Towards Greener Yet Powerful Code Generation via Quantization: An Empirical Study

Xiaokai Wei
xiaokaiw@amazon.com
AWS AI Labs
USA

Sujan Kumar Gonugondla
gsujan@amazon.com
AWS AI Labs
USA

Shiqi Wang
wshiqi@amazon.com
AWS AI Labs
USA

Wasi Ahmad
wuahmad@amazon.com
AWS AI Labs
USA

Baishakhi Ray
rabaisha@amazon.com
AWS AI Labs
USA

Haifeng Qian
qianhf@amazon.com
AWS AI Labs
USA

Xiaopeng Li
xiaopel@amazon.com
AWS AI Labs
USA

Varun Kumar
kuvrun@amazon.com
AWS AI Labs
USA

Zijian Wang
zijwan@amazon.com
AWS AI Labs
USA

Yuchen Tian
tiayuche@amazon.com
AWS AI Labs
USA

Qing Sun
qinsun@amazon.com
AWS AI Labs
USA

Ben Athiwaratkun
benathi@amazon.com
AWS AI Labs
USA

Mingyue Shang
myshang@amazon.com
AWS AI Labs
USA

Murali Krishna Ramanathan
mkraman@amazon.com
AWS AI Labs
USA

Parminder Bhatia
parmib@amazon.com
AWS AI Labs
USA

Bing Xiang
bxiang@amazon.com
AWS AI Labs
USA

ABSTRACT

ML-powered code generation aims to assist developers to write code in a more productive manner by intelligently generating code blocks based on natural language prompts. Recently, large pretrained deep learning models have pushed the boundary of code generation and achieved impressive performance.] However, the huge number of model parameters poses a significant challenge to their adoption in a typical software development environment, where a developer might use a standard laptop or mid-size server to develop code. Such large models cost significant resources in terms of memory, latency, dollars, as well as carbon footprint.

Model compression is a promising approach to address these challenges. We have identified quantization as one of the most promising compression techniques for code-generation as it avoids expensive retraining costs. As quantization represents model parameters with lower-bit integer (e.g., `int8`), the model size and runtime latency would both benefit.] We empirically evaluate quantized models on code generation tasks across different dimensions: (i) resource usage and carbon footprint, (ii) accuracy, and (iii) robustness. Through systematic experiments we find a code-aware quantization recipe that could run even a 6-billion-parameter model in a regular laptop without significant accuracy or robustness degradation. We find that the recipe is readily applicable to code summarization task as well.

CCS CONCEPTS

• **Software and its engineering** → **Software creation and management**; **Automatic programming**; **Integrated and visual development environments**.

KEYWORDS

Quantization, Code Generation, Large Language Models, Generative AI, Model Hosting

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0327-0/23/12.
https://doi.org/10.1145/3611643.3616302

ACM Reference Format:
Xiaokai Wei, Sujan Kumar Gonugondla, Shiqi Wang, Wasi Ahmad, 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. 2023. Towards Greener Yet Powerful Code Generation via Quantization: An Empirical Study. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616302

1 INTRODUCTION

In recent years, ML-powered code generation tools, like Codex [5], GitHub Copilot [14], Amazon CodeWhisperer[1] and ChatGPT[2] have gained significant traction. These services aim to generate a computer program in response to a human-written specification (commonly called *prompt*), as shown in Figure 1. Such tools bring promise to significantly automate the software development process and improve developers' productivity.

The backbone of ML-powered code generation tools are transformer based large pretrained language model (PLM) [3, 7, 30]. With recent rapid developments, PLMs have exhibited superior performance in multiple code-related tasks, including code generation, code summarization and type inference [3, 6, 7, 11, 12, 22, 30]. Despite their great success, there are multiple challenges and downsides associated with applying these gigantic code generation models, with often tens of billions of parameters, in a typical software development environment.

Importance of Model Compression. The need of compressed models in a typical development scenarios include:

- **Hosting.** The huge number of model parameters poses a significant challenge. For example, let us consider one of the largest publicly available models, CodeGen [30] with up to 16B parameters. Hosting this model requires 72 GB of memory, which is impossible on a typical laptop with 16 GB or 32 GB RAM. An alternative is to host this model on a server equipped with GPUs with sufficient memory and with model parallelism. Using EC2 pricing as reference, using such models is expensive and costs around $100+ per 1K queries. Furthermore, sizes of PLMs continue to grow and further limit deployment of future and more powerful models in real-life development environment. Note that, although services like Copilot, ChatGPT, etc. provide APIs for code generation, hosting customized model is still important for custom tasks [3].
- **Latency and user experience.** State-of-the-art code generation models typically consist of $20 \sim 50$ transformer layers and $2B \sim 16B$ parameters. Model inference/serving on a single GPU machine might incur a latency of several seconds. Such a delay in response would cause a negative user experience, especially for interactive code development.
- **Carbon footprint.** Recently, researchers [17, 31] start to pay more attention to examining PLMs from the perspective of responsible and green AI. The training and inference of large PLMs typically involve a considerable amount of $\mathbf{CO_2}$ emission. For example, the $\mathbf{CO_2}$ emission of training GPT-3 model (175B parameters) amounts to three times that of a jet plane flight from San Francisco to New York [31]. With increasing adoption of PLMs, e.g., by hundreds of thousands of software developers in the near future, the carbon footprint of inference will become a bigger issue.

To address these challenges, Machine Learning researchers have investigated different model compression techniques [37]. A key challenge, however, is to preserve the powerfulness of the gigantic models while significantly reducing the computational cost by compressing them. Addressing this challenge would be crucial to democratizing the power of AI. In this paper, we empirically investigate the impact of model compression techniques on code generation tasks.

Selecting Compression Technique. We focus on light-weight compression techniques that do not incur additional re-training cost. Also, the compressed model should be run efficiently in a developer's laptop or a moderate-sized server. In such a scenario, we identify the following desirable properties that a practical model compression strategy needs to satisfy:

- **Minimal compression cost**: Converting a pretrained model to a more efficient version typically involves certain processing/training costs. If the compression technique requires significant re-training of the large model over substantial amounts of data, it could result in undesirable environmental impacts (large power consumption and carbon footprint) and the cost could be prohibitive for a typical user. High compression costs would defeat the purpose of greener AI and democratizing AI.
- **Substantial reduction in hosting cost:** As state-of-the-art models are gigantic with billions of parameters and are expected to continue growing in sizes, minor reductions in size or latency would not be useful. Ideally, one would expect a properly designed model compression method to bring at least 50% improvement in key hosting metrics of size and latency.
- **Preservation of generation capabilities**: The compressed model should have similar generation accuracy as the original model. Model compression at the cost of significantly degenerated predictions would not be appealing.
- **Minimal adverse side effect**: In addition to preserving generation accuracy, we also expect the model to not degenerate in other important aspects of generation, such as weakened robustness.

Model compression techniques developed by ML community, such as distillation [39, 42], pruning [18, 21] and quantization-aware training [25, 40, 50] are often associated with large training costs. Training or finetuning large transformer models requires access to training data and substantial compute resources. Often, fine-tuning data for many freely available models is proprietary, which prevents a compression technique, which relies on fine-tuning, to reproduce the performance of the original models.

Considering all the above factors, out of many model compression options, we are able to identify a compression recipe with negligible processing cost and preserved accuracy with a specific subcategory of quantization methods, i.e., Post-Training Quantization (PTQ).

