Leveraging Efficient Transformer Quantization for CodeGPT: A Post-Training Analysis

Mauro Storti

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: Mauro Storti
Final project course: CSE3000 Research Project
Thesis committee: Prof. Dr. Arie van Deursen, Dr. Maliheh Izadi, ir. Ali Al-Kaswan, Avishek Anand

An electronic version of this thesis is available at https://github.com/AISE-TUDelft/LLM4CodeCompression

Throughout this paper, ChatGPT, Grammarly, and Writefull were used to aid with grammar and rephrasing for clarity.

Abstract

The significant advancements in large language models have enabled their use in various applications, such as in code auto-completion. However, the deployment of such models often encounters challenges due to their large size and prohibitive running costs. In this research, we investigate the effectiveness of post-training quantization techniques in compressing a CodeGPT model, specifically using the "Per-embedding-group" and "Mixed precision" post-training quantization methods. Our evaluation is done on the code completion task of the CodeXGLUE benchmark using the Edit Similarity and Exact Match metrics, offering a comprehensive understanding of the impact of post-training quantization on the accuracy of the model. We also compare our results with three other compression approaches for the same model. From our analysis, we find that CodeGPT is very resilient to quantization noise, allowing the model to be compressed by 4 times its size with negligible accuracy loss. Furthermore, post-training quantization seems to be the best option for compressing the CodeGPT model when accuracy is a priority. Our work only simulates post-training quantization to draw conclusions on its performance on accuracy, future work should analyze the inference speed and memory use at runtime on such a post-trained quantized model.

1 Introduction

Large Language Models (LLMs) have emerged as a pivotal advancement in the field of machine learning due to their superior ability to understand and generate natural language content [2]. One of the most notable applications of LLMs lies in the realm of software engineering, where they are used for code auto-completion [9]. Code auto-completion tools provide context-aware suggestions to developers to help them code more efficiently by providing relevant suggestions based on their coding context.

Despite their many capabilities, the deployment of LLMs is often hindered by their large size, which has been steadily growing [21]. This large size brings several obstacles. The models have prohibitive running costs which limit their application in resource-constrained devices. Additionally, accessing these models via online services poses both practical and ethical challenges. For instance, users will need a WiFi connection with low latency to ensure an enjoyable user experience, and there is also the threat of sensitive queries being stored by the service provider. These challenges show the need for efficient model compression techniques to optimize the deployment of LLMs, particularly for code auto-completion.

In this research, we strive to investigate the question, *"How effective are post-training quantization techniques (PTQ) for compressing a CodeGPT generation model?"*. This question is of considerable importance, given that this compression technique family is the least reliant on the training data of the original model and requires little re-training time. In particular, we focus on replicating mixed-precision and per-embedding-group PTQ techniques [1], given their promising results on a different LLM architecture.

The effectiveness of these techniques is evaluated based on the code completion task of the CodeXGLUE benchmark [6], using the edit similarity and exact match metrics. We also compare the accuracy and compression rate of our techniques to the findings of de Moor [3], Malmsten [7], and Sochirca [15], which apply different compression approaches to the same model we are analyzing.

The main contributions of our study can thus be summarized as follows:

- An investigation on how naive quantization of CodeGPT's weights and activations affect its accuracy on the code completion task.
- An analysis of the performance of different PTQ Range Estimators on the code completion task of CodeXGLUE.
- An adaptation of the "Per-embedding-group Post Training Quantization" and "Mixed precision Post Training Quantization" [1], to work with CodeGPT and an analysis of their performance.
- A comparison of the compression rate and accuracy of our most promising results with other viable compression techniques.

Ultimately, our research contributes to the broader aim of enhancing the accessibility and efficiency of LLMs, especially for code auto-completion tasks, thereby increasing developer productivity and improving software quality.

2 Background

In this section, we give some entry knowledge on how LLMs work, their challenges, and some possible solutions. Most LLMs are equipped with "transformers", attention-based models that weigh the significance of different words in a text while handling a task [16]. Unlike traditional sequential models, transformers can process all words or symbols in the sentence at once, making them highly effective for many language tasks [16].

Distinctive functionalities characterize various LLMs. For instance, Bidirectional Encoder Representations from Transformers (BERT) employs a bidirectional transformer architecture, capturing context from both the left and right sides of a sentence [4]. On the other hand, Generative Pre-trained Transformer 2 (GPT-2), a unidirectional LLM, learns by scanning the input from left to right and trying to predict the rest. Thanks to this architecture, GPT models can generate coherent text based on a given prompt [8].

