architecture that makes up our CodeGPT model[11]. This is reflected in our quantization results and the comparison with concurrent work, where we clearly see that some models outperform our method. While the compression process was successful, we were unable to use certain functionalities such as using the library's ONNX-export functionality and its optimized pipeline, which at the time of writing only supports the text classification task. That is the reason why we do not notice any inference speedup in our results.

Model. The CodeGPT model we used has a relatively small architecture, consisting of 124.2M parameters. Moreover, the model was fine-tuned for the code completion task only for a single epoch. This decision, while efficient in terms of time and computational resources, might have limited the baseline accuracy of the model, and furthermore can be the reason why the model performs so poorly when compressed. It is because of their higher size that code-generation models such as GitHub Copilot perform significantly better at this task. Using a bigger model, or training the current one for more epochs could potentially have led to improved results.

Despite these limitations, we believe our study provides a valuable starting point for future research in this area.

7.4 Implications

Accessible models. By compressing the CodeGPT model at almost half its size without a significant loss in performance, we have shown that it is possible to have efficient code completion tools that are more accessible for local deployment.

Carbon footprint and computational cost. Both the size reduction and the inference speedup achieved using ONNX inference indicate that less storage and data transfer are required to run and store the models. In turn, this means that fewer computational resources are needed to generate code suggestions. This further reduces overall energy use and shows great potential for making the deployment of large language models more environmentally friendly.

[11]Intel Extension model support: https://github.com/intel/intel-extension-for-transformers/blob/main/docs/examples.md

Model	Disk size	GPU size	CPU Mem. Usage decrease (%)	GPU Mem. Usage decrease (%)	CPU inf. increase (%)	GPU inf. increase (%)	Edit sim	EM	Parameters (Mil.)
Baseline	510	510	-	-	-	-	39.05	14.5	124.2
Our solution	240.5	-	-6.7	-	4.89	-	30.54	9	**49.8**
CodeGPT on XTC	**32.3**	341.8	10.58	-9.66	**100**	**60**	33.9	10.9	81.72
Distill-CodeGPT	280	**280**	**19.05**	**37.8**	41.2	39.5	34.6	8.3	68
MP & PEG PTQ	127.5	-	-	-	-	-	**40**	**17**	-

Table 2: Comparison with concurrent work. The highest values are highlighted in bold. We report only the approximate percentage improvement (i.e. increase/decrease) for some of the measurements due to the usage of different evaluation environments for each model.

7.5 Threats to validity

Insufficient documentation. The lack of comprehensive documentation for the intel-extension-for-transformers library posed several challenges. The toolkit's code lacks comments that would otherwise provide a better understanding of its inner workings, and its GitHub documentation only scratches the surface. This may have led to misinterpretations and potential misuse of certain functions, thereby impacting our study's outcomes and affecting the reliability of our results.

Overlooked bugs. We reviewed our code multiple times, and our results are consistent across multiple runs. We are extremely confident in our solution, but one can never say that their code is bug-free. There is always the risk of overlooked bugs, in both our code and the library's code. These potential errors could have significant effects on our experimental outcomes. To mitigate this, besides multiple inspections, we open-sourced our code and we welcome any contributions from the community, which might uncover potential errors.

Novelty of GPT model compression. The relative complexity of GPT models compared to BERT models and the limited research done so far in this specific area might have limited our ability to draw on established methods and compare our results. Our methods and results might not be entirely generalizable, and the real-world applicability of our approach to a broader range of GPT and other models remains uncertain. Nonetheless, we do contribute significantly to a better understanding of these methods and their trade-offs, and pave the way for future research in this exciting area.

Noise in measurements. As mentioned in subsection 7.1, we observed frequent fluctuations in CPU inference and memory usage metrics when rerunning our experiments. These variations are caused by background processes running concurrently on the same system, which we could not completely shut off. We have taken steps to reduce this threat, which involves conducting the tests with minimal system activity, testing all models in the same run, and inducing manual garbage collection. However, the noise is still present and can affect the overall reliability and reproducibility of our measurements, therefore readers should consider these factors when interpreting or reproducing our results.