Quantization is a compression technique where the weights and activations of an ML model are converted to and computed with

[1] https://aws.amazon.com/codewhisperer
[2] https://chat.openai.com/chat
[3] https://ai.googleblog.com/2022/07/ml-enhanced-code-completion-improves.html

Input Prompt	Code	Test Cases

"""
# Write a Function to find the largest integers from 	
# a given list of numbers using heap queue algorithm. 	
"""
	import heapq as hq
	def heap_queue_largest(nums,n):
	    largest_nums = hq.nlargest(n, nums)
	    return largest_nums
		assert heap_queue_largest([25,35,22,85,14,65,75,22,58],3)==[85, 75, 65]
		assert heap_queue_largest([25,35,22,85,14,65,75,22,58],2)==[85, 75]
		assert heap_queue_largest([25,35,22,85,14,65,75,22,58],5)==[85, 75, 65, 58, 35]

Figure 1: Sample prompt, code, and test cases taken from MBPP dataset [3]. Given the NL prompt, a code generation model aims to generate the corresponding code. The associated test cases run the generated code to check functional correctness.

integer data types such as `int8` instead of commonly used float-point data types such as `fp32`. As data is represented with lower-bits (e.g., 8 or 4) the model would be much smaller in size. Also, most hardware types (either CPU or GPU) perform integer operations (e.g., multiplication) at a much faster speed; The quantized model would also likely to enjoy reduced computational cost and latency.

Code-aware Quantization. In order to quantize the model for code domain, we observe the activation ranges and calibrate the quantization step sizes that minimize the mean square error on the code data. Properly designed PTQ methods would require none or a relatively small amount of code data for post-training calibration, and experimental results show that this approach is highly effective on multiple code specific tasks. This means one can get all the compression benefits (e.g., latency/memory/storage/carbon emission) with negligible cost while retaining the generation power of the full-precision model.

Our contribution can be summarized as follows:

(1) We recognize the importance of model compression in the context of code generation and identify the adequacy of post-training quantization for this purpose. To our best knowledge, this is the first attempt at compressing a state-of-the-art code generation model. For example, the quantized model with 16B parameters could run on a personal laptop with only CPUs, and generate a 20-token long prediction within 25 seconds (as opposed to 70 seconds by the corresponding full-precision model).
(2) We perform an extensive empirical study on multiple code generation models with their quantized variations on both NL-to-code and code-to-NL tasks. We observe comparable accuracy across multiple model types and sizes with the proposed quantization techniques. Even for the extremely large CodeGen-16B, we can preserve accuracy with quantization. We also experiment in different ablation settings to provide guidelines for properly employing quantization.
(3) We present an in-depth empirical analysis on the layers, activations, and weights of the state-of-the-art code generation models to gain deeper insights on the effect of quantization. This helps us understand why certain quantization methods perform better than others, for code generation task.
(4) Beyond accuracy, we also investigate the impact of quantization on model robustness, which is often overlooked by the existing code generation literature. We show that properly designed quantization recipe would have no adverse impact on model robustness.

Relevance to Software Engineering. We identify the importance and requirements of model compression techniques for a typical development environment. We then adapt quantization, a light-weight compression technique, for code domain. We empirically

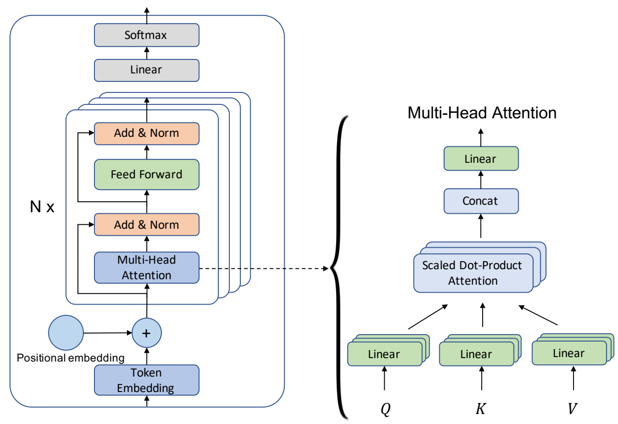


Figure 2: Transformer structure and multi-head attention cell. The feed-forward layer and all linear layers inside multi-head attention are colored in green. We quantize all these linear layers in the network.

establish that such approach can be significantly greener without loosing power of code generation in both CPU and GPU environments. To this end, this is the first systematic study to show that a specially crafted code-aware quantization technique can democratize gigantic pre-trained code generation models such that even a regular software development environment with 16GB laptop can utilize their power.

2 BACKGROUND & RELATED WORK

2.1 Code Generation with Transformers

Recently, applying transformer-based Pretrained Language Models (PLMs) to the source code generation task, have drawn considerable attention and set overwhelmingly strong state of the arts in this field [3, 6, 7, 12, 22, 30]. The goal is to generate complete code or code fragments given natural language or partial code as prompts. To achieve this goal, large language models are trained on humongous code corpora, typically curated from open-source code archives like GitHub, Stack Overflow, etc.

The PLMs typically use decoder-only (e.g., GPT [32]) or encoder-decoder architectures (e.g., BART [23]/T5 [33]). For code generation tasks, decoder-only models (e.g., CodeGen [30] and InCoder [12]) take some pre-encoded code representations and learn to decode, i.e., synthesize next token sequences. Typically, these models use causal language modeling, i.e, generate the tokens conditioned on the previous token sequences. Thus, decoder-only models are a natural fit for code completion tasks where the previous code context is given, and the model is expected to generate the next tokens. In contrast, encoder-decoder based code generation models like PLBART [1] and CodeT5 [43] are typically trained to learn to reconstruct the original code sequence that is corrupted using an arbitrary noise function. Therefore, such models do not naturally fit the code completion tasks but are found effective when finetuned for code generation or summarization tasks.

2.2 Model Compression

The large transformer models use billions of parameters and may require trillions of operations for generating code. Model compression tackles this high costs of large models to enable their wider and easier adoption. Model compression is a class of techniques designed to reduce model size (i.e., bytes required to represent the model) and improve generation latency while maintaining minimum accuracy (i.e., ability to generate useful and correct code) degradation. Some representative techniques include:

(1) *Knowledge Distillation.* A small student model is trained on the outputs of a larger teacher model that we want to compress [39, 42].
(2) *Pruning.* It constitutes a class of techniques that make the weight matrices sparse to reduce the number of parameters as many of the matrix entries will now be zeros [18, 21, 44, 45].
(3) *Quantization.* This technique uses fewer bits to represent the weights of parameterized functions [4, 46, 47, 50].

Most model compression techniques are often associated with large training costs. For example, distillation requires us to train a new model from scratch, and pruning requires multiple cycles of finetuning. Training or finetuning large transformer models requires access to the training data and large compute resources. This is often not an option for many users who typically use fundamental models that are pretrained on large training corpus by others. Therefore, we study post-training quantization (PTQ) for model compression as this does not require us to train or finetune a pretrained model.

We particularly limit to 8-bit and 4-bit integer quantization as the hardware accelerators such as GPUs/TPUs support and compute faster with this precision that the floating point alternatives such as 16-bit Float.

2.3 Quantization for Model Compression

Here we describe the process of quantizing a tensor and discuss different model-quantization techniques.

2.3.1 Quantization Operation. Quantization refers to the conversion of a full-precision (or floating-point) tensors to tensors with integer values. An example of the quantization operations is depicted in Figure 3. Given a matrix W, a basic quantizer $Q(\cdot)$ uses scale and rounding operations to get the quantized version of the matrix:

$$Q(W) = \frac{W_q}{s_W}, \text{ where } s_W = \frac{2^{B-1}}{\alpha_W} \text{ and } W_q = \text{round}(s_W W)$$

Here, α_W is the quantization range, and B is the bitwidth (which is 8 in case of `int8`), W_q is the quantized integer matrix, s_W is the quantization scale, and $Q(W)$ is the quantized approximation of the matrix W

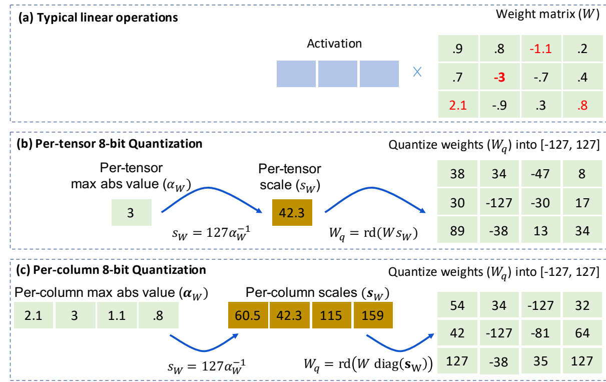


Figure 3: Toy example for quantizing the typical floating-point weight matrix (a) into `int8` matrix using (b) per-tensor v.s. (c) per-column quantization.