In recent years, the application of LLMs has extended into the realm of code generation. CodeGPT, for example, is a GPT-2 style model pre-trained on programming language, to support code-completion and text-to-code generation tasks [6].

However, the broad range of capabilities offered by LLMs is accompanied by substantial challenges. Their size and consequential computational demands limit their application,

particularly when deployed on resource-restricted devices [21]. To navigate this challenge, research is being focused on the development of efficient compression techniques. These techniques are designed to shrink the size of LLMs by pruning redundant or less important parameters without undermining their performance.

A substantial body of work has been carried out to compress BERT models [1; 10; 13; 14; 18; 20; 22]. However, when it comes to code-oriented LLMs like CodeGPT, exploration has been limited. This opens a crucial research gap that calls for investigating the efficacy of compression techniques on CodeGPT models.

2.1 Transformers

At the heart of several LLMs lies the "transformer", a novel type of neural network architecture which is highly parallelizable [16]. The power of transformers lies in their ability to capture the complexity and nuances of sequential data, like natural or programming language, making them integral to the success of LLMs.

2.2 Embeddings

Embeddings map discrete categorical values (like words) to vectors of continuous numbers [5]. These vectors are designed to capture semantic relationships between the input values. For example, words with similar meanings should be close to each other in the embedding space. They are used at the start of the transformer pipeline to convert the input data to meaningfully represent categories in the transformed space [5].

2.3 Weights

Weights are used to represent the strength or influence that one node in the network has over another. During training, these weights are iteratively adjusted based on the error the model makes in its predictions. These adjusted weights encapsulate what the model has "learned" about the task it is performing.

2.4 Activations

Activation functions are mathematical equations that help determine the output of a neural network. They decide whether a given neuron should be activated or not based on the weighted sum of its input.

2.5 Attention mechanisms

The attention mechanism is central to the transformer architecture. It allows the model to focus on different parts of the input sequence when producing an output sequence. Attention scores determine the weight or significance that the model assigns to different parts of the input. This mechanism enables the model to capture long-range dependencies in the data, a feature particularly beneficial for natural language processing tasks.

2.6 Attention scores

In a transformer, each layer computes a set of query, key, and value vectors from the input data. Attention scores are computed by taking the dot product of the query and key vectors, followed by a softmax activation to obtain probabilities [16]. These scores are then used to weight the value vectors, resulting in the output of the attention layer [16].

3 Related Work

There are several techniques for compressing LLMs, but the three primary approaches are knowledge distillation, pruning, and quantization:

- **Knowledge distillation** involves training a smaller model to emulate the larger one. These models are referred to as student and teacher respectively [11; 12; 20; 22].
- **Pruning** involves removing unimportant parameters from the model based on some criterion, such as their magnitude or contribution to the overall loss [12; 18; 20]. This can be done during or after training.
- **Quantization** involves reducing the precision of the model's weights and activations, typically from 32-bit floating-point values to 8-bit integers, thereby diminishing the memory footprint and enhancing inference speed [1; 12; 14; 19; 20; 22].

3.1 The potential of quantization

While all three techniques have shown promising results for compressing LLMs, quantization has some advantages over knowledge distillation and pruning. Knowledge distillation requires training a new model from scratch, which can be computationally expensive and requires access to the original training data. Similarly, pruning requires either retraining the model or fine-tuning the pruned model, which can also be computationally expensive. In contrast, quantization can be applied to a pre-trained model without the need for additional training, making it a faster and more resource-efficient approach. Additionally, quantization can reduce the memory footprint of the model by a larger factor compared to pruning, which only removes a subset of parameters [19].

3.2 Quantization basics

As explained by Bodarenko et al. [1]: asymmetric quantization is a method of quantization that is frequently applied due to its ability to efficiently implement fixed-point arithmetic. This process is characterized by three parameters: **bit-width**, **scale factor**, and **zero point**. The quantization process transforms a real-valued tensor into an unsigned integer grid, which can then be approximated back to its original real value through a process known as de-quantization. In symmetric quantization, the quantization grid is adjusted to be symmetric around the zero point.