8 Conclusions and Future Work

In this paper, we explored the effect of group lasso pruning and dynamic post-training quantization on a code generation model. Using pruning at 60% target sparsity, we successfully reduced the model's size by 48% with a mild drop in accuracy, which is a good outcome considering the difficulty of this task. We did not find any speedups by applying post-training quantization. Using the optimizations offered by the ONNX runtime we were able to achieve a twice increase in inference on a regular laptop. The results are impressive since the smaller-sized model makes it faster to use, reduces its carbon footprint, and makes it more accessible to the everyday developer. Going forward, future work could explore more complex models, longer fine-tuning periods, the use of more mature compression libraries, and the exploration of more advanced compression techniques than the ones explored in this paper.

9 Responsible Research

In this section, we reflect on the ethical aspects of our research and the reproducibility of our methods. We conducted this research within the guidelines provided by the Netherlands Code of Conduct for Research Integrity [32].

Data. Our research involved the use of the PY150 dataset [22] consisting of Python code, for training and evaluation of our models. This dataset is part of the CodeXGlue benchmark [22], a scientific peer-reviewed work that has been used by many others in the field. Furthermore, strings and numbers have been removed from this dataset, to respect user privacy. Therefore, the data was collected ethically and responsibly. There are no exclusions that we performed on it (that somehow would yield better results), and this can be checked in the code and by reproducing our results.

Reproducibility. We have made efforts to ensure that our experiments can be replicated by other researchers. We have provided detailed descriptions of our methods and have made our code available for review. Moreover, we used seeds in pseudo-number generators, which guarantees that running our code will produce identical results.

Honesty and Independence. Firstly, we reported our entire research process in the most detailed manner we could come up with and presented all the setbacks we stumbled upon. Everything is presented thoroughly, from the preliminaries down to our results. Secondly, our research was guided by the pursuit of knowledge and understanding, not by a commercial or any other personal interest. Surely, finishing a Bachelor's degree as soon as possible would make a plausible reason for one to influence their results, however, this is not the case here. We openly acknowledge that our quantization results are not the greatest out there, and we invite everyone to reproduce our work to confirm they are real.

Broader implications. We acknowledge the broader implications of our work. We thoroughly considered the impact on the environment and the everyday user: compressing LLMs will help make them better for the environment and more accessible to the public, especially programmers.

The use of language models. ChatGPT [26] was used in this research to speed up the writing process. We would like to point out that all our research and information presented in this work is based on the state of the art techniques for compression and we have appropriately cited all relevant studies. We came up with our own ideas, with ChatGPT being solely used for rephrasing sentences to a more academic and formal style, and asking for directions and advice for academic writing. We have never used raw outputs but instead cleaned them up and corrected them to our liking. Examples of the prompts used are provided in Appendix A.

Acknowledgements

We thank Dr. Maliheh Izadi, ir. Ali Al-Kaswan and Prof. Dr. Arie van Deursen for their help and guidance. Their assistance and invaluable feedback greatly enriched this research, and we are truly grateful for the support.

A Appendix

A.1 ChatGPT prompts used to write this paper

Most prompts we used relate to rephrasing, asking for help to generate tables and references, and putting information together in a paragraph. We only provide a few examples, because the rest of the used prompts are similar and repetitive, like: "Please rephrase the following: ...", or "Please put the following discussion points in a paragraph ...". See Figures 5, 6, 7, 8 for prompt examples.

References

[1] Maximiliana Behnke and Kenneth Heafield. Pruning neural machine translation for speed using group lasso. In *Proceedings of the Sixth Conference on Machine Translation*, pages 1074–1086, Online, November 2021. Association for Computational Linguistics.

[2] Yelysei Bondarenko, Markus Nagel, and Tijmen Blankevoort. Understanding and Overcoming the Challenges of Efficient Transformer Quantization. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 7947–7969, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics.

[3] Tianyi Chen, Bo Ji, Tianyu DING, Biyi Fang, Guanyi Wang, Zhihui Zhu, Luming Liang, Yixin Shi, Sheng Yi, and Xiao Tu. Only train once: A one-shot neural network training and pruning framework. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, 2021.

[4] Xiaohan Chen, Yu Cheng, Shuohang Wang, Zhe Gan, Zhangyang Wang, and Jingjing Liu. Earlybert: Efficient bert training via early-bird lottery tickets, 2021.

[5] Aral D de Moor. Codegpt on xtc, 2023.

[6] Delft High Performance Computing Centre (DHPC). DelftBlue Supercomputer (Phase 1). https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase1, 2022.