Quantization Noise. We assess the quality of quantization by estimating the relative quantization noise q_a, defined as [36]:

$$q_a = \frac{||A - Q(A)||_2}{||A||_2} \approx \frac{\Delta_A^2}{12||A||_2} \approx \frac{1}{12 s_A^2 ||A||_2} \tag{1}$$

where $||x||_2$ is the the L_2-norm of the vector x, and $\Delta_W = 1/s_W$ quantization step size. The quantization noise increases with Δ_W (or decreases with s_W), as the approximation of the full precision parameters becomes coarser.

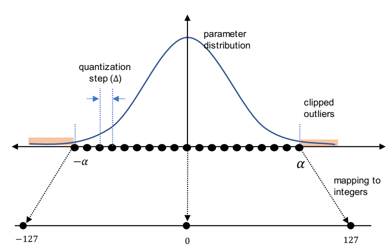
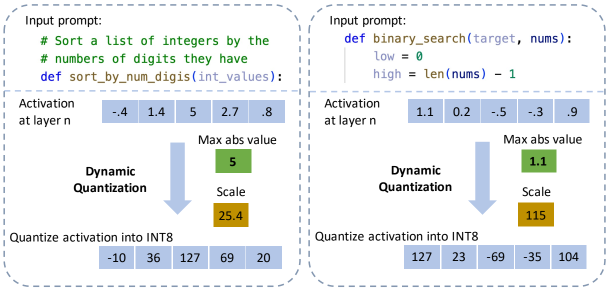


Figure 4: Illustration of quantization operation showing, quantization step, clipping, scaling, and mapping.

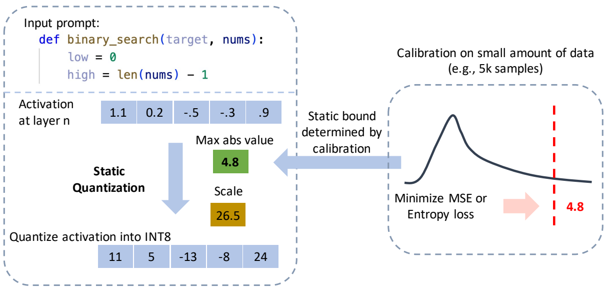
Quantization Range and Scale Factor. The quantization range α_W is the value that will be mapped to the largest representable integer (127 in the case of `int8`). Typically we set $\alpha_W = \max(\text{abs}(W))$, consequently setting the scale factor $s_W = 2^{B-1}/\max(\text{abs}(W))$. However, having a large outlier in W will increase α_W and therefore increase the quantization noise. To avoid that, some choose to clip the data by choosing $\alpha_W < \max(\text{abs}(W))$ (see Figure 4), where the matrix elements are clipped the s before the quantization operation i.e., matrix elements $> \alpha_W$ are set to α_W and those $< -\alpha_W$ are set to $-\alpha_W$.

2.3.2 Quantization Techniques. Model quantization techniques can be classified based on the following:

Methods to Obtain Quantized Network. This can be broadly classified into



(a) Dynamic quantization where the clipping range is determined dynamically as the max abs value in activation tensors.



(b) Static quantization where the fixed clipping range is learned through calibration.

Figure 5: Toy example on quantizing activations with dynamic quantization v.s. static quantization.

- *Post-Training Quantization (PTQ)*: PTQ derives a quantized (e.g., `int8`) network from an existing full-precision network without any training or finetuning with additional data. Since the model is not originally trained to perform inference with quantized parameters and activation, models quantized by PTQ tend to be more susceptible to quantization noise. However, the low costs associated with PTQ make it a very popular choice for obtaining quantized models.
- *Quantization-Aware Training (QAT):* QAT requires training the model from scratch with additional simulated quantization operations during training process to ensure the learned parameter values are quantization-friendly. This is expensive due to the potentially huge cost of training but would lead to models that potentially have higher accuracy than a PTQ model.

Methods to Choose the Activation Scale. Here we choose the ranges and the values of the activations change for each example. There are various options to choose the quantization scale parameters for activations that can be classified into (see Figure 5):

- *Dynamic quantization:* Here we determine the clip range (α) and scale parameter (s) on the fly for activations, in order to minimize quantization noise where possible. One could typically use the maximum (absolute) value of the activation tensors as the clip range for each input. However, determining the clip range dynamically would incur an additional scanning cost to find the max value.
- *Static quantization:* Here we use the same pre-deter-mined scale through so-called **calibration** on samples by minimizing certain loss (e.g., MSE/Entropy) between original activation and quantized activations. Static quantization might be susceptible to higher quantization noise though it would lower computational cost during inference.

Quantization Granularity. As we discussed in the Section 2.3.1, choosing a large clip range (accordingly, small scales s_A) due to outliers can lead to a large quantization step which adds to quantization noise. To avoid outliers, column-wise quantization scales can be used where the scales are selected based on the max value of each column instead of the entire matrix. We can classify quantization techniques based on the granularity of the quantization scales into 1) *per-tensor scales* where the entire tensor uses a single scale s_A, and 2) *per-column* where each column uses a different scale.

Figure 3 illustrates the differences between the two scaling options. Here, we are choosing scale values based on the maximum absolute value of the block. Choosing per-column scales avoids tensor-wide outliers and allows for finer quantization steps than per-tensor scales.

In the rest of the paper, we will primarily use PTQ as this has minimal post/re-training cost. We examine the accuracy of the models with dynamic and static quantization and discuss the impacts of choosing per-tensor and per-row scales. We will use `int8` precision for quantization as it is widely supported across all major CPUs and GPUs that are used today.

3 METHODOLOGY

The goal of this work is to provide an empirical and conceptual analysis of quantization techniques, originally developed as a core ML technique, in the context of large code generation models. To do that, we analyze the characteristics of the models using different dimensions of quantization techniques, as discussed in Section 2.3.2. This section discusses our study methodology in detail.

3.1 Quantized Model Preparation

For quantization techniques, we investigate both schemes of quantization (dynamic and static) described in previous sections and prepare the quantized models as follows.

- **Dynamic quantization:** For implementation, we use the native PyTorch Quantization [4] API and convert all the weight matrices in Feed Forward Network (FFN) and Self-Attention to `int8`. As explained in the previous section, the min/max bound of each layer's activation is determined in a dynamic manner depending on the input during inference. The processing time needed for this scheme is minimal, which typically takes < 1 minutes for 2*B* models and < 4 minutes for 6*B* models.
- **Static quantization:** Static quantization needs to determine the clipping range for activations before inference, and such ranges are typically obtained from calibration by minimizing the quantization noise. We perform the activation-bound calibration with a tiny fraction (5k samples) from the CodeSearchNet (Python) training set. In preliminary experiments, we find *MSE (Mean Squared Error)* based loss to be most effective, so we minimize

[4] https://pytorch.org/docs/stable/quantization.html

Table 1: Details of models under investigation.

Models	#Parameters	Training Cost			Architecture
		#Steps / #Epochs	Data (approx.)	Compute Resources	
PLBART	140M / 406M	100k / -	250 GiB	8 NVIDIA GeForce RTX 2080 Ti	Encoder-Decoder
Code-T5	60M / 220M / 770M	- / 150	25 GiB	16 NVIDIA A100 GPUs	Encoder-Decoder
InCoder	1.3B / 6.7B	- / 1	216 GiB	248 NVIDIA V100 GPUs	Decoder-only
CodeGen	350M / 2B / 6B / 16B	650k / -	1812 GiB	Google's TPU-v4	Decoder-only

the MSE between the quantized activations and full-precision ones as the calibration.

