

Quadapter: Adapter for GPT-2 Quantization

Stanford CS224N Default Project

Wesley Bian
ICME
Stanford University
wbian@stanford.edu

Ethan Cohen
ICME
Stanford University
ethancoh@stanford.edu

Abstract

Large Language Models (LLMs) are powerful but expensive to run because they require substantial memory and computation. Quantization can reduce these costs by using lower-precision numbers, but it may also hurt accuracy, especially in transformer models such as GPT-2.

In this project, we build a GPT-2 pipeline for several CS224N tasks and study quantization on Quora Question Pairs (QQP) paraphrase detection and sonnet generation. On QQP, we compare dynamic quantization, a harder simulated static-style baseline, and a Quadapter-inspired method[1] with learnable scaling. On sonnet generation, we compare dynamic quantization and a Quadapter-based quantization-aware model using chrF on held-out prompts.

Our results show that dynamic quantization causes a small accuracy drop, while harsher quantization leads to much larger degradation. Quadapter recovers a substantial part of this loss, and its effectiveness depends strongly on where scaling modules are inserted inside GPT-2.

1 Key Information to include

- Mentor: Simon Kim
- External Collaborators (if you have any): No
- Sharing project: No
- Late Days (if you are using any): No

2 Introduction

A common way to reduce the cost of LLMs deployment is *quantization*, which stores weights and activations at lower numerical precision (for example, 8-bit integers instead of 32-bit floating point numbers). This reduces memory usage and can also speed up inference.

Applying quantization to transformer models, however, is not always straightforward. Transformer hidden states can contain unusually large activation values (*outliers*) that are hard to represent accurately at low precision. This creates two main sources of information loss: multiple distinct floating-point values may be rounded to the same low-precision integer value, and very large activations may be clipped or saturated by the limited quantized range. When outliers force the quantizer to cover a wide range of values, much of the available precision is effectively wasted on a small number of channels. As a result, even small numerical approximations can noticeably affect model behavior. In this project, we study this efficiency–performance trade-off using GPT-2 on two CS224N default project tasks.

Our work has three parts. First, we establish full-precision GPT-2 baselines for SST, CFIMDB, Quora Question Pairs, and sonnet generation. Second, we implement Quadapter on QQP and sonnet generation and compare it with dynamic quantization and simulated static quantization. Third, we

conduct an ablation study showing that Quadapter performance depends strongly on architectural choices, especially the insertion points of its scaling blocks.

Our main question is whether quantization-aware scaling can recover performance lost under low-precision inference across both classification and generation tasks, while still preserving the practical efficiency benefits of quantization.

3 Related Work

Our work builds on prior research on transformer language models and model compression through quantization.

Transformer language models. Our base architecture follows GPT-2, a decoder-only transformer with causal self-attention designed for autoregressive language modeling.

Quantization of neural networks. Quantization reduces the memory and computation cost of neural networks. Two common approaches are post-training quantization (PTQ), which quantizes a pretrained model without retraining, and quantization-aware training (QAT), which simulates quantization during training to adapt the model to quantization noise. QAT often achieves better accuracy but requires training data and may overfit to the fine-tuning dataset.

Quantization challenges for transformers. Quantizing transformer models is difficult because hidden activations often contain large outlier values, which increase quantization error [2]. Prior work shows that such activation outliers create large inter-channel variance and make transformer models harder to quantize than other architectures [3]. As a result, quantization research on autoregressive transformers such as GPT-2 remains relatively limited compared to encoder models like BERT.

Parameter-efficient adaptation and quantization. Adapters have been proposed as a parameter-efficient way to adapt large pretrained models by training only a small set of additional parameters while keeping the original weights fixed [4]. Building on this idea, Quadapter [1] introduces learnable channel-wise scaling parameters that make activations more quantization-friendly. These parameters reduce the effect of activation outliers and can be folded into neighboring linear layers at inference time. Our work follows this direction by evaluating quantization and implementing a Quadapter-inspired scaling approach for GPT-2 on paraphrase detection and sonnet generation tasks it has never been applied to before, and also by performing an original ablation study.