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[8] Angela Fan, Edouard Grave, and Armand Joulin. Reducing transformer depth on demand with structured dropout. *arXiv preprint arXiv:1909.11556*, 2019.

[9] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding, 2016.

[10] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network, 2015.

[11] Mark Horowitz. 1.1 computing's energy problem (and what we can do about it). volume 57, pages 10–14, 02 2014.

[12] Lu Hou, Zhiqi Huang, Lifeng Shang, Xin Jiang, Xiao Chen, and Qun Liu. Dynabert: Dynamic bert with adaptive width and depth, 2020.

[13] Ting Hu, Christoph Meinel, and Haojin Yang. Empirical Evaluation of Post-Training Quantization Methods for Language Tasks, October 2022. arXiv:2210.16621 [cs].

[14] Intel. Intel extension for transformers. https://github.com/intel/intel-extension-for-transformers, 2023. Accessed: 13-06-2023.

[15] Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. TinyBERT: Distilling BERT for Natural Language Understanding. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 4163–4174, Online, November 2020. Association for Computational Linguistics.

[16] Kelvin. Model compression via pruning. https://towardsdatascience.com/model-compression-via-pruning-ac9b730a7c7b, 2020. Accessed: 2023-06-05.

[17] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper, 2018.

D What do the research proposal and the findings mean in the introduction section of a paper?

In the introduction section of a research paper, the "research proposal" and "findings" refer to two critical aspects of the study:

1. **Research Proposal:** This typically refers to a brief summary of what the research will investigate. It outlines the research question or problem statement that the paper seeks to answer or resolve. It also describes the methodology the researchers plan to use to investigate this question and why this particular approach was chosen. The research proposal is usually discussed in the introduction to give readers an overview of what to expect in the paper.
2. **Findings:** This is a brief summary of the results or conclusions of the research. It's unusual to see detailed findings in the introduction, as they're typically discussed more fully in the results or discussion sections. However, in the introduction, the authors might give a brief overview of the main results to provide readers with a sense of what the research uncovered. This is particularly common in formats like abstracts, where space is limited.

It's important to note that the structure of a research paper can vary based on the field of study, the specific journal or conference, or the preferences of the authors or reviewers. Not all papers will include detailed proposals and findings in the introduction, and some might include additional elements such as a literature review or theoretical framework.

Figure 5: Asking ChatGPT about the findings and research proposal in the introduction section.

D Give me a bibtex reference to the following website: https://github.com/intel/intel-extension-for-transformers

Finished browsing

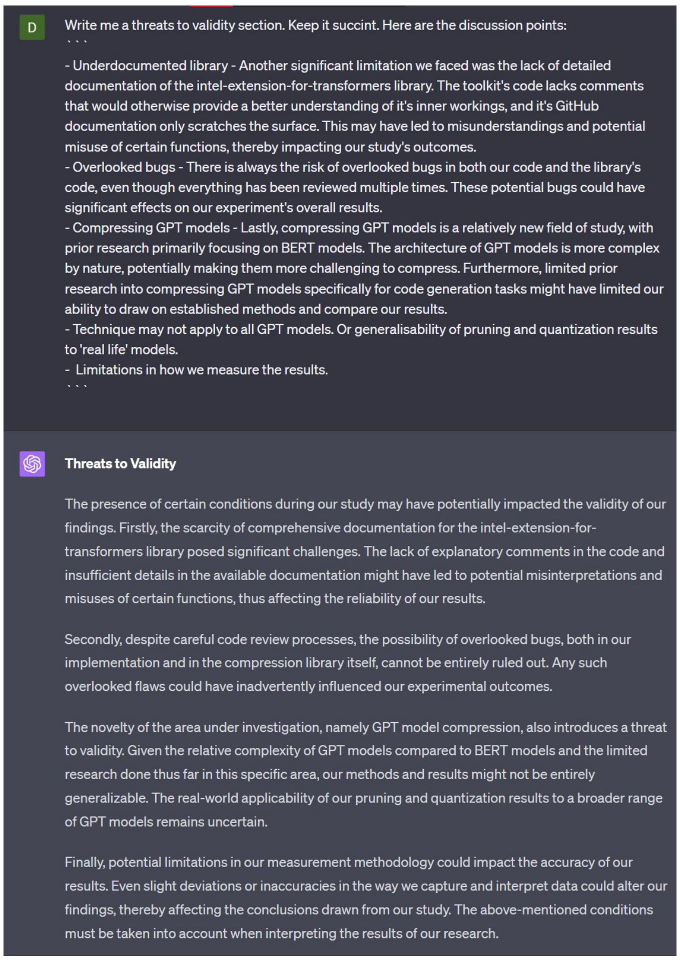
Here is the BibTeX citation for the Intel Extension for Transformers on GitHub:

bibtex Copy code

```
@misc{IntelExtensionForTransformers,
  author = {Intel},
  title = {Intel Extension for Transformers},
  year = {2023},
  note = {GitHub repository},
  howpublished = {\url{https://github.com/intel/intel-extension-for-transfor
  commit = {8671d14},
  accessed = {June 13, 2023}
}
```

This citation includes the repository's URL, the commit ID of the latest update (as of June 13, 2023), and the date you accessed it[1].

Figure 6: Asking ChatGPT to generate a reference of a website.



D

Write me a threats to validity section. Keep it succint. Here are the discussion points:

```
- Underdocumented library - Another significant limitation we faced was the lack of detailed documentation of the intel-extension-for-transformers library. The toolkit's code lacks comments that would otherwise provide a better understanding of it's inner workings, and it's GitHub documentation only scratches the surface. This may have led to misunderstandings and potential misuse of certain functions, thereby impacting our study's outcomes.
- Overlooked bugs - There is always the risk of overlooked bugs in both our code and the library's code, even though everything has been reviewed multiple times. These potential bugs could have significant effects on our experiment's overall results.
- Compressing GPT models - Lastly, compressing GPT models is a relatively new field of study, with prior research primarily focusing on BERT models. The architecture of GPT models is more complex by nature, potentially making them more challenging to compress. Furthermore, limited prior research into compressing GPT models specifically for code generation tasks might have limited our ability to draw on established methods and compare our results.
- Technique may not apply to all GPT models. Or generalisability of pruning and quantization results to 'real life' models.
- Limitations in how we measure the results.
```

Threats to Validity

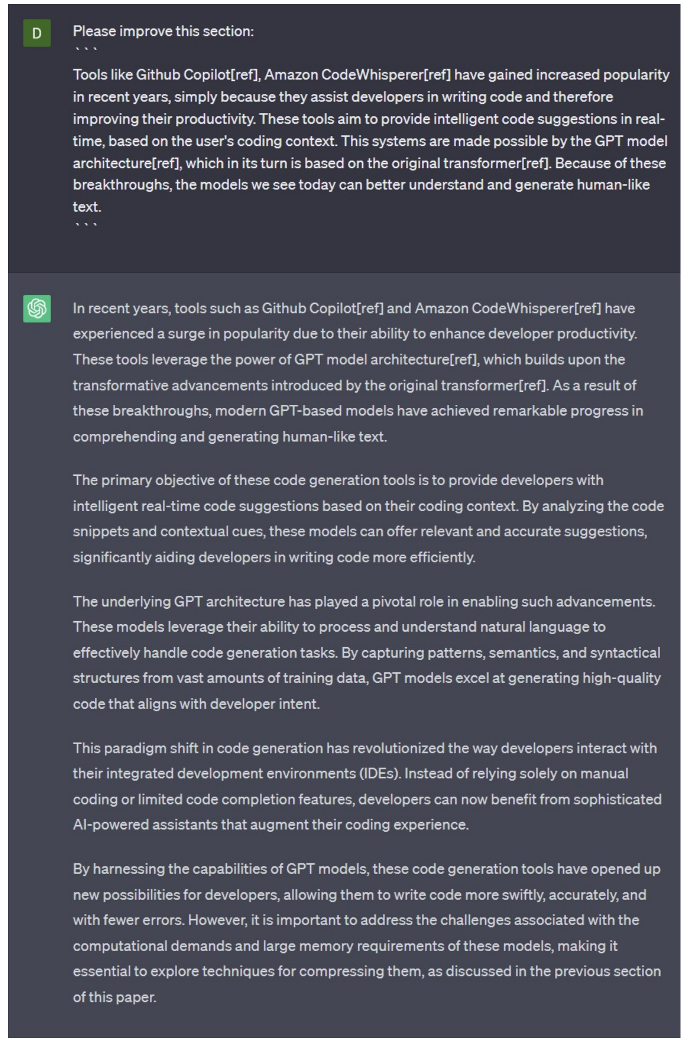
The presence of certain conditions during our study may have potentially impacted the validity of our findings. Firstly, the scarcity of comprehensive documentation for the intel-extension-for-transformers library posed significant challenges. The lack of explanatory comments in the code and insufficient details in the available documentation might have led to potential misinterpretations and misuses of certain functions, thus affecting the reliability of our results.