Quantization, however, can lead to the model performing worse due to noise being added to the network [1]. One way to mitigate this noise is by performing a process called **calibration** [19]. In this process, we use range estimators to determine the scale factor and zero point of the quantizer so that we can get a result as close as possible to the original non-quantized model [1; 19]. There are several range estimators, but in this work, we will only look at **current min-max**, **running min-max**, and **mean squared error** (MSE), which are further explained in the literature [1].

The quantization parameters are typically set for each tensor individually, a strategy called per-tensor quantization. Alternatively, you could enhance the quantization granularity by designating separate quantizers for different parts of a tensor. This approach could improve the accuracy of the network, but would also require additional computational resources and memory [1].

3.3 The different quantization approaches

Quantization techniques can be broadly classified into two categories, quantization-aware training (QAT) and post-training quantization (PTQ):

- **QAT** involves training the model with quantization in mind, which means that the model is trained to be more robust to quantization noise. This is because the learned parameters are already compatible with quantized values. QAT can be achieved by introducing noise into the weights and activations during training or by using a modified loss function that considers the quantization error. The advantage of QAT is that it can achieve higher accuracy compared to PTQ since the model is trained to be more quantization friendly. However, QAT requires retraining or fine-tuning the model, which can be computationally expensive and time-consuming. Additionally, QAT may require labeled data for the noise injection process, which may not be available in all cases [19; 22].
- **PTQ**, on the other hand, involves quantizing a pre-trained model without any additional training. This is done by applying quantization to the model's weights and activations after training. The advantage of PTQ is that it is a fast and efficient way to compress a pre-trained model. PTQ also does not require additional training or fine-tuning, making it easier to apply to pre-trained models. However, PTQ may not achieve the same level of accuracy as QAT since the model was not trained with quantization in mind. Additionally, PTQ may require calibration to adjust the quantization parameters for optimal accuracy, which can be challenging to do in practice [19; 22].

3.4 The challenges of post-training quantization

In LLMs, weights and activations are the key components that determine the model's behavior. When applying PTQ to LLMs, the accuracy loss can be analyzed by looking at the effects of quantization on both activations and weights. For both the BERT and GPT-3 models, research papers have shown that the worse performance with PTQ is due to different tokens having dramatically different activation ranges [1; 22]. Specifically, for BERT models, some tensors are more susceptible to quantization than others, especially for deeper encoder layers [1]. Similarly, for GPT-3 models, there is a 10x difference between the largest magnitudes of different rows, leading to a worse generation performance of the INT8 weight PTQ [22].

3.5 Solving PTQ issues

To solve the accuracy issues of PTQ, several methods have been proposed:

- **Per-embedding-group quantization**: the activation tensors are split into evenly sized groups based on their activation ranges. Quantization is then applied with different parameters for each group, thus improving accuracy [1].
- **Mixed precision PTQ**: a combination of low-precision and high-precision data types is used to represent weights and activations during quantization. This method can reduce the accuracy loss while still achieving high compression rates [1].
- **Group-wise quantization for weights**: addresses the accuracy issue by grouping the weight matrices into smaller groups and applying quantization to each group separately. This allows for more fine-grained control over the quantization process and can improve accuracy [22].
- **Token-wise quantization for activations**: Similar to per-embedding-group quantization, token-wise quantization improves accuracy by applying quantization with different parameters. It differs however by applying different quantizations for each singular token, rather than for groups [22].

Overall, these techniques show promising results for compressing LLMs while mitigating accuracy loss.

3.6 Competing Approaches

The results of our compression methods will be compared with those of three other papers [3; 7; 15], which also adapted existing BERT compression techniques to codeGPT:

- **De Moor** [3] utilizes techniques from Wu et al. [20] to evaluate hybrid in-training knowledge distillation, layer reduction, and quantization techniques. They accomplish this by retraining the baseline CodeGPT teacher and a compressed student model for the Code Completion fine-tuning assignment. The resulting 6-layer model has a 15x compression thanks to its 1-bit weight and 8-bit activation quantization.
- **Malmsten** [7] utilizes techniques from Sanh et al. [11] to evaluate in-training knowledge distillation techniques. The model applies a three-part loss function, to draw knowledge from the teacher model. The most significant outcomes involved two models, one with one-third of the layers of the standard model, and another retaining two-thirds.
- **Sochirca** [15] utilizes techniques from Shen et al. [13] to evaluate hybrid post-training pruning and quantization techniques on CPUs. Sochirca's research utilizes the Intel-extension-for-transformers toolkit to trim and quantize the CodeGPT model after training. Their method consists of applying group lasso structured pruning at 60% sparsity. This means that instead of individual nodes, whole groups of neurons with unimportant weights are pruned together. Their best model is half the size of the original.

