TorchAO: PyTorch-Native Training-to-Serving Model Optimization

Andrew Or [1] **Apurva Jain** [1] **Daniel Vega-Myhre** [1] **Jesse Cai** [1] **Charles David Hernandez** [1] **Zhenrui Zhang** [1] **Driss Guessous** [1] **Vasiliy Kuznetsov** [1] **Christian Puhrsch** [1] **Mark Saroufim** [1] **Supriya Rao** [1] **Thien Tran** [2] **Aleksandar Samardžić** [3]

Abstract

We present TorchAO, a PyTorch-native model optimization framework leveraging quantization and sparsity to provide an end-to-end, training-to-serving workflow for AI models. TorchAO supports a variety of popular model optimization techniques, including FP8 quantized training, quantization-aware training (QAT), post-training quantization (PTQ), and 2:4 sparsity, and leverages a novel tensor subclass abstraction to represent a variety of widely-used, backend agnostic low precision data types, including INT4, INT8, FP8, MXFP4, MXFP6, and MXFP8.

TorchAO integrates closely with the broader ecosystem at each step of the model optimization pipeline, from pre-training (TorchTitan (Liang et al., 2024)) to fine-tuning (TorchTune (torchtune, 2025), Axolotl (Axolotl, 2025)) to serving (HuggingFace (Wolf et al., 2019), vLLM (Kwon et al., 2023), SGLang (Zheng et al.), ExecuTorch (executorch, 2025)), connecting an otherwise fragmented space in a single, unified workflow. TorchAO has enabled recent launches of the quantized Llama 3.2 1B/3B (MetaAI, 2025b) and LlamaGuard3-8B models (Inan et al., 2023) and is open-source at `https://github.com/pytorch/ao/`.

1. Introduction

Large Language Models (LLMs) have been at the forefront of content creation, text summarization, chatbots, and code generation, among a wide variety of other use cases. However, such capabilities often require substantial infrastructure, as seen in top-performing models such as Qwen3 (235B parameters) (Qwen3, 2025), DeepSeek-v3 (671B) (Guo et al., 2025), Llama 3.1 (405B) (Grattafiori et al., 2024), and Llama 4 Behemoth (2T) (MetaAI, 2025a).

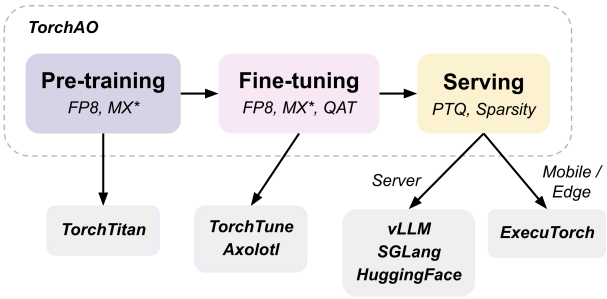

Figure 1. **TorchAO optimization workflow.** TorchAO is closely integrated with popular pre-training, fine-tuning, serving, and model-definition frameworks to provide a seamless, PyTorch-native end-to-end workflow for users to optimize their models. *MX training is a prototype feature.

The computational costs and memory footprint of these models pose significant challenges in every step of the LLM pipeline, from training to fine-tuning to serving. For instance, training Llama 3.1 took 30.84M GPU hours on 16K H100 GPUs (Grattafiori et al., 2024), and even serving the model in its original BF16 precision requires at least 800GB aggregate memory just to fit the model, exceeding the memory limitations of a single server with 8 H100 GPUs. Even at the smaller 1-8B parameter scale, reducing the sizes of these models is important for deploying them in resource-constrained environments such as mobile and edge devices.

However, the existing LLM optimization pipeline is highly fragmented. For instance, a researcher may pre-train their model using mixed FP8/BF16 precision support in Transformer Engine (NVIDIA, 2025), load the pre-trained model into Unsloth (Han & Han, 2025) or Axolotl (Axolotl, 2025) for further fine-tuning, perform quantization using bitsandbytes (Dettmers et al., 2022) before finally serving the model using llama.cpp (GGML, 2025). In each step, the user may need to manually convert the model format (e.g. from HuggingFace's safetensors to GGUF in llama.cpp), and the quantization schemes may diverge from the ones used in previous steps with subtle discrepancies.

[1]Meta Platforms Inc. [2]Independent [3]OpenTeams Inc.. Correspondence to: Andrew Or <andrewor@meta.com>, Supriya Rao <supriyar@meta.com>.

Proceedings of the ICML 2025 Workshop on Championing Open-source Development in Machine Learning (CODEML '25).

FP8 Training*: `convert_to_float8_training(model)`

QAT: `quantize_(model, IntxQuantizationAwareTrainingConfig(...))`

PTQ: `quantize_(model, Int4WeightOnlyConfig())`

Sparsity: `sparsify_(model, SemiSparseWeightConfig())`

Figure 2. **TorchAO APIs.** Users can optimize their models using the above one-line transformations. *API subject to change.

We present TorchAO, a PyTorch-native model optimization framework leveraging quantization and sparsity to provide an end-to-end, training-to-serving workflow for AI models. TorchAO integrates closely with the broader ecosystem at each step of the model optimization pipeline (Figure 1):

- **Pre-training.** TorchAO's FP8 training support (§2.1) composes natively with PyTorch features such as `torch.compile`, autograd, FSDP2, and tensor parallelism support, reaping throughput gains (∼1.5x at 405B scale) (Wright et al., 2024) with virtually no change in model quality when integrated into TorchTitan's pre-training (Liang et al., 2024).