4 Approach

4.1 Full-Precision GPT-2 Baseline

Our base model follows **GPT-2 Small** [5]. We use pretrained initialization and our own implementation of the main GPT-2 components in `modules/attention.py`, `modules/gpt2_layer.py`, and `models/gpt2.py`, optimized with a custom AdamW implementation in `optimizer.py`.

We use lightweight task-specific heads on top of the shared GPT-2 backbone:

- **Sentiment classification:** final-token representation $\rightarrow$ dropout $\rightarrow$ linear classifier (`classifier.py`).
- **Paraphrase detection:** final-token representation $\rightarrow$ binary linear head (`paraphrase_detection.py`).
- **Sonnet generation:** GPT-2 language modeling head with temperature and top- p sampling (`sonnet_generation.py`).

This full-precision model serves as our main performance reference.

For sonnet generation, we improved both training and decoding. We masked padded targets in the loss using `ignore_index=-100`, so the model was trained only on real sonnet tokens. We also froze the word and position embeddings during finetuning, added early stopping based on held-out validation loss, and switched from top- p to top- k sampling. After hyperparameter tuning, we selected $k = 100$ and temperature = 0.9 to reduce both repetition and gibberish.

4.2 Simulated Static-Style Quantization Baseline

Our main low-precision baseline on QQP uses `simulated_static_quantization.py`. This is our own experimental setup. We simulate INT8 quantization in two ways: we replace floating-point backbone weights by 8-bit rounded approximations, and during inference we apply the same quantize–dequantize step to tensors passed through the GPT-2 backbone. The model still runs in floating point, but these operations inject the approximation error expected under low-precision inference. More concretely, for a given tensor we compute a quantization scale from its magnitude, map values into an INT8 range, round them to nearby integers, and then dequantize them back to floating point before continuing the forward pass.

This gives us a controlled and deliberately harder quantized setting than dynamic quantization. Because the approximation is applied broadly across the backbone, it creates a demanding test case for semantic robustness on QQP. We therefore use it as our main low-precision reference when evaluating whether Quadapter can recover more of the full-precision model’s performance.

4.3 Dynamic Quantization Comparison

Then, we apply **post-training dynamic INT8 quantization** to the QQP model using `torch.quantization.quantize_dynamic`. The idea is to take a model that has already been trained in full precision and convert some of its linear layers to use 8-bit integers at inference time. This is done *after* training, so the model is not retrained to adapt to quantization.

In general, quantization replaces a floating-point value x by a lower-precision integer value q using a scale s and a zero-point z :

$$q = \text{round}\left(\frac{x}{s}\right) + z.$$

Intuitively, quantization replaces each floating-point value x with a nearby integer q using a scale s and zero-point z . This reduces model size and can speed up inference, but the approximation may also introduce accuracy loss.

Dynamic quantization stores weights in low precision while computing activation ranges on the fly during inference. This makes it a simple and post-training baseline, which has less storage and compute requirements than the full precision model, but more storage and compute used than the statically quantized model. We use it as a comparison to our harder simulated static-style baseline, which uses fixed quantization scales and introduces a more severe form of quantization error.

4.4 Quadapter

Our main extension is inspired by **Quadapter** [1], but adapted to our GPT-2 codebase and project setting rather than being a reproduction of the original method. This paper addresses a key challenge in transformer quantization: activation outliers.