4 Methodology

Our methodology consists of implementing, comparing, and evaluating different PTQ methods. The results will help us determine if CodeGPT is a good candidate for quantization and give us a broader understanding of GPT architectures' ability to be compressed.

4.1 Baseline model

Our quantization methods are applied to a CodeGPT model finetuned on an auto-completion task. CodeGPT is part of the GPT family and is optimized for understanding and generating code.

4.2 Naive PTQ

To evaluate the resilience of CodeGPT to the performance degradation due to PTQ, we implement naive quantization on both weights and activations. Naive means that no optimization methods are used, but only simple per-tensor symmetric or asymmetric quantization on every weight and activation. Naive PTQ is the easiest to implement and is supported by most hardware.

4.3 Optimized quantization methods

Adapting from the techniques developed by Bondarenko et al. [1], we implemented the proposed solutions for a GPT-like model as opposed to the original BERT-based approach. In particular, we focused on two main methods: mixed-precision and per-embedding-group PTQ.

Mixed-precision (MP) PTQ utilizes a blend of low-precision and high-precision data types for representing weights and activations during quantization, which aids in reducing accuracy loss while maintaining high compression rates. Unfortunately, MP PTQ is not supported by all hardware.

Per-embedding-group (PEG) PTQ instead applies quantization to separate groups of embeddings based on their usage patterns, thereby acknowledging and accounting for the disparate activation ranges of different tokens: the activation tensor is split into K evenly-sized groups that share quantization parameters among elements in the same group. Tensors can be permuted along the embedding dimension so that outliers can end up in the same embedding dimension. This comes at the cost of $d + 2 \cdot 3 \cdot K$ extra parameters per attention layer where d is the number of embedding dimensions [1]. This cost is however negligible for CodeGPT given that $d = 1024$, it has 12 attention layers and 124.24 Million total parameters, meaning that for $K = 10$ the model increases its size by 0.01%. PEG is not natively supported by all hardware but can be simulated on those that support per-tensor operation following the procedure suggested in the literature [1].

Because we could not find a replication package we do not analyze the group-wise quantization for weight and token-wise quantization for activations techniques of Yao et al. [22]. However, we do test the performance of the PTQ per-token offered in the Bondarenko et al. replication package [1].

4.4 Comparison with other compression techniques

Finally, our methodology also encompasses a comparative study in which the performance of the implemented methods (PEG and MP PTQ) is benchmarked against other prevailing compression techniques. This comparison serves to highlight the relative strengths and weaknesses of each technique, thereby providing a comprehensive understanding of their applicability to LLMs and the benefits they offer in terms of model compression and accuracy.

5 Experimental setup

This section outlines our experimental setup and provides details on replicability, our computational infrastructure, model training, experimental approach, and quantization techniques.

5.1 Replicability

All tests conducted in our experiments can be replicated using the code provided at our GitHub repository[1]. This repository contains all necessary information and commands to rerun our experiments, including seeds for our samples and an environment file specifying all versioned libraries used.

5.2 Computational infrastructure

We conduct our experiments on the Google Colab platform[2], using a CUDA-enabled V100 graphic card and the "Runtime Shape" option set to "High-RAM".

De Moor [3], Malmsten [7], and Sochirca [15] perform their experiments on the DelftBlue supercomputer, which is also equipped with a CUDA-enabled V100 graphic card, thus the results are comparable.

5.3 Model training

We train the CodeGPT model[3] on the PY150 code completion task of the CodeXGLUE benchmark[4]. PY150 is a Python language dataset, using the CodeGPT framework. The model is trained for only one epoch and is available on HuggingFace[5]. We use this model as our baseline for our tests, as well as for quantization.

De Moor [3], Malmsten [7], and Sochirca [15] use a different baseline model, also available on Hugging Face[6]. We choose not to use this model because of the lack of a tokenizer, which is necessary for our quantization. The model they use is trained identically to ours, but has slightly lower baseline scores, probably due to training on different machines. We make our comparisons with this difference in mind.