- **Fine-tuning.** Quantization-Aware Training (QAT) is a popular technique for mitigating quantization degradation by simulating quantization numerics during training. TorchAO's QAT support can recover up to 96% of the degradation in quantized accuracy (Or et al., 2024) and can be composed with LoRA (Hu et al., 2022) to improve the training throughput by 1.89x compared to vanilla QAT (Or, 2024; MetaAI, 2025b). Additionally, TorchAO also provides the NF4 data type for QLoRA (Dettmers et al., 2023) to further reduce resource requirements during training.

  TorchTune (torchtune, 2025) natively integrates TorchAO's QAT, NF4, and FP8 training support into its fine-tuning recipes. Axolotl (Axolotl, 2025) also provides fine-tuning workflows that leverage TorchAO QAT, including one that composes QAT with Direct Preference Optimization (DPO) (Rafailov et al., 2023).

- **Serving.** TorchAO provides flexible support for a wide variety of backend agnostic Post-Training Quantization (PTQ) schemes for model optimization before serving. For server backends, TorchAO's PTQ support is integrated as an optional quantization transformation before serving in SGLang (PyTorch et al., 2024) or vLLM (vLLM, 2025). For edge and mobile backends, users can lower their TorchAO quantized models in ExecuTorch (ExecuTorch, 2025; MetaAI, 2025b), which provides lightweight runtimes with static memory planning to reduce performance and power overheads, and serve their models on-device using our custom quantized ARM CPU and metal kernels (torchao, 2025a).

```python
from torchao.quantization import (
  Int4WeightOnlyConfig,
)
from transformers import (
  AutoModelForCausalLM,
  TorchAoConfig,
)
config = Int4WeightOnlyConfig()
config = TorchAoConfig(quant_type=config)
mod = AutoModelForCausalLM.from_pretrained(
  "meta-llama/Llama-3.2-3B",
  device_map="auto",
  torch_dtype=torch.bfloat16,
  quantization_config=config,
)
mod.push_to_hub(
  f"{user_id}/Llama-3.2-3B-int4",
  safe_serialization=False,
)
```

Listing 1. **TorchAO as a quantization backend in HuggingFace.** TorchAO quantization can be applied to any model on HuggingFace Hub through the `TorchAoConfig`. Users can then serialize and deserialize their quantized models and upload them to HuggingFace Hub at ease with native HuggingFace APIs: `save_pretrained`, `load_pretrained`, and `push_to_hub`.

TorchAO is also closely integrated with HuggingFace Transformers (Wolf et al., 2019) and Diffusers (von Platen et al.), two popular model-definition frameworks for state-of-the-art machine learning models (Listing 1). More specifically, TorchAO is natively integrated as one of the quantization backends in HuggingFace, enabling users to optionally apply post-training quantization when loading their models from the HuggingFace Hub. Our integration also supports serialization through native HuggingFace APIs such as `save_pretrained`, `load_pretrained`, and `push_to_hub`, further enabling users to seamlessly save and load their quantized models for future inference or generation on their desired serving frameworks.

Alternatively, users can simply call TorchAO's one-line APIs directly to optimize their models (Figure 2). For instance, users may wish to use a custom training loop or a training framework not yet integrated with TorchAO for FP8 pre-training or QAT fine-tuning. For more detailed usage, please refer to Appendix B.

In the ensuing sections, we will walk through two end-to-end example workflows that leverage different model optimization techniques in TorchAO targeting different backends. In the first workflow, we will use FP8 training and inference as our unifying theme and target server GPUs as the serving backend (§2). In the second workflow, we will leverage QAT to mitigate quantization degradation and lower our model to the XNNPACK (Google, 2025) backend, which provides optimized ARM kernels used by the most popular Android and iOS phones (§3).

```
# Pre-training in FP8 with TorchTitan
torchtitan/run_train.sh
  --training.compile
  --model.converters="float8"

# Fine-tuning in FP8 with TorchTune
tune run --nnodes 1 --nproc_per_node 8
  full_finetune_distributed
  --config llama3.1/8B_full
    enable_fp8_training="true"

# Upload FP8 model to HuggingFace hub
# (Optional, not shown)

# Serve FP8 model through vLLM
vllm serve
  <HF_USER_ID>/Llama-3.1-8B-Instruct-fp8
  --tokenizer meta-llama/Llama-3.1-8B
```

Listing 2. **Example end-to-end FP8 flow.** The user leverages TorchAO's FP8 training support to pre-train and fine-tune their model using TorchTitan and TorchTune respectively, then (optionally) uploads their trained model to HuggingFace hub and serves it using vLLM. Detailed FP8 training options are omitted for brevity.

2. Workflow: FP8 Targeting Server GPUs

The first end-to-end workflow leverages TorchAO's FP8 training support to pre-train and fine-tune the model using TorchTitan and TorchTune respectively, and then serves the trained model in vLLM using the same FP8 configurations.

Typically, pre-training is a time consuming step that trains on large, general datasets like C4 (allenai, 2025), while fine-tuning is a separate step that adapts the pre-trained model to more domain specific tasks. In this example workflow, both training steps leverage TorchAO's dynamic FP8 training with tensorwise scaling to speed up the training process. In each step of the workflow, users may optionally upload their pre-trained or fine-tuned checkpoints to HuggingFace hub for others in the community to continue fine-tuning the models or directly use them for inference (Listing 2).

2.1. FP8 Training

TorchAO's FP8 training dynamically casts activations, weights, and gradients to FP8 and leverages specialized GEMM kernels to take advantage of the FP8 tensor cores (NVIDIA, 2023) in the underlying GPUs. This technique is most useful for training models in which the majority of GEMM operations are large enough such that the speedup achieved by using FP8 tensor cores is greater than the overhead of dynamic quantization (Appendix C).

