tensor-net-lm

Compute Request — ExeaLabs

Daksh Jain · Quantum + AI/ML Researcher — MIT CSAIL, NexEdge, Qinetic Research Labs

---

Compressing LLM weights with Matrix Product States / Tensor Trains, and actually measuring the Pareto frontier instead of guessing at it. Below is the project laid out the way I'd want to read it if I were on the other side of this application.

1 · Problem — what and why

Linear projection layers dominate LLM parameter counts, in GPT-2 alone they're roughly two-thirds of everything, and the two mainstream ways to shrink them are honestly pretty blunt. Pruning needs retraining to claw quality back, and quantization only gives you a handful of fixed compression levels with no way to dial it in. Tensor Train / Matrix Product State decomposition, the same math physicists use to represent quantum many-body systems, gives you one tunable knob (the bond dimension) with a provable error bound attached, and yet nobody's actually measured what that tradeoff looks like on a real, modern transformer with honest latency numbers next to it. That's the gap I'm closing.

2 · Approach — model / method, and why this one

I'm using GPT-2 124M as the primary testbed, small enough to iterate on fast but still a real, widely-understood transformer, and decomposing its MLP and attention projection layers into TT and MPS cores via truncated SVD (the TT-SVD algorithm). Rather than one fixed rank across the whole model, I sweep the bond dimension from 2 to 512 and run a per-layer adaptive rank search, since not every layer compresses the same way. I picked this over pruning or quantization specifically because it's a one-shot post-processing step, no retraining required, the error is provably tied to the singular value spectrum of the weight matrix, and the compression-quality tradeoff is continuous instead of a step function.

3 · Dataset — source, size, preprocessing

This decomposes already-pretrained weights rather than training from scratch, so there's no training set at all, just the pretrained GPT-2 124M weights pulled from HuggingFace in fp16. For evaluation I'm using the WikiText-2 raw test split for perplexity and the HellaSwag validation set for a 0-shot reasoning check. Preprocessing is minimal: standard HuggingFace tokenization, with perplexity computed on a sliding window (stride 512, max length 1024) so long documents don't get truncated and quietly skew the numbers.

4 · Metrics — how I'll know it worked

The main signal is the compression-ratio-vs-perplexity Pareto curve across the full bond-dimension sweep, I'd call it a real win at 4x–8x compression with under 1% perplexity degradation. Alongside that I'm tracking HellaSwag accuracy delta, tokens/sec on CPU and GPU, memory footprint, and per-layer

reconstruction error as a sanity check that the decomposition math itself is behaving. None of that means much sitting on its own though, so I'm running magnitude pruning and INT8 quantization at the exact same compression ratios as baselines, so the tensor-network numbers are actually being measured against something real.

5 · Compute — training time, memory, dependencies

There's no gradient descent here, so time is dominated by SVDs and evaluation passes, not training epochs. The full experiment set, baseline comparison across 5 compression levels and 3 methods, the full 9-point rank sweep, and the adaptive sweep, comes out to a few hours of GPU time end-to-end, and needs a single GPU with at least 6GB VRAM, ideally something in H100 territory so the sweep doesn't eat the whole day. Disk footprint is small, around 5GB for the HuggingFace model cache and results. Dependencies are torch, transformers, opt_einsum, numpy/scipy, datasets, and matplotlib, all already pinned in requirements.txt. This isn't a build problem anymore, it's purely a compute problem, which is exactly what's been capping me so far.

6 · Deliverables — model, paper, demo

First deliverable is the actual compression-perplexity Pareto curve, along with the decomposed model artifacts behind it. Second is a short, honest research write-up documenting what the tradeoff looks like against the pruning and quantization baselines, and if the results end up genuinely compelling, I'd look at shaping that into something submittable as a paper. Third is a clean, documented private repository that can reproduce the main curve from scratch in under an hour on a single GPU.

7 · Risk — what breaks first, and plan B

The thing most likely to break first is perplexity collapsing at low bond dimension, especially if it hits the first or last transformer layers, which tend to be more sensitive than the rest. Plan B is the adaptive per-layer rank search I've already built, so sensitive layers get assigned a higher rank automatically instead of the whole model taking the hit uniformly. Second risk is exact SVD getting slow or memory-heavy on the larger matrices, like the 3072×768 MLP layers, plan B there is randomized SVD (torch.svd_lowrank), which costs basically nothing in quality. Third is the usual dependency drift, HuggingFace or lm-eval changing their API mid-run, so every version is pinned and there's a fallback eval path built in for exactly that. And if compute ends up being the constraint again even after this, plan B is simple: keep results to GPT-2 124M and drop the LLaMA-350M stretch goal, since that was never the core deliverable to begin with.