[1] https://github.com/AISE-TUDelft/LLM4CodeCompression

[2] https://colab.research.google.com/

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

[4] https://huggingface.co/datasets/0n1xus/codexglue

[5] https://huggingface.co/AISE-TUDelft/BRP-Storti-CodeGPT-Py150

[6] https://huggingface.co/AISE-TUDelft/CodeGPT-Py150

5.4 Evaluation metrics

For the evaluation phase, we utilize the CodeXGLUE code benchmark. The effectiveness of our model and the quantization methods implemented are measured using two key metrics: Exact Match (EM) and Edit Similarity (ES). The Exact Match metric measures the number of times the model's output matches the expected output exactly, while the Edit Similarity metric quantifies the structural and semantic similarity between the model's output and the expected output.

5.5 Experimental Approach

Our experimental methodology closely aligns with the procedures laid out by Bondarenko et al. [1]. We incorporate uniform affine quantization which means using symmetric weights and asymmetric activations, coupled with a static activation range configuration. Quantization is applied identically to the weights and activations across all layers of the model.

For the **Range Estimators** investigation we use the following options: mean squared error (MSE), current min-max (CMM), and running min-max (RMM). We conduct an 8-bit post-training quantization on both weights and activations, as outlined in our methodology. We then use the best combination of range estimators to understand the effect of using different bit-widths for all weights and all activations in **Naive quantization**.

For the **MP PTQ** approach, we start with 8 bits for all activations, except for one which we kept to 16 bits. This strategy allowed us to examine which activation in the layers' block is the most susceptible to quantization. We also explore the effect of keeping multiple parts of the feed-forward network (FFN) to 16 bits, as suggested in the literature [1] for BERT.

We also evaluate the efficacy of **PEG PTQ**, which hinges on the number of groups K. We perform these tests with both permutation and no permutation of the columns. We also test if accuracy is regained on 4 bits weights using PEG. Finally, we test the per-tensor quantization method.

6 Results

In this section, we present the results of our research. In all tables, the nomenclature WxAy is used, where W stands for weights and A stands for activations, while x and y are integers representing the respective number of bits used for their quantization. For example, W8A16 means we are using 8 bits for the quantization of all the weights and 16 for the quantization of all the activations. Furthermore, we bold the rows in the table, outside of the baselines, that show the best accuracy results. We do so to increase the readability of the experiments.

6.1 Range estimators

Referring to Table 1, it is evident that the activation range does not significantly influence the Code Completion task. A noticeable performance improvement was only recorded when MSE was used for weight range estimation. Thus, for all subsequent experiments, we utilized MSE for weight range estimation and the current min-max method as range estimators.

Weights	Activations	ES	EM
None	None	41	17
MSE	**CMM**	**38**	**16**
MSE	**RMM**	**38**	**16**
CMM	CMM	38	14
RMM	CMM	38	14

Table 1: Range estimation experiments

Method	ES	EM	Compression
Baseline	41	17	1x
W8A8	38	16	4x
W8A32	**40**	**17**	**4x**
W8A16	**40**	**17**	**4x**
W4A32	36	14	8x
W4A16	36	14	8x

Table 2: Naive PTQ experiments

6.2 Naive quantization

For naive quantization, we investigate the effect of using different bit-widths for the weights and activations. As we can see in Table 2, we have a high drop in accuracy when we quantize the activations to less than 16 bits or the weights to less than 8 bits.

6.3 Mixed precision quantization

In Table 3 we can see that by far the biggest performance increase occurs when we allow the activation of the residual sum of the block to be 16 bits. Keeping multiple parts of the FFN to 16 bits does not lead to significant performance increases compared to keeping only the residual sum. The nomenclature used is explained in Appendix A.

6.4 Per-embedding-group quantization

We can see in Table 4 that the best performance is achieved with $K = 8$ if we do not permute the groups or $K = 4$ if we do. There is no performance increase on the quantization of 4 bits with the PEG permute technique. We also see that, as expected, the per-tensor quantization performs really well.

6.5 Other approaches results

In Table 5 we can see that up to 4x compression, our methods have greater compression while also having the smallest accuracy loss compared to the baseline. De Moor achieves a compression of 16x for an ES loss of 5.18 and EM loss of 3.6, which we can compare only to our 4-bit weight quantization as we did not test for 1-bit weight quantization.