We support three FP8 training scaling recipes, each with different throughput/accuracy trade-offs: tensorwise, rowwise, and rowwise with high precision `grad_weight` (see Appendix A for full recipe details). Combined with `torch.compile`, tensorwise scaling achieved training throughput speedups of up to 1.5x at 405B parameter scale on 512 H100 GPUs (Wright et al., 2024), and rowwise scaling achieved speedups of up to 1.43x at 70B parameter scale on 1920 H200 GPUs (Wright et al., 2025), compared to training with BF16.

Quantization	Output token throughput (tok/s)	Time per output token (ms)	Inter-token latency (ms)
None (BF16)	103.6 (+0%)	9.50 (+0%)	9.47 (+0%)
FP8 tensorwise	132.8 (+28.2%)	7.48 (-21.2%)	7.47 (-21.1%)

Table 1. **Serving FP8 pre-trained and fine-tuned Llama3.1-8B on vLLM.** Serving this model in FP8 vs in the original precision (BF16) saw a 28% increase in throughput and a 21% reduction in latency. Clients in both experiments used the ShareGPT dataset and number of prompts = 1.

TorchAO's FP8 training leverages the tensor subclass abstraction (PyTorch, 2025) to compose with PyTorch autograd and PyTorch distributed. Our FP8 training support is integrated into both TorchTitan and TorchTune, so all of the above scaling recipes can be used for pre-training and fine-tuning with minimal setup (Listing 2).

For further throughput gains, users can leverage asynchronous tensor parallelism (Wang et al., 2024b), a compiler driven approach to overlapping the compute and communications in tensor parallel, via using SymmetricMemory APIs (Wang et al., 2024a) for copy-engine based communications instead of SM-based ones, and micro-pipelining the computations. TorchAO's FP8 tensorwise and rowwise recipes are composable with asynchronous tensor parallelism, yielding up to 17% additional training throughput improvement, depending on factors like model size and activation checkpointing strategy (torchtitan, 2025).

2.2. Post-Training Optimization

Similar to FP8 training, both post-training quantization (PTQ) and sparsity support in TorchAO leverage PyTorch's tensor subclass abstraction to provide native composability with other PyTorch features and seamless serialization support. TorchAO's PTQ supports a wide range of popular data types, including INT4, INT8, FP8, MXFP4, MXFP6, and MXFP8, and can be lowered to efficient kernels across different backends such as CUDA and ARM CPU. TorchAO's PTQ is also integrated with GemLite (gemlite, 2025) to leverage specialized Triton kernels (triton, 2025) to further speed up inference by 1.1-2x across different batch sizes and tensor parallel sizes (PyTorch et al., 2024).

TorchAO's sparsity support accelerates inference by leveraging hardware support for sparse matrix multiplications offered by modern NVIDIA GPUs (Mishra et al., 2021). Our benchmarks have shown up to 1.3x speedup with rela-

tive model accuracy of 91-100% on ViT models compared to the non-sparse baseline. TorchAO provides APIs and kernels for different sparsity techniques including sparse marlin 2:4 (Frantar et al., 2025), 2:4 sparsity, block sparsity, INT8 dynamic quantization + 2:4 sparsity, and rowwise FP8 + 2:4 sparsity for weights and activations (Haziza et al., 2025).

For detailed quantization and sparsity benchmarks, refer to the respective READMEs (torchao, 2025c;d).

2.3. Serving

TorchAO's PTQ support is closely integrated with popular model serving frameworks such as vLLM (Kwon et al., 2023) and SGLang (Zheng et al.) as one of the quantization backends. In particular, our FP8 inference support uses the same configurations as FP8 training to provide consistent end-to-end numerics across different steps in the workflow. Initial benchmarks on serving Llama3.1-8B in FP8 vs in the original precision (BF16) demonstrates a 28% increase in throughput and 21% reduction in latency (Table 1).

3. Workflow: QAT Targeting Mobile Devices

The second end-to-end workflow leverages TorchAO's Quantization-Aware Training (QAT) support to fine-tune the model using TorchTune, and then lowers the model to the XNNPACK backend using ExecuTorch, enabling the model to be accessed on mobile phones and wearable devices such as smart glasses.

3.1. Quantization-Aware Training

QAT refers to inserting "fake" quantization operations into the model, which simulate the quantization process during training. This allows the model to learn to be robust to quantization errors, improving the accuracy of the model when it is ultimately quantized post-training. TorchAO offers simple and flexible QAT APIs that allows users to specify different quantization schemes (e.g., INT8 dynamic activations with INT4 weights), and provides corresponding PTQ configurations to ensure end-to-end numerical consistency (see Listing 7 in Appendix B for a detailed example).

TorchAO's QAT support is integrated into TorchTune's and Axolotl's fine-tuning workflows, enabling effortless fine-tuning of a model that is intended to be quantized (Listing 3). During fine-tuning, all "fake" quantization operations are still performed in high precision (e.g. BF16) even though they simulate low precision numerics (e.g. INT4). The resulting QAT checkpoint retains the exact same model structure as the original checkpoint, and so can be used as a drop-in replacement that offers superior post-training quantized accuracy with identical inference speeds. TorchTune additionally provides a recipe that composes QAT with LoRA (Hu et al., 2022) to reduce the overheads of the extra

```
# Fine-tuning using torchtune's QAT recipe
tune run --nnodes 1 --nproc_per_node 4
  qat_distributed
  --config llama3/8B_qat_full

# Lower model to executorch
python -m
    examples.models.llama.export_llama
  --checkpoint <checkpoint.pth>
  --params <params.json>
  --use_sdpa_with_kv_cache
  --xnnpack
  --use_kv_cache
  --embedding-quantize 4,32
  --quantization-mode 8da4w

# Serve on mobile device
# Build llama runner and run on Android
adb shell "cd /data/local/tmp/llama &&
    ./llama_main --model_path <model.pte>
    --tokenizer_path <tokenizer.model>
    --prompt 'What is the capital of
    France?' --seq_len 120" --warmup=1
```

Listing 3. **Example end-to-end QAT Targeting Mobile Devices.** The user leverages TorchTune's QAT and PTQ support to fine-tune and quantize Llama3.1-8B, then exports the model using ExecuTorch and serves it as an INT4 model on CPU. Detailed export flags are omitted for brevity.