Quadapter introduces a learnable channel-wise scaling vector $\alpha \in \mathbb{R}^d$, forming a diagonal matrix

$$A = \text{diag}(\alpha),$$

to rescale activations before quantization and undo the scaling afterward. For two consecutive affine layers,

$$y = W_2(W_1x + b_1) + b_2,$$

the transformed computation becomes

$$\hat{y} = Q_{\theta_2}(W_2A^{-1}) Q_{\theta_a}\left(Q_{\theta_1}(AW_1)x + Ab_1\right) + b_2,$$

where Q_{θ_1} and Q_{θ_2} are weight quantizers and Q_{θ_a} is an activation quantizer. Intuitively, the learned scaling redistributes magnitude across channels so that outliers do not dominate the quantization range, making hidden activations easier to represent at low precision and reducing clipping and quantization error. A useful property of this design is that the scaling and inverse scaling can be folded into neighboring linear layers at inference time, so the method introduces no extra runtime module after deployment.

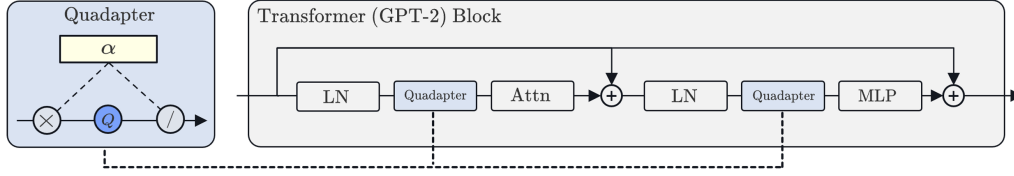


Figure 1: Illustration of the Quadapter mechanism and its placement inside a GPT-2 transformer block, adapted from [1]

4.5 Implementation of the Extension

We implement and wrote original code for an original Quadapter-inspired version of this idea in our GPT-2 codebase. In particular, we designed our own wrapper logic, chose where to insert the scaling modules inside GPT-2 blocks, integrated simulated quantization into our existing forward pass, and evaluated several training variants under our own simulated static-style quantization setting. Our implementation includes:

- Learnable channel-wise scaling parameters for selected GPT-2 block activations,
- Simulated quantization wrappers for weights and activations during training,
- Integration of these wrappers into the GPT-2 forward pass,
- Two-phase settings: Phase 1 calibration minimizing MSE loss between full precision and simulated quantized hidden states while setting quantization scales. Phase 2 continued learning of quadapter block parameters using calibrated quantization scales from Phase 1, minimizing paraphrase classification objective.

In our experiments, we compare training schedules, optimizer choices, and insertion points for the Quadapter modules. These comparisons are also part of our experimental contribution. A key implementation improvement was to move these components from the ends of GPT-2 layers to internal locations within each transformer block, where quantization error is introduced more directly. This substantially improved dev accuracy.

4.6 Sonnet Quantization Extension

We also extend this quantization-aware approach to sonnet generation. Our goal is to test whether a Quadapter-style method can preserve generation quality in an autoregressive setting.

Starting from a pretrained full-precision sonnet model, we decided to freeze the GPT-2 backbone and replace each transformer block with a Quadapter-enabled block. During training, the model exposes both the original hidden states and their fake-quantized counterparts so that quantization-aware losses can be applied.

Training is performed in two phases in `quadapter_sonnet_train.py`. In Phase 1, we minimize MSE between full-precision and simulated-quantized hidden states on a held-out split of the sonnet training set. In Phase 2, we fine-tune the model on the standard next-token prediction objective with padded targets masked using `ignore_index=-100`. We evaluate generation quality with chrF on held-out prompts.

We also evaluate a dynamic quantized sonnet model as a simpler post-training baseline, allowing us to compare post-training quantization with our own quantization-aware adaptation method on a generative task.

5 Experiments

5.1 Data

We use the provided CS224N default project splits.

- SST-5 [6]: 8,544 train / 1,101 dev / 2,210 test examples, with text and 5 sentiment classes.
- CFIMDB [7]: 1,707 train / 245 dev / 488 test, movie reviews with binary sentiment classes used for sentiment classification task.
- Quora (QQP) [8]: 283,003 train / 40,429 dev / 80,858 test examples, pairs of Quora questions and binary paraphrase labels.
- Sonnets: 131 training sonnets, with held-out prompt sets for dev/test generation.