3.2 Study Subjects

Studied Models. We leverage the state-of-the-art and representative code generation models that have open sourced model checkpoints available, to study the efficacy of different calibration techniques. We aim to cover models with different sizes and backbone architectures. In particular, we focus on CodeGen [30], as they open sourced models with different sizes {350M, 1B, 6B, 16B} and different language support (mono v.s. multi-language generation). Additionally, we also include InCoder [12] to further confirm the patterns we observe with CodeGen models. We also studied two more models Code-T5 [43] and PLBART [1] for code summarization task. The statistics of these models are summarized in Table 1. These models represent the set of all publicly available state of the art models code-generation models at the time of writing this paper.

Studied Tasks. In this paper, our main focus is code generation task (NL-to-code). Further, to stress test the effectiveness of quantization on other generative tasks, we study code summarization task for models' accuracy evaluation (RQ4). Thus, we study the following two tasks:

- **NL-to-code generation**: Here we evaluate the models' code generation ability. A user gives a natural language prompt as input. These are loosely defined specifications. The model is expected to generate the corresponding code fragments. The generated code is tested by running the test cases. Figure 1 shows an example.
- **Code-to-NL generation**: We further evaluate a generative model's capability on code summarization task, where given the function signature and body, the model generates an NL description of the function.

Studied Dataset: We use HumanEval [6] and MBPP [3] for evaluating the functional correctness of generated programs. The MBPP dataset [3] contains 974 short Python functions with their textual descriptions and test cases to evaluate correctness (see Figure 1). HumanEval [6] is a similar dataset released by OpenAI, which is widely used in evaluating code generation tasks. It contains 164 hand-written Python programs, associated with their natural language descriptions and test cases.

Evaluation Metrics: Generative models in NLP domain traditionally use some form of textual matching (exact or fuzzy match) between the generated text and ground truth and often report BLEU scores. Such textual similarity is problematic for evaluating code generation tasks, as the same functionality can be implemented in many ways. To overcome this, recent papers on code generation task [5, 20, 35] recommend to evaluate functional correctness by running the generated code against test cases. Here we follow a similar evaluation criterion.

Each sample in our studied dataset is equipped with multiple test cases, as shown in Figure 1. The generated code needs to pass *all* provided tests to be considered as "*pass*". Following [5, 20], we report pass@k to estimate the model's ability to generate code that will "*pass*". Pass@k measures the fraction of examples that are "*pass*" by at least one of the k solutions that the model generates. However, given the ML model is probabilistic, we expect pass@k to have a high variance. To address this, a standard practice is to generate $n > k$ solutions and estimate the statistical mean of pass@k from these n samples, i.e., estimate the fraction of times we "*pass*" if we randomly pick k samples from n. In this paper, we use pass@1 and pass@5 as a metric for evaluations, which is estimated by generating 10 samples per problem in the dataset. The reported accuracy (pass@k) is averaged on all samples generated for all programs in each dataset.

To evaluate the code summarization models, we use smoothed BLEU score [26] following prior works [1, 43].

4 RESULTS

We evaluate the effect of quantization across three dimensions: greener, accuracy, and robustness for code generation tasks. To evaluate generalizability, we further evaluate quantization techniques for code summarization tasks, as code summarization is a popular code-related generative task where a different modality, i.e., text, is generated. In particular, we aim to answer the following four research questions:

- **RQ1.** How effective are quantization techniques for greener code generation models?
- **RQ2.** Can quantized models maintain the prediction power of the corresponding full-precision models?
- **RQ3.** How robust are quantized models compared to the corresponding full-precision models?
- **RQ4.** Are quantization techniques effective for other code related generative tasks such as code summarization?

4.1 Quantization for Greener Code Generation (RQ1)

Motivation. The heart of this paper lies on this RQ, i.e., whether a quantized model can be substantially greener than its full precision counterpart. Here by green, we mean less resource usage and less carbon footprint. Our use case is to facilitate a regular development environment that can benefit from such large models. Thus, a full precision model can be pretrained with larger resource (even at industry scale). However, a developer will be using the model in an environment which is either CPU-only or contain a smaller number of GPUs. To this end, this RQ evaluates the model's resource usage and carbon footprint at inference time.

Experimental Setup. We aim to answer RQ1 by investigating quantization from a model hosting perspective, with GPU or CPU as the underlying hardware. We consider both on-cloud and on-device settings as both can be important use cases for code generation models. The environment used for experiment is the following:

- **On cloud**: We use an AWS `p3dn.24xlarge` instance[5] which have both CPUs and GPUs available with NVMe-based SSD storage.
- **On device**: We use a typical developer's laptop, a MacBook Pro which runs macOS Monterey (version 12.5), with 32 GB memory and M1 processor.

Metrics. We report inference latency and model storage size as primary metrics for model hosting. Based on the latency result and the specification of underlying hardware, we also estimate (assuming sequential prediction) the potential cost[6] (in US$) and carbon emission [7] (in gCO_2eq) for evaluating the impact in terms of green AI.

Table 2: Comparison on different hosting metrics between full-precision and quantized (`int8` dynamic) versions of CodeGen-2B, CodeGen-6B and InCoder-6B.

	CodeGen-2B		CodeGen-6B		InCoder-6B	
On Cloud / Precision	**fp32**	**int8**	**fp32**	**int8**	**fp32**	**int8**
Storage (GB)	10.7	3.5	27.1	7.9	25.4	7.6
Latency (s/pred.)	3.47	2.47	7.81	4.02	7.38	3.32
Est. gCO_2eq (1k pred.)	1309	932	2940	1516	2783	1252
Est. pricing (1k pred.)	$30.1	$21.4	$67.7	$34.8	$64.0	$28.8
On Device						
Latency (s/pred.)	23.7	10.7	70.4	25.4	52.6	18

Observations. **CPU-based results.** In Table 2, we report (on cloud and on device) hosting metrics of CodeGen-2B/6B and InCoder-6B model for generating 20 tokens for each example. As the quantization kernel in Pytorch only supports CPU inference, we collect all the metrics on CPUs. For both CodeGen-6B and InCoder-6B, we observe that `int8` quantization reduces the model size to about 29% of `FP32` counterpart and also reduces latency significantly (e.g., by about 50% on ec2 instance and > 60% on laptop). As the carbon emission and pricing are roughly linear w.r.t. the runtime, using a quantized model would also contribute significantly to green AI and reduced hosting cost. With the much less stringent requirements on the underlying hardware, quantization makes it possible to run large (e.g., 6B) code generation models on a personal laptop within a reasonable latency constraint. Such capability can be helpful for developers to get high-quality code recommendation/auto-completion in their local environment.

[5]More details on the hardware specification can be found at https://aws.amazon.com/ec2/instance-types/p3/

[6]Based on estimate in https://www.instance-pricing.com/provider=aws-ec2/instance=p3dn.24xlarge

[7]https://engineering.teads.com/sustainability/carbon-footprint-estimator-for-aws-instances/

Table 3: Latency and memory usage of CodeGen models on a Nvidia A100 GPU with context-lengths of 1792 tokens and generation lengths of 256 tokens.

Model Size	**2B**	**6B**	**16B**
context encoding latency (ms)			
`fp32`	173.9 ± 0.02	266.6 ± 0.69	N/A
`fp16`	97.8 ± 0.08	146.13 ± 0.36	290.43 ± 0.50
`int8`	89.5 ± 0.04	117.3 ± 0.27	221.5 ± 0.49
per-token generation latency (ms)			
`fp32`	20.1 ± 0.021	38.6 ± 0.009	N/A
`fp16`	16.9 ± 0.017	23.1 ± 0.12	40.87 ± 0.003
`int8`	13.5 ± 0.024	19.6 ± 0.004	32.02 ± 0.01
peak memory usage (GB)			
`fp32`	15.14	32.63	OOM
`fp16`	8.54	17.34	35.25
`int8`	6.45	11.69	22.11

GPU-based Results. To assess the impact of using INT8 inference on GPUs, we developed an inference pipeline using INT8 CUDA kernels from NVIDIA's Cutlass library. Our end-to-end latency measurements, with a 1792-token context and 256-token generations, are reported in Table 3 for CodeGen models. By comparing the `fp16` and `int8` implementations, we observed a reduction of up to 20% in both context encoding latency and generation latency, along with a 30% decrease in GPU memory usage. The gains in comparison to the `fp32` implementation are approximately 2 to 2.5 times reduction in latency and 2.5 times more memory-efficient. This effectively results in a doubling of the inference speed and an halving of the number of required GPUs for deployment.