"fake" quantization operations, yielding an 1.89x improvement in training throughput compared to vanilla QAT (Or, 2024; MetaAI, 2025b).

Recent launches of the quantized Llama 3.2 1B/3B (MetaAI, 2025b) and LlamaGuard3-8B models (Inan et al., 2023) leveraged TorchAO's QAT support to mitigate quantization degradation in their INT4 checkpoints targeting the ARM CPU backend. This resulted in a 2-4x inference speedup, 56% reduction in model size, and 41% reduction in memory usage compared to the original BF16 checkpoints, while maintaining competitive performance across a wide variety of inference tasks.

3.2. Lowering to Edge

Models quantized using TorchAO's QAT or PTQ APIs can be easily lowered to edge backends using ExecuTorch, which provides conversions that are adapted to natively match TorchAO's quantization patterns (Listing 3). This enables efficient deployment of these models on various edge backends, including Android, iOS, and CoreML. TorchAO also provides sub-8-bit ARM CPU and Metal kernels for quantized linear and embedding operations that can be used in eager model execution, with `torch.compile`, with PyTorch AOTInductor, or in the exported model. For detailed instructions on how to export TorchAO's quantized models to run on mobile backends, refer to the ExecuTorch README (executorch, 2024).

Model	Quantized hellaswag accuracy	Quantized wikitext word perplexity	Training throughput (tok/s)	Training peak memory (GB)
Llama3-8B	47.0% (57.1% BF16)	26.270 (9.422 BF16)	480.3 (+0%)	17.6 (+0%)
Llama3-8B (QAT)	52.8% (recovered 57.8%)	12.312 (**recovered 82.8%**)	323.0 (-32.7%)	32.9 (+86.8%)
Llama3.1-8B	51.8% (57.9% BF16)	18.628 (9.164 BF16)	492.4 (+0%)	17.7 (+0%)
Llama3.1-8B (QAT)	55.5% (recovered 60.0%)	10.901 (recovered 81.6%)	323.0 (-34.4%)	33.0 (+86.5%)
Llama3.2-3B	46.8% (51.7% BF16)	17.461 (12.051 BF16)	1408.8 (+0%)	13.8 (+0%)
Llama3.2-3B (QAT)	50.2% (**recovered 69.8%**)	13.220 (recovered 78.4%)	737.7 (-47.6%)	14.5 (+5.24%)

Table 2. **Quantization-Aware Training (QAT) on Llama3 models, fine-tuned on OpenAssistant Conversations Dataset (OASST1) (OpenAssistant, 2025).** For this workload, QAT can recover up to 69.8% of the degradation in quantized hellaswag accuracy and 82.8% of the degradataion in quantized wikitext word perplexity.

Scaling	Peak Mem (GB)	Median tok/s	Speedup
None (BF16)	47.65	6150	1.0
tensorwise + FP8 all-gather	47.77	7689.5	1.25
rowwise + BF16 all-gather	47.79	6768	1.10

Table 3. **FP8 pre-training on Llama3-8B using TorchTitan.** Tensorwise scaling with FP8 all-gather achieves 1.25x faster training throughput on this model with on par memory usage.

Quantization Technique	Acc	Word perplexity	Tput (tok/s)	Model size (GB)
None	60.01	7.33	132.41	15.01
int4wo-64	58.10	8.25	268.88	4.76
int8wo	59.92	7.34	216.38	8.04
float8wo	59.83	7.37	213.88	8.03
float8dq (PerRow)	59.86	7.41	167.13	8.04
float8dq (PerTensor)	59.95	7.42	176.44	8.03

Table 4. **Post-training quantization (PTQ) on Llama3.1-8B**. Quantization reduced model size by 2-4x and increased inference throughput by up to 2x with minimal quantization degradation. All experiments use a batch size of 1 with `torch.compile`.

4. Evaluation

In this section, we evaluate TorchAO's FP8 training, post-training quantization (PTQ), and quantization-aware training (QAT) support. All experiments are performed on 1-8 H100 GPUs, each with 96GB of HBM3 memory.

FP8 training. We benchmark training Llama3-8B on the C4 dataset (allenai, 2025) on 8x H100 GPUs for 100 steps using TorchTitan. All experiments use a batch size of 1, a sequence length of 8192, `torch.compile`, and per op selective activation checkpointing (SAC). For this workload, tensorwise scaling combined with FP8 all-gather operations achieved a speedup of 1.25x over the BF16 baseline with on par peak memory usage (Table 3) and virtually identical loss curves (Appendix D).

Post-training quantization (PTQ). We quantize Llama3.1-8B across a variety of PTQ settings and evaluated the quantized models on the hellaswag and wikitext tasks on a single H100 GPU using a batch size of 1 with `torch.compile`. PTQ reduced the model size by 2-4x and increased the inference throughput by up to 2x, while mostly maintaining parity on hellaswag accuracy and wikitext word perplexity across all quantization settings compared to the baseline BF16 model (Table 4).