7 Discussion of the Results

PTQ has proven to be very suited to be used on CodeGPT, achieving compression rates of up to 4 times with a loss of accuracy of only 3 points for the ES and 1 on the EM metrics with 8-bit quantizations and activations. In particular, CodeGPT loses most of its accuracy once the weights are

Method	ES	EM
Baseline	41	17
Naive W8A8	38	16
Naive W8A16	40	17
First Layer Normalization (Input FFN)	37	15
Second Layer Normalization	37	16
Residual Sum of attn	38	16
Residual Sum of block	**40**	**17**
Query $\times$ Key	37	16
Softmax	37	15
weights_output $\times$ value	38	16
Attention projection layer	37	16
MLP fully connected layer	37	16
MLP projection layer (Output FFN)	38	17
Input and Output of FFN	37	17
Input and Residual Sum of FFN	39	16
Output and Residual Sum of FFN	39	17

Table 3: Mixed precision PTQ experiments

Method	ES	EM
Baseline	41	17
Naive W8A8	38	16
Naive W4A32	38	16
W8A8 $K = 2$	38	16
W8A8 $K = 4$	38	17
W8A8 $K = 8$	**40**	**17**
W8A8 $K = 2$ permuted	39	17
W8A8 $K = 4$ **permuted**	**40**	**17**
W4A32 $K = 8$ permuted	36	14
W8A8 per-tensor	**40**	**17**

Table 4: Per-embedding-group PTQ experiments

Method	Compression	ES	EM
Our Baseline	-	41.0	17.0
MP PTQ	**4x**	**40.0**	**17.0**
PEG PTQ	**4x**	**40.0**	**17.0**
Naive W4A32	8x	38.0	16.0
Their Baseline	-	39.1	14.5
de Moor	15.8x	33.92	10.9
Malmsten 4 layers	1.8x	27.6	5.9
Malmsten 8 layers	1.3x	29.5	6.1
Sochirca	1.9x	30.5	9.0

Table 5: Comparison of our method to competing approaches

quantized to less than 8 bits or the activations to less than 16 bits. This property is shared with the BERT and GPT-3 models [1; 22].

The choice of range estimator for the quantization of the activations does not seem to influence the performance. It does seem instead that the best performances are achieved using MSE for weight quantization.

Mixed-precision PTQ seems to be an effective technique proved to regain most of the accuracy loss due to PTQ, with the best results achieved by quantizing the residual sum of the layers' block. MP PTQ, however, leads to a small computational overhead compared to naive quantization and is not natively supported by all hardware.

Per-group-embedding is the preferred technique, given that it can be simulated on all hardware that supports per-tensor operations and has negligible parameter increase. The best results are achieved by using 8 groups and no permutation, or 4 with permutation. This technique however does not seem able to recover the performance loss due to 4-bit weight quantization.

Per-tensor, quantization would also be an optimal choice if not for the overhead introduced by the additional operations between main memory and GPU [22]. Thus, not having tested the token-wise quantization for activations proposed by Yao et al. [22], we cannot make recommendations for or against it.

Compared to the techniques implemented by Malmsten [7], and Sochirca [15], our techniques seem to retain the most accuracy while also compressing the model more. To be noted is however the fact that Sochirca's work focuses on compressing the model on CPUs, while we did not have this restriction. De Moor's results [3] instead are promising due to the almost 16x compression, while retaining decent performances. Future work should test the abilities of CodeGPT to have weights quantized to only 1 bit.

Post-training quantization is thus an optimal choice for compressing CodeGPT on the AutoCompletion task. It achieves good performance on accuracy and compression of up to 4 times without the need for training data or model retraining.

7.1 Threats to validity

To test our techniques, we adapted the Bondarenko et al. [1] replication package, which was originally meant for BERT models, to our CodeGPT model. This brings several challenges that might affect the validity of our results which we will discuss. In this section, we analyze the internal, external, and construct threats to the validity of our paper.

Internal validity

The quantization is simulated by mapping floating number tensors to their quantized values, without changing the datatype of the tensor. For this reason, there are several possible points of failure where the values of the tensors could be inadvertently converted back to their original bit-width. During the testing phase, we logged the intermediate values to manually check if they were being converted properly, but this brings limited assurance. Still, the accuracy loss seems

to be similar to the one presented by Bondarenko et al. [1] on the BERT models and by Yao et al. [22] on the GPT-3 model.