> **Result 1:** *The quantized models have lower latency, memory, and storage than the corresponding full precision model. It also has remarkably less carbon footprint. Thus, it is possible to fit even a 6B-parameter model within a regular laptop.*

4.2 Accuracy Evaluation for Code Generation Task (RQ2)

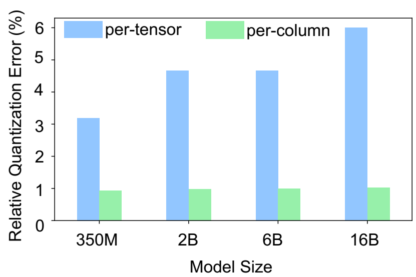
Motivation. Although greener, a quantized model will be mostly useful if it maintains the accuracy of the original full precision model. In this RQ, we evaluate the functional correctness of code generation models for full precision and their different quantized variants.

Experimental Setup. We evaluate the code generation tasks using CodeGen and InCoder quantized models with static and dynamic activation quantization. We tested the models with per-column scales and per-tensor scales while quantizing the weights as well. We report both pass@1 and pass@5 accuracies.

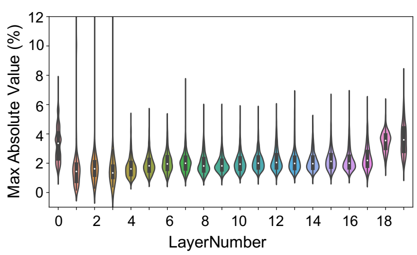
Observations. Table 4 summarizes the results. We see accuracy gain for InCoder-6.7B models across all the quantization settings, while InCoder-1B shows an average accuracy drop of 0.84% on HumanEval and 2.47% on MBPP datasets. CodeGen models show < 2% average degradation with pass@1 metric on HumanEval and

Table 4: Pass@k (%) accuracy on HumanEval and MBPP. Performance gains are in blue and drops in red.

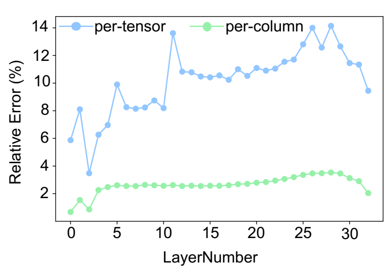
Dataset	Model	Full-precision		Dynamic Quant. (per-tensor)		Static Quant. (per-tensor)		Static Quant. (per-column)	
		pass@1	pass@5	pass@1	pass@5	pass@1	pass@5	pass@1	pass@5
HumanEval (Python)	InCoder-1.3B	7.13	8.98	5.55 (**-1.58**)	8.33 (**-0.65**)	5.85 (**-1.28**)	7.99 (**-0.99**)	6.71 (**-0.42**)	8.86 (**-0.12**)
	InCoder-6.7B	8.11	9.70	8.23 (**+0.12**)	10.52 (**+0.82**)	8.41 (**+0.30**)	10.46 (**+0.76**)	9.27 (**+1.16**)	11.38 (**+1.68**)
	CodeGen-350M	11.71	16.21	11.77 (**+0.06**)	14.70 (**-1.51**)	10.79 (**-0.92**)	14.90 (**-1.31**)	11.83 (**+0.12**)	16.66 (**+0.45**)
	CodeGen-2B	20.91	27.75	18.48 (**-2.43**)	26.56 (**-1.19**)	17.87 (**-3.04**)	26.13 (**-1.62**)	22.50 (**+1.59**)	29.59 (**+1.84**)
	CodeGen-6B	24.02	36.82	26.71 (**+1.69**)	34.27 (**-2.55**)	25.37 (**+1.35**)	34.02 (**-2.80**)	25.73 (**+1.71**)	33.74 (**-3.08**)
MBPP (Python)	InCoder-1.3B	5.92	10.27	4.11 (**-1.82**)	7.87 (**-2.40**)	3.68 (**-2.25**)	7.06 (**-3.21**)	3.82 (**-2.10**)	7.22 (**-3.05**)
	InCoder-6.7B	7.53	11.55	7.75 (**+0.23**)	11.79 (**+0.24**)	7.86 (**+0.34**)	12.30 (**+0.75**)	7.80 (**+0.28**)	12.37 (**+0.82**)
	CodeGen-350M	16.99	25.39	15.32 (**-1.67**)	23.35 (**-2.04**)	15.32 (**-1.67**)	23.85 (**-1.54**)	15.87 (**-1.12**)	24.28 (**-1.12**)
	CodeGen-2B	31.57	41.97	28.10 (**-3.47**)	38.24 (**-3.73**)	27.38 (**-4.19**)	39.04 (**-2.93**)	30.59 (**-0.98**)	40.93 (**-1.04**)
	CodeGen-6B	34.00	51.97	34.49 (**+0.49**)	45.42 (**-6.55**)	34.74 (**+0.49**)	45.74 (**-6.23**)	37.35 (**+3.35**)	48.90 (**-3.07**)



(a) Weight quantization noise for CodeGen models.



(b) Distributions of maximum activation value across different layers in CodeGen-350M model on validation data.



(c) MSE/layer between activations from original vs. quantized model. The error is estimated on a weight-only quantized CodeGen-6B model generating a token on HumanEval.

Figure 6: Statistics impacting quantized model accuracy.

MBPP datasets with both Dynamic and Static quantization. However, we observe 3%-4% and 2% average drop in accuracy in the pass@5 metrics with dynamic quantization and static (per-tensor) quantization respectively. With static (per-column) quantization the average pass@5 accuracy drop is < 2% for CodeGen models.

Overall, dynamic (per-tensor) quantization tends to outperform static (per-tensor) quantization by a small margin and static (per-column) quantization outperforms static (per-tensor) quantization. This is because:

- **Weight Quantization.** Weight distributions have a high variance within a kernel, accounting for outliers that result in large quantization noise. This is particularly an issue with increasing matrix sizes in larger models. Figure 6a shows how the quantization noise increases with model sizes with per-tensor scales, but not with per-column scales. This reduced quantization noise with per-column scales explains why static (per-column) setting outperforms static (per-tensor) one.
- **Activation Quantization.** The primary challenge in activation quantization is in choosing the quantization scales. With static-quantization, we have to pick pre-determined scales based on validation data. This pre-determined scale is picked conservatively On the other hand, dynamic quantization allows us to adjust the scales for every input example and for every token, thereby making it attractive to reduce quantization noise. Dynamic quantization will be useful if we observe high variance in the max-values across different inputs/tokens. For example, Figure 6b shows the max value of the activation across different layers in CodeGen-350M.
- **Error Accumulation.** Quantization noise accumulates with depth, making deeper models more challenging to quantization. Figure 6c shows the relative quantization noise with model depth for CodeGen-6B model, showing quantization error growing with depth. We observe that a) per-column quantization results in smaller accumulated error with depth and b) the error tends to reduce in the last few ($\tilde{4}$) layers of the model. The latter could be due to the inherent robustness of the model.