Quantization-aware training (QAT). At lower precisions such as 4-bits, quantization degradation from PTQ alone is more pronounced and QAT becomes more effective. We evaluate QAT by fine-tuning Llama3-8B, Llama3.1-8B, and Llama3.2-3B on the OpenAssistant Conversations Dataset (OpenAssistant, 2025) on 4 H100 GPUs for 1000 steps using TorchTune. All experiments use a batch size of 8, a learning rate of 2e-5, a weight quantization group size of 32, and activation checkpointing. For this workload, QAT was able to recover up to 69.8% of the degradation in quantized hellaswag accuracy and 82.8% of the degradation in quantized wikitext word perplexity (Table 2).

5. Conclusion

In this paper, we introduced TorchAO, a PyTorch-native model optimization framework that is closely integrated into each step of the pre-training, fine-tuning, and serving lifecycle of LLMs. TorchAO supports popular model optimization techniques, including FP8 training, QAT, PTQ, and 2:4 sparsity, and targets a variety of backends including server CPU/GPU and mobile/edge. We welcome your contributions at `https://github.com/pytorch/ao/`.

Acknowledgement

We express our gratitude towards our many open-source contributors and collaborators, including but certainly not limited to Hicham Badri, Sayak Paul, Aryan V S, Marc Sun, Matthew Douglas, Salman Mohammadi, Lianmin Zheng, Michael Goin, Daniel Han, Peter Yeh, Jeff Daily, Pawan Jayakumar, Jerome Ku, Tobias van der Werff, Vaishnavi Gupta, Andreas Köpf, Diogo Venâncio, Leslie Fang, Weiwen Xia, Yi Liu, Xuan Liao, Yanbing Jiang, Xiao Wang,

Sanchit Jain, Hengyu Meng, Devang Choudhary, Ethan Petersen, Martin Cala, Chip Smith, Scott Roy, Digant Desai, Kimish Patel, Manual Candales, Mengwei Liu, Songhao Jia, Chen Lai, Zeyu Song, Yanan Cao, Andrey Talman, Huy Do, Catherine Lee, Svetlana Karslioglu, Evan Smothers, Will Feng, Will Constable, Ke Wen, Tianyu Liu, Gokul Nadathur, Tristan Rice, Shuqi Yang Lisa Jin, Zechun Liu, Tijmen Blankevoort, Hanxian Huang, Luca Wehrstedt, Daniel Haziza, Timothy Chou, Dhruv Choudhary, Francisco Massa, Jiecao Yu, Geonhwa Jeong, Patrick Labatut, Josh Fromm, Less Wright, and Hamid Shojanazeri. We also thank Joe Isaacson, Jane Xu, and Scott Roy for your early feedback on the paper.

References

allenai. C4 dataset. https://huggingface.co/datasets/allenai/c4, 2025.

Axolotl. Axolotl: Open source fine tuning. https://axolotl.ai/, 2025.

Badri, H. and Shaji, A. Half-quadratic quantization of large machine learning models. https://mobiusml.github.io/hqq_blog/, November 2023.

Dettmers, T., Lewis, M., Belkada, Y., and Zettlemoyer, L. Llm.int8(): 8-bit matrix multiplication for transformers at scale. *arXiv preprint arXiv:2208.07339*, 2022.

Dettmers, T., Pagnoni, A., Holtzman, A., and Zettlemoyer, L. Qlora: Efficient finetuning of quantized llms. *Advances in neural information processing systems*, 36:10088–10115, 2023.

executorch. Run model on android phone. https://github.com/pytorch/executorch/blob/main/examples/models/llama/README.md#step-4-run-benchmark-on-android-phone, 2024.

executorch. Executorch: End-to-end solution for enabling on-device inference capabilities across mobile and edge devices. https://github.com/pytorch/executorch, 2025.

ExecuTorch. export_llama_lib. https://github.com/pytorch/executorch/tree/main/examples/models/llama, 2025.

Frantar, E., Castro, R. L., Chen, J., Hoefler, T., and Alistarh, D. Marlin: Mixed-precision auto-regressive parallel inference on large language models. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pp. 239–251, 2025.

gemlite. Gemlite: Triton kernels for efficient low-bit matrix multiplication. https://github.com/mobiusml/gemlite, 2025.

GGML. Llama.cpp: Inference of meta's llama model (and others) in pure c/c++. https://github.com/ggml-org/llama.cpp, 2025.

Google. Xnnpack. https://github.com/google/XNNPACK, 2025.

Grattafiori, A., Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Vaughan, A., et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Guo, D., Yang, D., Zhang, H., Song, J., Zhang, R., Xu, R., Zhu, Q., Ma, S., Wang, P., Bi, X., et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Han, D. and Han, M. Unsloth: Open source fine-tuning for llms. https://unsloth.ai/, 2025.

Haziza, D., Chou, T., Choudhary, D., Wehrstedt, L., Massa, F., Yu, J., Jeong, G., Rao, S., Labatut, P., and Cai, J. Accelerating transformer inference and training with 2:4 activation sparsity. *arXiv preprint arXiv:2503.16672*, 2025.

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., Chen, W., et al. Lora: Low-rank adaptation of large language models. *ICLR*, 2022.

Inan, H., Upasani, K., Chi, J., Rungta, R., Iyer, K., Mao, Y., Tontchev, M., Hu, Q., Fuller, B., Testuggine, D., et al. Llama guard: Llm-based input-output safeguard for human-ai conversations. *arXiv preprint arXiv:2312.06674*, 2023.