The activation quantization was achieved by copying the GPT-2 code of the Hugging Face Transformers library[7] and adding quantization operations after each activation. This brings the threat that the code might be missing some activation and thus skewing the results of our tests by keeping some activation to 32 bits. This however seems unlikely and would probably not skew the results majorly since all the main activations are quantized.

External validity
All of our testing was conducted only on the PY150 dataset of CodeXGLUE. This dataset contains only Python code, meaning that the ES and EM loss might be greater when quantizing a CodeGPT model fine-tuned on code generation tasks with other programming languages. Furthermore, the PY150 has a training set of 100k rows for the AutoCompletion task, while PTQ might not be as effective at compressing models fine-tuned on smaller datasets. Dataset size and task, however, do not seem to be relevant to the performance of quantization since we see in Bondarenko et al. [1] that there are no dramatic performance changes between GLUE tasks, which have all different dataset sizes [17]. Further investigation is however advised for future work.

Construct validity
When making comparisons with the competing approaches, it must be noted that, unlike their models, our approach only simulates quantization. This is done to ensure that the techniques would not be influenced by the limitations of the hardware. This allowed us to investigate the full potential of each PTQ technique. Their methods instead create a new compressed model, meaning that the compression might not be as good as theoretically possible due to hardware limitations, which might skew results in favor of our technique. Furthermore, their techniques are also evaluated on CPU and GPU inference as well as size on GPU. Because of the limitations of the simulation, our PTQ methods were not experimentally evaluated on those metrics. We can, however, hypothesize that our GPU inference might increase greatly due to the fact that GPUs perform better with fixed point arithmetic and that the GPU size would be compressed greatly due to activation quantization [1].

8 Conclusion

In this paper we analyzed how effective post-training quantization techniques are at compressing a CodeGPT model. Through our analysis, we have shown that MSE is the best range estimator for weights quantization on the code completion task of CodeXGLUE. Our experimental results demonstrated that a naive PTQ implementation already achieves great results, with a compression rate of 4x and negligible accuracy loss.

Furthermore, we can virtually remove all accuracy loss with more advanced PTQ methods like mixed-precision, per-tensor, or per-embedding-group, the latter of which is supported by all hardware that enables per-tensor operations and presents minimal overhead in computation. Our findings further emphasized that, like BERT and GPT-3, CodeGPT is most susceptible to quantization of fewer than 8 bits for quantization and less than 16 bits for activations. We could not however recover performance loss for the 4-bit quantization of weights with the techniques we implemented.

It is important to note that our implementation solely simulated quantization, highlighting the need for future investigations to evaluate these techniques with tensor quantization on different hardware. Such analyses could further strengthen the claims we made on the performance of PTQ methods on CodeGPT and provide valuable insights into how much our methods speed up inference time and reduce GPU size during runtime.

9 Responsible Research

This section provides a reflection on the ethical aspects and reproducibility of our research. While this component is not always explicitly included in published works, we believe that it is critical to consider and report these aspects to maintain transparency and accountability in our research.

9.1 Ethical Considerations

We acknowledge that the code utilized in our research is not original; rather, it is an adaptation of the code provided in the replication package of "Understanding and Overcoming the Challenges of Efficient Transformer Quantization". The original authors of the code are credited accordingly.

Our research contributes to the body of knowledge not through the development of new techniques but rather through the application of existing methods to a different model and architecture. This includes an in-depth analysis of the results, providing new insight and understanding.

9.2 Reproducibility and Transparency

We have made significant efforts to ensure the reproducibility of our experiments. The code used in our experiments, as well as all necessary information and commands to rerun the experiments, are publicly available in our GitHub repository[8]. The repository also includes an environment file specifying all versioned libraries used, ensuring a consistent environment setup for anyone looking to replicate our work. Furthermore, the model was trained and tested from scratch, adhering to the same training arguments as the original model to maintain consistency.

9.3 Use of AI tools in this paper

While writing this paper, we have utilized technologies such as ChatGPT, Grammarly, and Writefull to assist us in the correct use of grammar, tone, and clarity. Examples of prompts used in ChatGPT are: "I have already used word x, can you rephrase this sentence to use another word for x, ..." or "Can you rephrase this sentence to use more formal language...". Because we do not trust ChatGPT to make statements that are backed by a source, ChatGPT was not used for the collection of any information and each output was sanitized to

[7] https://huggingface.co/docs/transformers/

[8] https://github.com/AISE-TUDelft/LLM4CodeCompression/tree/main/ETF

make sure no new unsubstantiated claims were added to the inputted text.