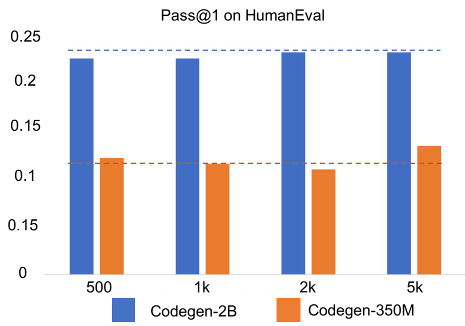


Figure 7: Execution accuracy on HumanEval with CodeGen-2B and CodeGen-350M (per-column static) when they are calibrated (*MSE* loss) on different amounts of data (from 500 **to** 5k**). Dotted lines denote the pass@**1 **of corresponding full-precision models.**

4.2.1 Ablation Study. To better understand the impact of different design choices on the model, as discussed in Section 3.1, we further investigated pass@1 scores for different model variations on HumanEval.

Size of Calibration Set. Here, we study how the size of calibration data affects the performance of quantized models. Figure 7 shows that the execution accuracy (on both 2B and 350M models) is typically stable across different sizes of calibration data. When using only 500 samples for calibration, the quantized model can already learn a reasonable clipping range (α) and achieve comparable accuracy as full-precision baselines. Such calibration cost (e.g., takes a few minutes on a single CPU/GPU) is almost negligible compared to other model compression options, such as distillation, which typically requires iterating over the whole training corpus and takes weeks to finish.

Impact of Precision. We experimented with using 4-bit precision instead of the 8-bits that we use in the rest of the paper. The experiments with different precision settings on CodeGen-2B models on HumanEval and the results are summarized in Table 5. We use the static (per-column) quantization setting for these experiments.

With 8-bit weights and activation (W8A8), we can meet the accuracy of a full-precision model on HumanEval. However, this accuracy drops by $\approx 4\%$ with weights quantized with 4-bits while activations remain quantized with 8-bits (W8A4). We find that the model does not generate any meaningful outputs when activations are quantized with 4-bits while the weights remain quantized with 8-bits (W8A4), indicating that the model is more sensitive to activation quantization than those of the weights.

4.2.2 Quantizing Extremely Large Code Generation Models. So far we have seen that appropriately designed quantization techniques could preserve accuracy for models with medium to large sizes (up to 6B parameters). Now we conduct an extreme study with CodeGen-16B, one of the largest publicly available code generation models.

From Table 6, one can observe that both dynamic and static (per-column) quantization achieve competitive results compared to the original model. For example, dynamic quantized model (model size: 17 GB) could achieve similar pass@5 and slightly lower pass@1 compared to the significantly more gigantic FP32 model (75 GB).

Table 5: Execution accuracy of CodeGen-2B model at different activation and weight precision settings on HumanEval. Here WxAy indicates x-bit weights and y-bit activations.

pass@	1	5
Full precision	20.91%	27.75%
W8A8	22.50%	29.59%
W4A8	18.54%	24.83%
W8A4	0.61%	1.39%

Table 6: Execution accuracy on CodeGen-16B and HumanEval.

pass@	1	5
Full-precision	29.39%	39.02%
Dynamic Quantization	27.68%	39.63%
Static (per column) Quantization	26.40%	34.78%

Result 2: *Quantization models often suffer minimal accuracy drop from the corresponding full precision models making them potential design choices for implementing Greener Code Generation models.*

4.3 Robustness Evaluation (RQ3)

Motivation. It is well known that DL models are sensitive to input perturbations [16, 19, 27, 41, 49]. In particular, a good quantized model should not adversely impact the robustness of a model, i.e., the original full-precision model's robustness should not decrease drastically after quantization.

Experimental Setup. To evaluate the effect of quantization on a model's Robustness, we evaluate both the original and the quantized models on HumanEval [6] and MBPP [3] dataset with perturbed inputs. In the NLP domain, researchers propose different semantic preserving perturbations to inputs; e.g., mutating words with their synonyms [2, 9, 34] or character-level mutations [13, 48]. We adapt similar techniques in our context. In particular, we perturb the text in each prompt with three different types of perturbations respectively (see Table 8):

(1) *Character-level Perturbations* by changing randomly selected characters to upper cases.
(2) *Word-level Perturbations* by substituting randomly selected words with synonyms from WordNet [28];
(3) *Sentence-level Perturbations* by paraphrasing the whole text with back translation [24, 38]. In specific, it transforms the English docstring into German and then translates back to English.

For these three types of perturbations, we use the default settings and implementations from a standard text perturbation benchmark NL-Augmenter [8]. These perturbations are designed such that the original semantics of the natural language remains unaltered [15, 29, 51]. Then we measure the average pass@1 with greedy sampling for each model on the three perturbed datasets along with the

Table 7: The percentage of the pass@1 *drop* on the datasets with character-level (Ch), word-level (W), and sentence-level (S) perturbations of prompt compared to the unperturbed ones.

		HumanEval			MBPP		
		Ch	W	S	Ch	W	S
InCoder							
1.3B	FP*	**0.00**	18.18	-9.09	30.00	35.00	8.33
	D (T)	11.11	11.11	11.11	**10.81**	24.32	13.51
	S (C)	**0.00**	18.18	0.00	40.00	30.00	**7.50**
	S (T)	10.00	**10.00**	**-10.00**	31.58	**23.68**	15.79
6.7B	FP	**-7.69**	30.77	7.69	24.68	25.97	10.39
	D (T)	7.69	7.69	7.69	18.42	26.32	15.79
	S (C)	0.00	**7.14**	14.29	**9.59**	**19.18**	**-4.11**
	S (T)	-7.14	14.29	**-7.14**	25.97	24.68	7.79
CodeGen							
350M	FP	**10.53**	**10.53**	15.79	13.56	19.21	6.78
	D (T)	15.79	15.79	**5.26**	17.72	13.92	7.59
	S (C)	22.73	18.18	13.64	14.91	**12.42**	**3.11**
	S (T)	33.33	23.81	14.29	**13.16**	14.47	5.26
2B	FP	**12.82**	**15.38**	20.51	7.99	**9.27**	6.39
	D (T))	29.73	32.43	27.03	6.79	11.79	**-1.07**
	S (C)	13.16	23.68	18.42	10.03	15.53	7.12
	S (T)	15.15	27.27	**12.12**	**7.72**	9.56	2.21
6B	FP	17.78	24.44	28.89	**-0.85**	**4.55**	0.28
	D (T)	30.00	40.00	34.00	6.34	12.97	6.05
	S (C)	20.93	**20.93**	**16.28**	6.96	8.36	**-0.84**
	S (T)	**15.56**	28.89	20.00	6.10	9.01	2.62

*FP=Full-precision; D (T)=Dynamic (per-tensor); S(C)=Static (per-column); S(T)=Static (per-tensor)

unperturbed ones to avoid randomness and better observe the robustness trends.

To measure the robustness of a model, we compute the change in pass@1 results between perturbed and unperturbed inputs. For each type of perturbation, we compute the percentage change across all the inputs in a dataset, as: $\%\Delta = \frac{\text{pass@1}_{\text{unperturbed}} - \text{pass@1}_{\text{perturbed}}}{\text{pass@1}_{\text{unperturbed}}}$.

Table 7 reports the results. The lower the value of Δ, the better the robustness of a model. A negative drop means the model performs better with perturbed inputs.

Observations. The results show that, overall all the quantization methods, including per-tensor dynamic, per-tensor static, and per-column static, have comparable robustness performance w.r.t. the corresponding full precision model. In certain cases, in fact, quantized models perform better (as shown in red). On average across all model types and perturbations, full precision, per-tensor dynamic, per-tensor static, and per-column static quantized models have 13.27%, 15.92%, 12.91%, and 13.33% percentage of the drops on MBPP and HumanEval datasets. Models quantized with static per-column overall have slightly better robustness performance compared to the ones quantized with per-tensor quantized models.

We further compute per sample difference in pass@1 result between a quantized and the corr. full-precision model using Wilcoxon-Mann-Whitney test [10]—this confirms the difference between the two models is statistically insignificant.