Jin, L., Ma, J., Liu, Z., Gromov, A., Defazio, A., and Xiao, L. Parq: Piecewise-affine regularized quantization. *arXiv preprint arXiv:2503.15748*, 2025.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023.

Liang, W., Liu, T., Wright, L., Constable, W., Gu, A., Huang, C.-C., Zhang, I., Feng, W., Huang, H., Wang, J., et al. Torchtitan: One-stop pytorch native solution for production ready llm pre-training. *arXiv preprint arXiv:2410.06511*, 2024.

Lin, J., Tang, J., Tang, H., Yang, S., Chen, W.-M., Wang, W.-C., Xiao, G., Dang, X., Gan, C., and Han, S. Awq: Activation-aware weight quantization for on-device llm compression and acceleration. *Proceedings of Machine Learning and Systems*, 6:87–100, 2024.

Liu, Z., Zhao, C., Fedorov, I., Soran, B., Choudhary, D., Krishnamoorthi, R., Chandra, V., Tian, Y., and Blankevoort, T. Spinquant: Llm quantization with learned rotations. *arXiv preprint arXiv:2405.16406*, 2024.

Liu, Z., Zhao, C., Huang, H., Chen, S., Zhang, J., Zhao, J., Roy, S., Jin, L., Xiong, Y., Shi, Y., et al. Paretoq: Scaling laws in extremely low-bit llm quantization. *arXiv preprint arXiv:2502.02631*, 2025.

MetaAI. The llama 4 herd: The beginning of a new era of natively multimodal ai innovation. https://ai.meta.com/blog/llama-4-multimodal-intelligence/, 2025a.

MetaAI. Introducing quantized llama models with increased speed and a reduced memory footprint. https://ai.meta.com/blog/meta-llama-quantized-lightweight-models/, 2025b.

Mishra, A., Latorre, J. A., Pool, J., Stosic, D., Stosic, D., Venkatesh, G., Yu, C., and Micikevicius, P. Accelerating sparse deep neural networks. *arXiv preprint arXiv:2104.08378*, 2021.

NVIDIA. Nvidia h100 tensor core gpu architecture. https://resources.nvidia.com/en-us-hopper-architecture/nvidia-h100-tensor-c, 2023.

NVIDIA. Using fp8 with transformer engine. https://docs.nvidia.com/deeplearning/transformer-engine/user-guide/examples/fp8_primer.html#Using-FP8-with-Transformer-Engine, 2025.

OpenAssistant. Openassistant conversations dataset. https://huggingface.co/datasets/OpenAssistant/oasst1, 2025.

Or, A. Speeding up qat by 1.89x with lora. https://dev-discuss.pytorch.org/t/speeding-up-qat-by-1-89x-with-lora/2700, 2024.

Or, A., Zhang, J., Smothers, E., Khandelwal, K., and Rao, S. Quantization-aware training for large language models with pytorch. https://pytorch.org/blog/quantization-aware-training/, 2024.

PyTorch. Subclassing torch.tensor. https://docs.pytorch.org/docs/stable/notes/extending.html#subclassing-torch-tensor, 2025.

PyTorch, MobiusLabs, and SGLang. Accelerating llm inference with gemlite, torchao and sglang. https://pytorch.org/blog/accelerating-llm-inference/, 2024.

Qwen3. Qwen3: Think deeper, act faster. https://qwenlm.github.io/blog/qwen3/, 2025.

Rafailov, R., Sharma, A., Mitchell, E., Manning, C. D., Ermon, S., and Finn, C. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 36: 53728–53741, 2023.

Rouhani, B. D., Zhao, R., More, A., Hall, M., Khodamoradi, A., Deng, S., Choudhary, D., Cornea, M., Dellinger, E., Denolf, K., et al. Microscaling data formats for deep learning. *arXiv preprint arXiv:2310.10537*, 2023.

Safaryan, M. and Richtárik, P. Stochastic sign descent methods: New algorithms and better theory. In *International Conference on Machine Learning*, pp. 9224–9234. PMLR, 2021.

torchao. Torchao: Low-bit arm cpu and metal kernels for linear and embedding ops. https://github.com/pytorch/ao/tree/main/torchao/experimental, 2025a.

torchao. Torchao prototypes. https://github.com/pytorch/ao/tree/main/torchao/prototype, 2025b.

torchao. Torchao post-training quantization llama3 benchmarks. https://github.com/pytorch/ao/tree/main/torchao/quantization#benchmarks, 2025c.

torchao. Torchao sparsity llama3 benchmarks. https://github.com/pytorch/ao/tree/main/torchao/sparsity#llama3, 2025d.

torchtitan. Torchtitan async tp benchmarks. https://github.com/pytorch/torchtitan/blob/main/benchmarks/asyncTP_llama3_h100_2025-06_torchtitan.md, 2025.

torchtune. Torchtune: Native pytorch library for llm fine-tuning. https://github.com/pytorch/torchtune, 2025.

triton. Triton: Language and compiler for writing highly efficient custom deep-learning primitives. https://github.com/triton-lang/triton, 2025.

vLLM. Torchao documentation. https://docs.vllm.ai/en/latest/features/quantization/torchao.html, 2025.

von Platen, P., Patil, S., Lozhkov, A., Cuenca, P., Lambert, N., Rasul, K., Davaadorj, M., Nair, D., Paul, S., Liu, S., Berman, W., Xu, Y., and Wolf, T. Diffusers: State-of-the-art diffusion models. URL https://github.com/huggingface/diffusers.