5.2 Evaluation method

For SST-5, CFIMDB, and QQP, we report accuracy on dev sets. For sonnet generation, we report corpus-level chrF [9] on held-out dev prompts, including both full-sonnet and continuation-only scoring.

5.3 Experimental details

All experiments used the provided GPT-2 project codebase with our implementations in `modules/`, `models/`, and the task-specific training scripts. Unless otherwise noted, the backbone was **GPT-2 Small** with hidden size $d = 768$, $l = 12$ transformer layers, and $h = 12$ attention heads. We fixed the random seed to 11711 for reproducibility.

Compute environment We used Modal for the main GPU experiments: `modal_runner.py` for sentiment classification on a T4, `modal_run_paraphrase.py` for paraphrase detection on an A100, and `modal_run_sonnet.py` for sonnet generation on an A100. From Modal logs, the approximate wall-clock training times were: SST $\approx$ 19m 29s, CFIMDB $\approx$ 24m 00s, and Quora paraphrase $\approx$ 4h 42m. For longer Quadapter and quantization experiments, we also used FarmShare scripts. Quadapter training time was 4 hours, 41 minutes, and 4 seconds, with 94 percent of the time spent on Phase 2.

Sentiment Classification (`classifier.py`) We trained GPT-2 sentiment classifiers on SST and CFIMDB. The model uses the GPT-2 final-token representation, followed by dropout and a linear classification head. In this report, the main full-model fine-tuning runs use 10 epochs, batch size 8, and learning rate $1e-5$. We also ran last-linear-layer fine-tuning for comparison. Training used cross-entropy loss and AdamW.

Optimization (`optimizer.py`) Both `classifier.py` and `paraphrase_detection.py` used our custom AdamW optimizer. We set

$$\beta_1 = 0.9, \quad \beta_2 = 0.999, \quad \epsilon = 10^{-6},$$

with bias correction enabled (`correct_bias=True`).

Paraphrase Detection (`paraphrase_detection.py`) We trained a GPT-2 paraphrase detector on QQP using a linear head over the final-token representation for binary paraphrase classification. The baseline reported here uses full-model fine-tuning with GPT-2 small, 10 epochs, batch size 8, and learning rate $1e-5$. Training used cross-entropy loss and AdamW with weight decay set to 0.

As our own initiative, we also tested several small training and representation changes in `paraphrase_detection.py`. In particular, we replaced final-token pooling by masked mean pooling over the last hidden states, added gradient clipping with max norm 1.0, and lowered the default learning rate from $1e-5$ to $5e-6$. In our experiments, these changes did not improve the QQP dev score over the main baseline, so we retained the original setup for the results reported here.

Quantization Experiments For QQP, we evaluated several low-precision settings. Our main hard baseline is `simulated_static_quantization.py` (Section 4.2). We also evaluated `dynamic_quantization.py`, as described in Section 4.3. Our main method is a Quadapter-inspired approach implemented in `quadapter_train.py` and `quadapter_test.py` (Section 4.4).

Sonnet Generation (`sonnet_generation.py`) We trained GPT-2 autoregressively on Shakespeare sonnets using next-token language modeling with token-level cross-entropy. In the updated run, we masked padded target positions in the loss using `ignore_index=-100`, so training only

penalized predictions on real target tokens. We then generated held-out continuations with nucleus sampling. The main generation settings were 15 epochs, learning rate $1e-5$, batch size 8, temperature 0.9, and top- p 0.9, evaluated on `data/sonnets_held_out_dev.txt`. We also evaluated quantization on sonnet generation, as described in Section 4.6.

5.4 Results

Task	Setting	Accuracy
SST-5	Last-linear	0.4587
SST-5	Full-model	0.5068
CFIMDB	Last-linear	0.8571
CFIMDB	Full-model	0.9183

Table 1: Dev-set classification results.

We see that full model performance beats the performance of last-linear finetuning as expected.