Secondly, despite careful code review processes, the possibility of overlooked bugs, both in our implementation and in the compression library itself, cannot be entirely ruled out. Any such overlooked flaws could have inadvertently influenced our experimental outcomes.

The novelty of the area under investigation, namely GPT model compression, also introduces a threat to validity. Given the relative complexity of GPT models compared to BERT models and the limited research done thus far in this specific area, our methods and results might not be entirely generalizable. The real-world applicability of our pruning and quantization results to a broader range of GPT models remains uncertain.

Finally, potential limitations in our measurement methodology could impact the accuracy of our results. Even slight deviations or inaccuracies in the way we capture and interpret data could alter our findings, thereby affecting the conclusions drawn from our study. The above-mentioned conditions must be taken into account when interpreting the results of our research.

Figure 7: Asking ChatGPT to write a section using the provided discussion points.



D

Please improve this section:

```
Tools like Github Copilot[ref], Amazon CodeWhisperer[ref] have gained increased popularity in recent years, simply because they assist developers in writing code and therefore improving their productivity. These tools aim to provide intelligent code suggestions in real-time, based on the user's coding context. This systems are made possible by the GPT model architecture[ref], which in its turn is based on the original transformer[ref]. Because of these breakthroughs, the models we see today can better understand and generate human-like text.
```

In recent years, tools such as Github Copilot[ref] and Amazon CodeWhisperer[ref] have experienced a surge in popularity due to their ability to enhance developer productivity. These tools leverage the power of GPT model architecture[ref], which builds upon the transformative advancements introduced by the original transformer[ref]. As a result of these breakthroughs, modern GPT-based models have achieved remarkable progress in comprehending and generating human-like text.

The primary objective of these code generation tools is to provide developers with intelligent real-time code suggestions based on their coding context. By analyzing the code snippets and contextual cues, these models can offer relevant and accurate suggestions, significantly aiding developers in writing code more efficiently.

The underlying GPT architecture has played a pivotal role in enabling such advancements. These models leverage their ability to process and understand natural language to effectively handle code generation tasks. By capturing patterns, semantics, and syntactical structures from vast amounts of training data, GPT models excel at generating high-quality code that aligns with developer intent.

This paradigm shift in code generation has revolutionized the way developers interact with their integrated development environments (IDEs). Instead of relying solely on manual coding or limited code completion features, developers can now benefit from sophisticated AI-powered assistants that augment their coding experience.

By harnessing the capabilities of GPT models, these code generation tools have opened up new possibilities for developers, allowing them to write code more swiftly, accurately, and with fewer errors. However, it is important to address the challenges associated with the computational demands and large memory requirements of these models, making it essential to explore techniques for compressing them, as discussed in the previous section of this paper.

Figure 8: Asking ChatGPT to improve a section.

[18] Eldar Kurtic, Daniel Campos, Tuan Nguyen, Elias Frantar, Mark Kurtz, Benjamin Fineran, Michael Goin, and Dan Alistarh. The Optimal BERT Surgeon: Scalable and Accurate Second-Order Pruning for Large Language Models, October 2022. arXiv:2203.07259 [cs].

[19] Woosuk Kwon, Sehoon Kim, Michael W. Mahoney, Joseph Hassoun, Kurt Keutzer, and Amir Gholami. A fast post-training pruning framework for transformers, 2022.

[20] Woosuk Kwon, Sehoon Kim, Michael W. Mahoney, Joseph Hassoun, Kurt Keutzer, and Amir Gholami. A Fast Post-Training Pruning Framework for Transformers. October 2022.

[21] Tianda Li, Yassir El Mesbahi, Ivan Kobyzev, Ahmad Rashid, Atif Mahmud, Nithin Anchuri, Habib Hajimolahoseini, Yang Liu, and Mehdi Rezagholizadeh. A short study on compressing decoder-based language models, 2021.