Wang, Y., He, H., and Wehrstedt, L. Pytorch symmetricmemory: Harnessing nvlink programmability with ease. https://dev-discuss.pytorch.org/t/pytorch-symmetricmemory-harnessing-nvlink-programmability-with-ease/2798, 2024a.

Wang, Y., He, H., Wright, L., Wehrstedt, L., Liu, T., and Liang, W. Introducing async tensor parallelism in pytorch. https://discuss.pytorch.org/t/distributed-w-torchtitan-introducing-async-tensor-parallelism-in-pytorch/209487, 2024b.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., et al. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019.

Wright, L., Feng, W., Kuznetsov, V., and Guesseous, D. e. a. Training using float8 with fsdp2. https://pytorch.org/blog/training-using-float8-fsdp2/, 2024.

Wright, L., Shojanazeri, H., Kuznetsov, V., Vega-Myhre, D., Nadathur, G., Constable, W., Liu, T., Rice, T., Guessous, D., Fromm, J., Wehrstedt, L., Yu, J., Petersen, E., Cala, M., and Smith, C. Accelerating large scale training and convergence with pytorch float8 rowwise on crusoe 2k h200s. https://pytorch.org/blog/accelerating-large-scale-training-and-convergence-with-pytorch-float8-rowwise-on-crusoe-2k-
2025.

Xiao, G., Lin, J., Seznec, M., Wu, H., Demouth, J., and Han, S. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning*, pp. 38087–38099. PMLR, 2023.

Zhao, J., Zhang, Z., Chen, B., Wang, Z., Anandkumar, A., and Tian, Y. Galore: Memory-efficient llm training by gradient low-rank projection. *arXiv preprint arXiv:2403.03507*, 2024.

Zheng, L., Yin, L., Xie, Z., Sun, C., Huang, J., Yu, C. H., Cao, S., Kozyrakis, C., Stoica, I., Gonzalez, J. E., et al. Sglang: Efficient execution of structured language model programs. *https://arxiv.org/abs/2312.07104*.

A. Appendix: TorchAO FP8 Scaling Recipes

TorchAO supports the following FP8 training recipes, which have different performance/accuracy trade-offs:

- **tensorwise**: This is the default recipe, with reasonable performance/accuracy trade-offs. It computes a single scaling factor for each tensor. This technique has the lowest overhead and highest performance, but is more sensitive to outliers since a single outlier anywhere in the tensor will affect the scaling of the entire tensor. This can cause higher quantization error as more values may underflow to 0. When training with FSDP, tensorwise scaling also supports an additional optimization `enable_fp8_all_gather` which will perform the all-gathers in FSDP using FP8 to reduce communication overhead.

- **rowwise**: This recipe trades off a bit of performance for better accuracy. It computes scaling factors along logical rows of the left GEMM operand, and along logical columns of the right GEMM operand. Computing scaling factors for more granular slices of the tensors reduces sensitivity to outliers, improving accuracy but with a performance cost compared to tensorwise.

- **rowwise_gw_hp**: This recipe is like `rowwise` but it keeps the $\frac{\partial L}{\partial W}$ computation in bfloat16, as experiments have shown this to be more sensitive to lower precision, and keeping it in higher precision is better for accuracy. This recipe can also achieve higher speedups than `rowwise` in models where the majority of GEMMs are small than M == N == K = 13k.

B. Appendix: TorchAO APIs

```
torch.compile(model)
convert_to_float8_training(model)
```

Listing 4. **TorchAO FP8 Training Example.**

```
# INT4 weight-only, targeting tinygemm cuda kernel
quantize_(model, Int4WeightOnlyConfig(group_size=32))

# OR INT8 dynamic activation + INT4 weight, targeting XNNPACK
quantize_(model, Int8DynamicActivationInt4Weight(group_size=32))

# OR FP8 dynamic activation + FP8 weight, targeting hopper GPUs and beyond
quantize_(model, Float8DynamicActivationFloat8WeightConfig())

# etc.
```

Listing 5. **TorchAO PTQ Example.**

```
model = model.cuda()

# Sparse Marlin 2:4
quantize_(model, Int4WeightOnlyConfig(layout=MarlinSparseLayout()))

# OR 2:4 Sparsity
sparsify_(model, SemiSparseWeightConfig())

# OR Block Sparsity
sparsify_(model, BlockSparseWeightConfig())
```

Listing 6. **TorchAO Sparsity Example.**

```
# prepare: insert fake quantization ops
# swaps `torch.nn.Linear` with `FakeQuantizedLinear`
activation_config = FakeQuantizeConfig(torch.int8, "per_token", is_symmetric=False)
weight_config = FakeQuantizeConfig(torch.int4, group_size=32)
qat_config = IntXQuantizationAwareTrainingConfig(activation_config, weight_config),
quantize_(model, qat_config)

# train
train_loop(model)

# convert: transform fake quantization ops into actual quantized ops
# swap `FakeQuantizedLinear` back to `torch.nn.Linear` and inserts
# quantized activation and weight tensor subclasses
quantize_(model, FromIntXQuantizationAwareTrainingConfig())
quantize_(model, Int8DynamicActivationInt4WeightConfig(group_size=32))

# inference or generate (not shown)
```

Listing 7. **TorchAO QAT Example.** TorchAO's QAT flow is separated into two steps, prepare and convert. During the prepare step, insert "fake" quantization operations into the linear and embedding modules in the model to simulate quantization numerics, but do not actually cast the dtypes to lower precision. After training, in the convert step, we replace these "fake" quantization operations and with real quantization operations, using the same code path as TorchAO's regular PTQ flow.