Leaderboard numbers (test):

- Paraphrase test leaderboard score: $0.866 > 0.856$ staff baseline
- Sonnet test leaderboard score: $41.637 > 41.208$ staff baseline

QQP quantization setting	Dev accuracy
Full-precision baseline	0.896
Dynamic INT8 quantization	0.880
Quadapter	0.672
Simulated static quantization	0.541

Table 2: QQP dev accuracy for intermediate quantization settings

From Table 2, we see that Quadapter is able to recover about 33 percent of the performance gap caused by static quantization while adding minimal overhead storage and compute requirement overhead to the statically quantized model. While dynamic quantization does perform closely to the full-precision baseline, it has more compute and storage requirements when compared to Quadapter. This supports our main intuition that quantization-aware scaling is most effective where quantization error is introduced.

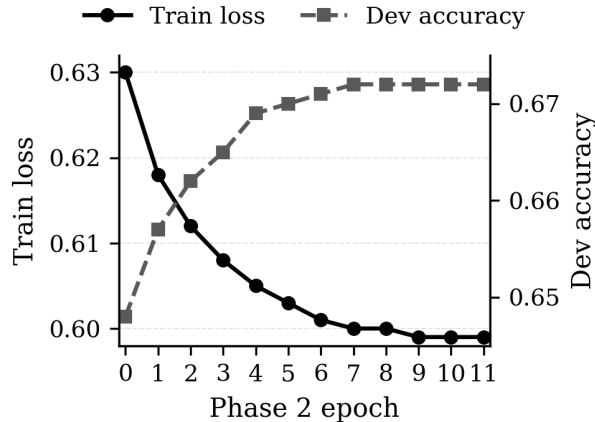


Figure 2: Phase 2 training dynamics of our best Quadapter QQP model under full simulated static quantization

Quadapter Block Placement	Training Phases	Optimizer	Dev Accuracy
Layer End	Phase 1 and 2	Custom AdamW	0.632
Layer End	Phase 2 only	Custom AdamW	0.636
Layer End	Phase 2 only	PyTorch AdamW	0.634

Table 3: QQP dev accuracy for our original Quadapter ablation study

Table 3 shows the results of our Quadapter ablation study, an original idea we came up with after our original novel implementation of Quadapter on the Quora paraphrase detection dataset. From this, it is clear that insertion location of Quadapter blocks is the dominant factor in improving accuracy, and without the correct insertion, the other factors do not make a significant impact.

Sonnet setting	chrF
Full-precision baseline	42.696
Quadapter sonnet	41.465
Dynamic quantized sonnet	39.869

Table 4: Sonnet generation quality on held-out dev prompts

For sonnet generation, Quadapter outperforms the simpler dynamic quantized baseline, reaching a dev chrF of 41.465 versus 39.869. After the study on QQP, we were not necessarily expecting Quadapter to outperform dynamic quantization. We believe this suggests that the benefit of Quadapter is task-dependent: dynamic quantization is already strong on QQP, while sonnet generation appears more sensitive to quantization error over long autoregressive decoding steps.

6 Analysis

For QQP, dynamic quantization lowers overall accuracy from 0.896 to 0.880. More precisely, non-duplicate (class 0) accuracy drops from 0.913 to 0.874, while duplicate (class 1) accuracy rises from 0.867 to 0.889. This suggests that quantization shifts the decision boundary toward more duplicate predictions. The intermediate quantization ablations further show that the damage is highly location-dependent.

For sonnet generation, outputs usually preserve the prompt style in the early lines but lose coherence and metrical quality over longer continuations. Sonnet 4 starts as:

If thy soul check thee that I come so near, Swear to thy blind soul that I was thy will, And will, thy soul knows, is admitted there.

Full precision continuation: Then, if thou wonder with make, To thy unsprung spirit stand send; For then now cannot my lovely union endure thy loss,...

Quadapter continuation: umbled, father me pleased, slaveishish commandments reomp me youYouess, Beautypr is course wont w ath which it to'd eyes it a her see many his slave, that ...