[22] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation, 2021.

[23] Emil Malmsten. Distilling code-generation models for local use, 2023.

[24] J. S. McCarley, Rishav Chakravarti, and Avirup Sil. Structured pruning of a bert-based question answering model, 2021.

[25] Paul Michel, Omer Levy, and Graham Neubig. Are sixteen heads really better than one?, 2019.

[26] OpenAI. Chatgpt (may 24 version). https://chat.openai.com/chat, 2023. Large language model.

[27] Minseop Park, Jaeseong You, Markus Nagel, and Simyung Chang. Quadapter: Adapter for gpt-2 quantization, 2022.

[28] David Patterson, Joseph Gonzalez, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. Carbon emissions and large neural network training, 2021.

[29] Sai Prasanna, Anna Rogers, and Anna Rumshisky. When bert plays the lottery, all tickets are winning, 2020.

[30] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

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

[32] Royal Netherlands Academy of Arts and Sciences (KNAW), Netherlands Federation of University Medical Centres (NFU), Netherlands Organisation for Scientific Research (NWO), Associated Applied Research Institutes (TO2), Netherlands Association of Universities of Applied Sciences (VH), Association of Universities in the Netherlands (VSNU). Netherlands code of conduct for research integrity. https://https://www.nwo.nl/en/netherlands-code-conduct-research-integrity, 2018. Accessed: 13-06-2023.

[33] Hassan Sajjad, Fahim Dalvi, Nadir Durrani, and Preslav Nakov. On the effect of dropping layers of pre-trained transformer models, 2022.

[34] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. October 2019.

[35] Haihao Shen, Ofir Zafrir, Bo Dong, Hengyu Meng, Xinyu Ye, Zhe Wang, Yi Ding, Hanwen Chang, Guy Boudoukh, and Moshe Wasserblat. Fast DistilBERT on CPUs, December 2022. arXiv:2211.07715 [cs].

[36] Sam Shleifer and Alexander M. Rush. Pre-trained Summarization Distillation, October 2020. arXiv:2010.13002 [cs].

[37] Mauro Storti. Leveraging efficient transformer quantization for codegpt: A post-training analysis, 2023.

[38] Chaofan Tao, Lu Hou, Wei Zhang, Lifeng Shang, Xin Jiang, Qun Liu, Ping Luo, and Ngai Wong. Compression of generative pre-trained language models via quantization, 2022.

[39] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[40] Elena Voita, David Talbot, Fedor Moiseev, Rico Sennrich, and Ivan Titov. Analyzing multi-head self-attention: Specialized heads do the heavy lifting, the rest can be pruned, 2019.

[41] Ziheng Wang, Jeremy Wohlwend, and Tao Lei. Structured Pruning of Large Language Models. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6151–6162, 2020. arXiv:1910.04732 [cs, stat].

[42] Xiaokai Wei, Sujan Gonugondla, Wasi Ahmad, Shiqi Wang, Baishakhi Ray, Haifeng Qian, Xiaopeng Li, Varun Kumar, Zijian Wang, Yuchen Tian, Qing Sun, Ben Athiwaratkun, Mingyue Shang, Murali Krishna Ramanathan, Parminder Bhatia, and Bing Xiang. Greener yet Powerful: Taming Large Code Generation Models with Quantization, March 2023. arXiv:2303.05378 [cs].

[43] Xiaoxia Wu, Zhewei Yao, Minjia Zhang, Conglong Li, and Yuxiong He. Extreme compression for pre-trained transformers made simple and efficient, 2022.

[44] Zhewei Yao, Reza Yazdani Aminabadi, Minjia Zhang, Xiaoxia Wu, Conglong Li, and Yuxiong He. ZeroQuant:

Efficient and Affordable Post-Training Quantization for Large-Scale Transformers. October 2022.

[45] Ofir Zafrir, Ariel Larey, Guy Boudoukh, Haihao Shen, and Moshe Wasserblat. Prune Once for All: Sparse Pre-Trained Language Models, November 2021. arXiv:2111.05754 [cs].

[46] Michael Zhu and Suyog Gupta. To prune, or not to prune: exploring the efficacy of pruning for model compression, 2017.