> **Result 3:** *Quantization does not have any negative impact on model's robustness— a quantized model reacts to perturbed inputs very similarly as the corresponding full-precision model.*

4.4 Accuracy for Code Summarization (RQ4)

Motivation. Here we check whether the quantization techniques studied so far are also applicable to other code-related tasks. In particular, we chose code summarization, as it is reversing the modality studied so far (NL for code).

Experimental Setup. Here, we use the *finetuned* PLBART and CodeT5 models on the code summarization task (in Python) released by the authors. Since CodeGen is not designed to generate summaries given a code snippet, we do not use it in the evaluation. In our early experiments, we evaluated InCoder full precision models on this task based on the author released code, but got very poor performance, therefore, we do not pursue the model.

Observations: Table 9 shows the results. We observe almost no drop in BLEU score for CodeT5 models with both Dynamic and Static quantization. In comparison, while PLBART with Dynamic quantization matches the full-precision performance, we observe a performance drop with Static quantization. To understand this performance degradation, we perform a qualitative comparison between these two settings. A few examples are provided in Table 10. Overall, we observe that PLBART with static quantization generates shorter summaries that affect the BLEU score. However, the generated summaries are semantically comparable to the full precision version.

> **Result 4:** *Quantized models behave comparably to the corr. full-precision models for code summarization.*

5 THREATS TO VALIDITY

The main threats to the validity of our conclusions are external, relating to the generalization of our findings, to both other types of compression techniques and to other ML-powered code related tasks. First, as discussed in Section 2, quantization-based compression techniques are mostly suitable for usecase as a typical developer may not have resources to retrain the model from scratch using other compression methods. Second, we focus on mostly generative tasks, and thus study code generation (NL-to-code) in detail. To evaluate the generalizability of our findings, we also investigate the effect of quantization on code summarization (RQ4). Finally, we have other threats including studying each of these tasks on two models and two dataset respectively. However, these are state-of-the-art open source models and data widely studied in the literature. We further studied the different sizes of these models. We evaluated on perturbed data (RQ3) which also gives us confidence on the stability of our results. Besides, all the other quantization-related parameters used in the experiments are empirically evaluated. We also report the most stringent measurement (pass@1) to reduce any measurement bias.

Table 8: Example impact of word-level, character-level, sentence-level perturbations on full-precision and per-tensor dynamic quantized models. The perturbed region is underlined.

	Examples	Docstring	Passing All Tests	
			Full-precision	Dynamic (per-tensor)
S1	Unperturbed	Write a python function to determine whether all the numbers are different from each other are not.	✓	✓
	Character-level	Write a python function to determine whetHer alL the numbers aRe different from each otheR are not.	✓	✓
	Word-level	Write a python function to determine whether all the numbers are unlike from each other are not.	✓	✓
	Sentence-level	Write a Python function to see if all numbers differ from each other.	✓	✓
S2	Unperturbed	Write a function to extract the index minimum value record from the given tuples.	✓	✓
	Character-level	Write a function to extract the index miniMum vaLue record froM the given tuples.	✓	✗
	Word-level	Write a function to extract the index minimal value record from the give tuples.	✓	✓
	Sentence-level	Write a function to extract the index minimum dataset from the given tuples.	✓	✓
S3	Unperturbed	Write a function to print check if the triangle is equilateral or not.	✓	✓
	Character-level	Write a function to print check if the triaNgle iS equilateral Or not.	✓	✓
	Word-level	Write a function to print check if the triangle equal equilateral or not.	✗	✓
	Sentence-level	Write a function to check whether the triangle is equilateral or not.	✗	✓

Table 9: Smoothed BLEU scores for code summarization.

	Full- -precision	Dynamic (per-tensor)	Static (per-tensor)	Static (per-column)
PLBART	17.02	17.00 (**-0.02**)	14.96 (**-2.06**)	15.19 (**-1.83**)
CodeT5	19.50	19.44 (**-0.06**)	19.27 (**-0.23**)	19.30 (**-0.20**)

Table 10: Qualitative comparisons of summaries by PLBART in full-precision and static quantization.

Full-precision	Static (per-tensor)
Copy an entire table to a temporary file.	dump the contents of a table to a temporary file
Recursively make all intermediate directories and subdirectories.	helper function to make intermediate dirs
Downloads a video by its id and title.	download by vid
Generate RST API documentation for a module.	Generate the documentation for the given module.

6 CONCLUSION

Code Generation models based on large PLMs have set the new state-of-the-art in generating functionally correct code given natural language description. However, the sizes of these models could be prohibitively large (e.g., billions of parameters), which can cause problems for green AI and responsible AI. Therefore, developing approaches towards improving model efficiency yet preserving their powerful generation capability is of great practical importance. In this paper, we address this problem by developing a quantization-based recipe for such models. We demonstrate the efficacy of proposed methods in terms of greenness, accuracy, and robustness. As future work, we would like to investigate the efficacy of quantization for more code intelligence applications, such as code search, code editing, and code translation.

DATA AVAILABILITY

The data presented in this paper is based on open-source models and on publicly available datasets, and therefore reproducible. The information required to reproduce the results are in Section 3.2. The code used to generate the the evaluation results is available at https://github.com/amazon-science/recode/tree/fse_quantization.

REFERENCES

[1] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, Online, 2655–2668. https://doi.org/10.18653/v1/2021.naacl-main.211

[2] Moustafa Alzantot, Yash Sharma, Ahmed Elgohary, Bo-Jhang Ho, Mani Srivastava, and Kai-Wei Chang. 2018. Generating Natural Language Adversarial Examples. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Brussels, Belgium, 2890–2896. https://doi.org/10.18653/v1/D18-1316

[3] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *arXiv e-prints*, Article arXiv:2108.07732 (Aug. 2021), arXiv:2108.07732 pages. arXiv:2108.07732 [cs.PL]

[4] Yelysei Bondarenko, Markus Nagel, and Tijmen Blankevoort. 2021. Understanding and Overcoming the Challenges of Efficient Transformer Quantization. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 7947–7969. https://doi.org/10.18653/v1/2021.emnlp-main.627

[5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *arXiv e-prints*, Article arXiv:2107.03374 (July 2021), arXiv:2107.03374 pages. arXiv:2107.03374 [cs.LG]

[7] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr

Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2022. PaLM: Scaling Language Modeling with Pathways. *arXiv e-prints*, Article arXiv:2204.02311 (April 2022), arXiv:2204.02311 pages. arXiv:2204.02311 [cs.CL]

[8] Kaustubh D Dhole, Varun Gangal, Sebastian Gehrmann, Aadesh Gupta, Zhenhao Li, Saad Mahamood, Abinaya Mahendiran, Simon Mille, Ashish Srivastava, Samson Tan, et al. 2021. Nl-augmenter: A framework for task-sensitive natural language augmentation. *arXiv preprint arXiv:2112.02721* (2021).

[9] Javid Ebrahimi, Anyi Rao, Daniel Lowd, and Dejing Dou. 2017. Hotflip: White-box adversarial examples for text classification. *arXiv preprint arXiv:1712.06751* (2017).

[10] Michael P Fay and Michael A Proschan. 2010. Wilcoxon-Mann-Whitney or t-test? On assumptions for hypothesis tests and multiple interpretations of decision rules. *Statistics surveys* 4 (2010), 1.

[11] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[12] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. InCoder: A Generative Model for Code Infilling and Synthesis. *arXiv e-prints*, Article arXiv:2204.05999 (April 2022), arXiv:2204.05999 pages. arXiv:2204.05999 [cs.SE]

[13] Ji Gao, Jack Lanchantin, Mary Lou Soffa, and Yanjun Qi. 2018. Black-Box Generation of Adversarial Text Sequences to Evade Deep Learning Classifiers. In *2018 IEEE Security and Privacy Workshops (SPW)*. 50–56. https://doi.org/10.1109/SPW.2018.00016

[14] OpenAI GitHub. 2022. GitHub Copilot. https://copilot.github.com Last accessed 31 August 2022.