ChatGPT was also used to help with LaTeX, with queries such as: "Can you make me a table in Latex for a scientific paper with the following columns: Method, Weight Range, Activation Range, Edit Similarity, Exact Match", and "Can you make a .bib reference for this paper:...". These queries were used to speed up the writing process but were then manually checked for correctness.

References

[1] 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.

[2] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

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

[4] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[5] Will Koehrsen. Neural network embeddings explained: How deep learning can represent war and peace as a vector. https://towardsdatascience.com/neural-network-embeddings-explained-4d028e6f0526, 2018. Accessed: 2023-06-25.

[6] Yiming Lu, Meng Zhang, Yuncong Li, and Xiaodong Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 6725–6736, 2021.

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

[8] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. *https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf*, 2018.

[9] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 419–428, 2014.

[10] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of BERT: smaller, faster, cheaper and lighter. https://doi.org/10.48550/arXiv.1910.01108, 2019.

[11] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.

[12] Hao Shen, Ori Zafrir, Bo Dong, Haoran Meng, Xinyi Ye, Zhongyuan Wang, ..., and Moti Wasserblat. Fast distilbert on cpus. *arXiv preprint arXiv:2211.07715*, 2022.

[13] Hao Shen, Oriol Zafrir, Biao Dong, Haoyu Meng, Xianming Ye, Ziyang Wang, Yilong Ding, Hong Chang, Guy Boudoukh, and Mayer Wasserblat. Fast distilbert on CPUs. https://doi.org/10.48550/arXiv.2211.07715, 2022.

[14] Shuangyan Shen, Zhaoyang Dong, Junjie Ye, Li Ma, Zhiqing Yao, Amir Gholami, Michael W. Mahoney, and Kurt Keutzer. Q-BERT: Hessian based ultra low precision quantization of BERT. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 8815–8821, 2020.

[15] Dan Sochirca. Compressing code generation language models on cpus, 2023.

[16] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[17] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 353–355, Brussels, Belgium, 2018. Association for Computational Linguistics.

[18] Zihao 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.

[19] Xiaohua Wei, Srinivas Gonugondla, Wasi Uddin Ahmad, Shuhuai Wang, Baishakhi Ray, Hao Qian, Xiaodong Li, Vipul Kumar, Zhongyuan Wang, Yuan Tian, Qi Sun, Ben Athiwaratkun, Mingbo Shang, Murugan K Ramanathan, Parminder Bhatia, and Bing Xiang. Greener yet powerful: Taming large code generation models with quantization. *arXiv preprint arXiv:2303.05378*, 2023.

[20] Xiaohua Wu, Zhiqing Yao, Meng Zhang, Changqing Li, and Yuan He. Extreme compression for pre-trained transformers made simple and efficient. *arXiv preprint arXiv:2206.01859*, 2022.

[21] Fuxuan Xu, Uri Alon, Graham Neubig, and Vincent J Hellendoorn. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 1–10, June 2022.

[22] Zhewei Yao, Ramin Yaghoubzadeh Aminabadi, Muhao Zhang, Xianjing Wu, Chao Li, and Yuan He. Zeroquant: Efficient and affordable post-training quantization for large-scale transformers. In *Advances in Neural Information Processing Systems*, October 2022.

A Name of Activations

Table 6 outlines the nomenclature used to represent each activation in the model's blocks.

Name	Activation
1st Layer Normalization (Input FFN)	ln_1
2nd Layer Normalization	ln_2
Residual Sum of attn	res_act_quantizer_1
Residual Sum of block	res_act_quantizer_2
$Query \times Key$	attn.attn_scores_act_quantizer
$Softmax$	attn.attn.attn_probs_act_quantizer
$Weights\_output \times Value$	attn.context_act_quantizer
Attention projection layer	attn.c_proj
MLP fully connected layer	mlp.c_fc
MLP projection layer (Output FFN)	mlp.c_proj
Input and Output of FFN	mlp.c_proj + ln_2
Input and Residual Sum of FFN	ln_2 + res_act_quantizer_2
Output and Residual Sum of FFN	mlp.c_proj + ln_2

Table 6: Names of activations