C. Appendix: Float8 Training Microbenchmarks

Observed float8 vs bfloat16 speedup of LayerNorm -> Linear -> Sigmoid fwd+bwd, by forward M, K, N

dynamic per-tensor scaling, NVIDIA H100, PyTorch version 2.5.0.dev20240821+cu121, using GPU kernel time

		speedup				
		N				
M	K	1024	2048	4096	8192	16384
1024	1024	0.77	0.80	0.89	0.94	0.99
	2048	0.77	0.74	0.82	0.94	0.92
	4096	0.83	0.78	0.83	0.93	0.93
	8192	0.83	0.91	1.01	1.07	0.94
	16384	0.78	0.87	0.95	0.92	1.10
2048	1024	0.80	0.83	0.89	0.96	1.01
	2048	0.75	0.86	1.04	1.10	1.09
	4096	0.79	0.98	1.15	1.19	1.18
	8192	0.84	1.02	1.18	1.25	1.24
	16384	0.91	1.02	1.15	1.21	1.35
4096	1024	0.84	0.89	1.00	1.05	1.06
	2048	0.83	0.98	1.12	1.18	1.20
	4096	0.86	1.08	1.26	1.35	1.35
	8192	1.03	1.11	1.28	1.36	1.39
	16384	0.95	1.09	1.26	1.35	1.47
8192	1024	0.92	0.98	1.06	1.08	1.09
	2048	0.87	1.06	1.16	1.30	1.25
	4096	0.91	1.17	1.32	1.35	1.39
	8192	0.97	1.18	1.30	1.44	1.50
	16384	0.92	1.12	1.30	1.46	1.57
16384	1024	0.99	1.03	1.08	1.11	1.14
	2048	0.90	1.03	1.19	1.26	1.29
	4096	0.92	1.13	1.30	1.37	1.44
	8192	0.95	1.19	1.34	1.49	1.51
	16384	1.03	1.21	1.41	1.60	1.80

Figure 3. **Observed float8 vs bfloat16 speedup of LayerNorm + Linear + Sigmoid forward and backward pass, by forward M, N, K** A common question about FP8 training is "when is FP8 linear faster vs BF16?". Given the M, K, N of the forward pass through your linear, you can reference the table below for a microbenchmark based speedup estimate on NVIDIA H100

D. Appendix: Float8 Training Loss Curves

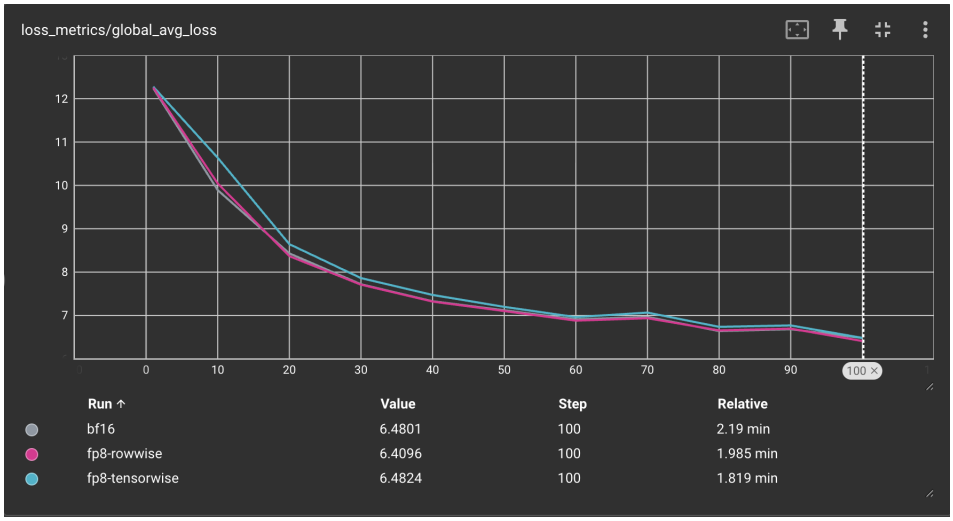

Figure 4. **Observed loss curves for float8 tensorwise and rowwise training with FSDP2 on 8xH100 GPUs, compared with the bfloat16 baseline.**

E. Appendix: TorchAO Prototype Features

TorchAO currently has many exciting **prototype** features that are not guaranteed to be suitable for production use yet, but are available for users to experiment with. The list of prototype features is provided below. You can find more details in the TorchAO prototypes folder (torchao, 2025b).

- **AutoRound:** optimize weight rounding via signed gradient descent (Safaryan & Richtárik, 2021)
- **AWQ:** activation-aware weight quantization (Lin et al., 2024)
- **Blockwise FP8:** FP8 blockwise quantization introduced by DeepSeek (Guo et al., 2025)
- **float8nocompile:** accelerating eager FP8 tensorwise training with triton kernels
- **GaLore:** memory-efficient training with gradient low-rank projection (Zhao et al., 2024)
- **HQQ:** half-quadratic quantization (Badri & Shaji, 2023)
- **MoE quantzation:** mixture of experts (MoE) quantization for inference
- **MX formats:** mxfp4, mxfp6, and mxfp8 for training (Rouhani et al., 2023)
- **ParetoQ:** scaling laws in extremely low-bit LLM quantization (Liu et al., 2025)
- **ParQ:** piecewise-affine regular quantization (QAT) (Jin et al., 2025)
- **INT8 quantized training:** quantized INT8 weights or dynamic INT8 quantization
- **scaled_grouped_mm:** differentiable scaled grouped GEMM for MoE FP8 training
- **SmoothQuant:** remove outliers for W8A8 quantization (Xiao et al., 2023)
- **SpinQuant:** quantization with learned rotations (Liu et al., 2024)