[15] Karan Goel, Nazneen Rajani, Jesse Vig, Samson Tan, Jason Wu, Stephan Zheng, Caiming Xiong, Mohit Bansal, and Christopher Ré. 2021. Robustness gym: Unifying the nlp evaluation landscape. *arXiv preprint arXiv:2101.04840* (2021).

[16] Wenjuan Han, Liwen Zhang, Yong Jiang, and Kewei Tu. 2020. Adversarial Attack and Defense of Structured Prediction Models. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Online, 2327–2338. https://doi.org/10.18653/v1/2020.emnlp-main.182

[17] Peter Henderson, Jieru Hu, Joshua Romoff, Emma Brunskill, Dan Jurafsky, and Joelle Pineau. 2020. Towards the Systematic Reporting of the Energy and Carbon Footprints of Machine Learning. *J. Mach. Learn. Res.* 21, 1, Article 248 (jan 2020), 43 pages.

[18] Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. 2020. TinyBERT: Distilling BERT for Natural Language Understanding. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 4163–4174. https://doi.org/10.18653/v1/2020.findings-emnlp.372

[19] Di Jin, Zhijing Jin, Joey Tianyi Zhou, and Peter Szolovits. 2020. Is bert really robust? a strong baseline for natural language attack on text classification and entailment. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 34. 8018–8025.

[20] Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. 2019. Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems* 32 (2019).

[21] François Lagunas, Ella Charlaix, Victor Sanh, and Alexander Rush. 2021. Block Pruning For Faster Transformers. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 10619–10629. https://doi.org/10.18653/v1/2021.emnlp-main.829

[22] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. 2022. CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning. *arXiv e-prints*, Article arXiv:2207.01780 (July 2022), arXiv:2207.01780 pages. arXiv:2207.01780 [cs.LG]

[23] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Online, 7871–7880. https://doi.org/10.18653/v1/2020.acl-main.703

[24] Zhenhao Li and Lucia Specia. 2019. Improving neural machine translation robustness via data augmentation: Beyond back translation. *arXiv preprint arXiv:1910.03009* (2019).

[25] Zheng Li, Zijian Wang, Ming Tan, Ramesh Nallapati, Parminder Bhatia, Andrew Arnold, Bing Xiang, and Dan Roth. 2022. DQ-BART: Efficient Sequence-to-Sequence Model via Joint Distillation and Quantization. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. 203–211.

[26] Chin-Yew Lin and Franz Josef Och. 2004. ORANGE: a Method for Evaluating Automatic Evaluation Metrics for Machine Translation. In *COLING 2004: Proceedings of the 20th International Conference on Computational Linguistics*. COLING, Geneva, Switzerland, 501–507. https://aclanthology.org/C04-1072

[27] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, et al. 2018. Deepgauge: Multi-granularity testing criteria for deep learning systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 120–131.

[28] George A Miller. 1995. WordNet: a lexical database for English. *Commun. ACM* 38, 11 (1995), 39–41.

[29] John X Morris, Eli Lifland, Jin Yong Yoo, Jake Grigsby, Di Jin, and Yanjun Qi. 2020. Textattack: A framework for adversarial attacks, data augmentation, and adversarial training in nlp. *arXiv preprint arXiv:2005.05909* (2020).

[30] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. A Conversational Paradigm for Program Synthesis. *arXiv e-prints*, Article arXiv:2203.13474 (March 2022), arXiv:2203.13474 pages. arXiv:2203.13474 [cs.LG]

[31] David Patterson, Joseph Gonzalez, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. 2021. Carbon Emissions and Large Neural Network Training. *arXiv e-prints*, Article arXiv:2104.10350 (April 2021), arXiv:2104.10350 pages. arXiv:2104.10350 [cs.LG]

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

[33] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research* 21, 140 (2020), 1–67. http://jmlr.org/papers/v21/20-074.html

[34] Shuhuai Ren, Yihe Deng, Kun He, and Wanxiang Che. 2019. Generating Natural Language Adversarial Examples through Probability Weighted Word Saliency. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Florence, Italy, 1085–1097. https://doi.org/10.18653/v1/P19-1103

[35] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *Advances in Neural Information Processing Systems* 33 (2020), 20601–20611.

[36] Charbel Sakr, Yongjune Kim, and Naresh Shanbhag. 2017. Analytical Guarantees on Numerical Precision of Deep Neural Networks. In *Proceedings of the 34th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 70)*, Doina Precup and Yee Whye Teh (Eds.). PMLR, 3007–3016. https://doi.org/10.5555/3305890.3305992

[37] Roy Schwartz, Jesse Dodge, Noah A Smith, and Oren Etzioni. 2020. Green ai. *Commun. ACM* 63, 12 (2020), 54–63.

[38] Amane Sugiyama and Naoki Yoshinaga. 2019. Data augmentation using back-translation for context-aware neural machine translation. In *Proceedings of the Fourth Workshop on Discourse in Machine Translation (DiscoMT 2019)*. 35–44.

[39] Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. 2020. MobileBERT: a Compact Task-Agnostic BERT for Resource-Limited Devices. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Online, 2158–2170. https://doi.org/10.18653/v1/2020.acl-main.195

[40] Chaofan Tao, Lu Hou, Wei Zhang, Lifeng Shang, Xin Jiang, Qun Liu, Ping Luo, and Ngai Wong. 2022. Compression of Generative Pre-trained Language Models via Quantization. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 4821–4836. https://doi.org/10.18653/v1/2022.acl-long.331

[41] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering*. 303–314.

[42] Fusheng Wang, Jianhao Yan, Fandong Meng, and Jie Zhou. 2021. Selective Knowledge Distillation for Neural Machine Translation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Association for Computational Linguistics, Online, 6456–6466. https://doi.org/10.18653/v1/2021.acl-long.504

[43] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Online and Punta Cana, Dominican Republic, 8696–8708. https://doi.org/10.18653/v1/2021.emnlp-main.685

[44] Ziheng Wang, Jeremy Wohlwend, and Tao Lei. 2020. Structured Pruning of Large Language Models. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Online, 6151–6162. https://doi.org/10.18653/v1/2020.emnlp-main.496

[45] Mengzhou Xia, Zexuan Zhong, and Danqi Chen. 2022. Structured Pruning Learns Compact and Accurate Models. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 1513–1528. https://doi.org/10.18653/v1/2022.acl-long.107

[46] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. 2023. SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models. In *Proceedings of the 40th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). 38087–38099. https://proceedings.mlr.press/v202/xiao23c.html

[47] Zhewei Yao, Reza Yazdani Aminabadi, Minjia Zhang, Xiaoxia Wu, Conglong Li, and Yuxiong He. 2022. ZeroQuant: Efficient and Affordable Post-Training Quantization for Large-Scale Transformers. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. 27168–27183. https://proceedings.neurips.cc/paper_files/paper/2022/file/adf7fa39d65e2983d724ff7da57f00ac-Paper-Conference.pdf

[48] Yuan Zang, Fanchao Qi, Chenghao Yang, Zhiyuan Liu, Meng Zhang, Qun Liu, and Maosong Sun. 2020. Word-level Textual Adversarial Attacking as Combinatorial Optimization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Online, 6066–6080. https://doi.org/10.18653/v1/2020.acl-main.540

[49] Jie M Zhang, Mark Harman, Lei Ma, and Yang Liu. 2020. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering* (2020).

[50] Wei Zhang, Lu Hou, Yichun Yin, Lifeng Shang, Xiao Chen, Xin Jiang, and Qun Liu. 2020. TernaryBERT: Distillation-aware Ultra-low Bit BERT. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Online, 509–521. https://doi.org/10.18653/v1/2020.emnlp-main.37

[51] Yunxiang Zhang, Liangming Pan, Samson Tan, and Min-Yen Kan. 2022. Interpreting the Robustness of Neural NLP Models to Textual Perturbations. In *Findings of the Association for Computational Linguistics: ACL 2022*. 3993–4007.

Received 2023-02-02; accepted 2023-07-27