Dynamic quantization continuation: His common stretched me and to you being; Then sick thy crea with bough that you didn himself'd mine injury till soft land did, Cette ghouls' smiles born with widens thy blindness whit body hath quit thy thoughts,

This example shows how Quadapter can retain the themes of love ("Beautypr") and will ("slaveishish commandments") that the full precision and original sonnet touch on, showing that it stays within the same stylistic neighborhood even if the output coherence is necessarily diminished due to the quantization process. On the other hand, the output of dynamic quantization lacks the original themes.

The contrast between QQP and sonnet generation suggests that the value of quantization-aware adaptation is task-dependent. On QQP, the model makes a single final classification decision, and dynamic quantization already provides strong performance, leaving limited room for improvement. On sonnet generation, by contrast, the model generates autoregressively over many steps, so small distortions in hidden representations can accumulate and affect output quality more strongly. This may explain why Quadapter is more helpful on sonnet generation than on QQP in our experiments.

7 Conclusion

In this project, we built a GPT-2 pipeline for the CS224N default tasks and studied the effect of quantization on downstream performance. We established full-precision baselines for sentiment classi-

fication, paraphrase detection, and sonnet generation, and compared a standard dynamic quantization baseline with a harsher simulated static-style quantization setting.

Our main technical contribution was an original implementation and adaption of Quadapter to new tasks. Through ablation experiments, we found that module placement is the key factor for performance recovery. Applying scaling at internal transformer activations, where quantization error is introduced, is substantially more effective than placing it only at the end of transformer blocks. Under our simulated static-style setting, Quadapter recovered part of the lost performance, although it did not match the stronger dynamic quantization baseline on QQP.

Our results also suggest that the usefulness of quantization-aware scaling is task-dependent. On QQP, dynamic quantization already performs strongly, leaving limited room for improvement. On sonnet generation, where hidden-state errors can accumulate, Quadapter better preserves generation quality.

This work highlights two key limitations: First, the Quadapter requires task-specific fine-tuning to achieve performance recovery. This dependency limits its 'plug-and-play' utility, as the method cannot be applied to new tasks without additional training iterations. Second, the modest accuracy gains on QQP suggest that current Quadapter architectures may be limited by their expressive capacity (rank) when tasked with correcting non-linear quantization noise. Future work could explore higher-rank adapters or hybrid approaches that combine quantization-aware training with per-channel scaling to further close the gap between low-precision inference and full-precision performance. Overall, we demonstrate that while aggressive quantization poses a significant challenge to GPT-2, quantization-aware module insertion offers a viable path toward efficient, low-precision deployment.

Team contributions

Wesley: Implementation of base GPT2 model, Quadapter implementation, Quadapter Ablation, quantization on QQP, report writing.

Ethan: Implementation of base GPT2 model, SST-5 classification task, Quora paraphrase detection, sonnet generation, quantization on sonnet generation, report writing.

References

- [1] Minseop Park, Jaeseong You, Markus Nagel, and Simyung Chang. Quadapter: Adapter for gpt-2 quantization. In *Findings of ACL: EMNLP*, pages 2510–2517, 2022.
- [2] Ritchie Zhao, Yuwei Hu, Jordan Dotzel, Christopher De Sa, and Zhiru Zhang. Improving neural network quantization without retraining using outlier channel splitting. In *Proceedings of the 36th International Conference on Machine Learning (ICML)*, 2019.
- [3] 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 (EMNLP)*, 2021.
- [4] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morroni, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for nlp. In *Proceedings of the 36th International Conference on Machine Learning (ICML)*, 2019.
- [5] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8):9, 2019.
- [6] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *EMNLP*, 2013.
- [7] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150, Portland, Oregon, USA, June 2011. Association for Computational Linguistics.
- [8] Quora. First quora dataset release: Question pairs, 2017. Used via CS224N Project subset.
- [9] Maja Popović. chrF: character n-gram f-score for automatic mt evaluation. In *WMT*, 2015.