International Journal of Science and Research Archive

eISSN: 2582-8185

Cross Ref DOI: 10.30574/ijsra

Journal homepage: https://ijsra.net/

(RESEARCH ARTICLE)

Check for updates

Optimizing Large Language Model Deployment in Edge Computing Environments

Raghavan Krishnasamy Lakshmana Perumal *

Member of IEEE Computer Society, Tampa, Florida, USA.

International Journal of Science and Research Archive, 2025, 14(03), 1658-1669

Publication history: Received on 21 February 2025; revised on 29 March 2025; accepted on 31 March 2025

Article DOI: https://doi.org/10.30574/ijsra.2025.14.3.0912

Abstract

Deploying large language models (LLMs) in edge computing environments is an emerging challenge at the intersection of AI and distributed systems. Running LLMs directly on edge devices can greatly reduce latency and improve privacy, enabling real-time intelligent applications without constant cloud connectivity. However, modern LLMs often consist of billions of parameters and require tens of gigabytes of memory and massive compute power, far exceeding what typical edge hardware can provide. In this paper, we present a comprehensive approach to optimize LLM deployment in edge computing environments by combining four existing classes of optimisation techniques: model compression, quantization, distributed inference, and federated learning, in a unified framework. Our insight is that a holistic combination of these techniques is necessary to successfully deploy LLMs in practical edge settings. We also provide new algorithmic solutions and empirical data to advance the state of the art.

Keywords: Large Language Models (LLMs); Edge Computing; Model Compression; Distributed Inference; Federated Learning

1. Introduction

Deploying large language models (LLMs) in edge computing environments is an emerging challenge at the intersection of AI and distributed systems. Running LLMs directly on edge devices (such as smartphones, IoT sensors, or edge servers) can greatly reduce latency and improve privacy, enabling real-time intelligent applications without constant cloud connectivity [1]. Edge-based LLMs avoid sending sensitive user data to the cloud, thus mitigating privacy risks, and they allow personalized on-device adaptation to user behaviors [1]. Despite these advantages, resource constraints at the edge make LLM deployment extremely difficult. Modern LLMs often consist of *billions* of parameters and require tens of gigabytes of memory and massive compute power, far exceeding what typical edge hardware can provide. The self-attention operations in transformers involve intensive tensor computations that are orders of magnitude slower on low-power edge CPUs/NPUs [1]. Simply loading a full LLM can exhaust the RAM of an edge device [1], making naive deployment infeasible. To bridge this gap, the research community has developed a spectrum of optimization techniques for deep models on resource-limited devices. Four classes of techniques have shown particular promise for LLMs on the edge: model compression, quantization, distributed inference, and federated learning.

Each of these techniques tackles a different aspect of the problem – model size, computational cost, device heterogeneity, and data locality. *Model compression (pruning and distillation)* (e.g., [2-5]) and *quantization* (e.g, [6-12]) have been used in conventional deep learning and smaller NLP models. However, applying them to *truly large* language models (with hundreds of millions or billions of parameters) is non-trivial [1]. Aggressive compression can degrade a model's linguistic capabilities if not carefully managed, and unique characteristics of transformers (e.g. attention heads, layer normalization) complicate simple pruning or quantization approaches [1]. *Distributed inference* on the edge introduces overhead from networking and demands intelligent partitioning/scheduling of model parts to avoid bottlenecks [1, 13-15]. *Federated learning* (e.g., [16-17]) with large models is constrained by limited client computing

* Corresponding author: Raghavan Krishnasamy Lakshmana Perumal.

power and communication bandwidth; naive federated training of a 10B+ parameter model is impractical when each edge client has perhaps 4–8 GB of memory. Recent studies highlight that existing FL methods often assume at least some clients can handle the full model or large portions of it, which is rarely true in edge scenarios. These challenges motivate new research into integrated strategies that make LLMs edge-friendly without significant performance loss.

In this paper, we present a comprehensive approach to optimize LLM deployment in edge computing environments by combining model compression, quantization, distributed inference, and federated learning in a unified framework. Our main contributions are summarized as follows: First, we propose a novel deployment framework that compresses and quantizes an LLM, partitions it across edge devices, and enables federated on-device learning for continual improvement. Further, we provide detailed mathematical formulations for each component of the optimization. We analyze the trade-offs introduced by compression (e.g. accuracy vs. model size), quantization (precision vs. error), and distributed inference (latency vs. communication cost). We prove complexity results for the model partitioning problem and discuss convergence guarantees for federated learning in our setting. Then, we develop efficient algorithms to implement the proposed framework. We present pseudocode for key procedures – including the compression & quantization pipeline and the federated distributed inference algorithm – to illustrate how these methods operate in concert. Finally, we conduct extensive experiments on benchmark datasets relevant to edge scenarios. We evaluate on natural language understanding tasks from GLUE [18] (to measure accuracy of compressed models on diverse tasks), the SQuAD question-answering dataset (to test comprehension capability under compression), and a language modeling corpus (WikiText-103) to measure generative perplexity. Our results demonstrate that the proposed optimization strategies enable up to an order-of-magnitude improvement in memory footprint and inference latency, with minimal loss in accuracy compared to an unoptimized baseline.

2. Materials and Methods

2.1. Framework Overview

We assume an edge computing scenario with a set of M edge devices (or edge servers) that are connected via a network (e.g., LAN or 5G). These devices collectively will host and serve an LLM. There is also a central server (or cloud) that coordinates federated learning rounds and can assist with heavy computation if needed (e.g., initial model training). The overall process is divided into two phases. In the *offline phase*, we start with a large pre-trained language model (which could be trained on a cloud with abundant computing). We then apply model compression techniques (pruning, distillation) to reduce its size, followed by quantization to lower precision. This yields a compressed, quantized model that is small and efficient enough for edge deployment. We denote this model as F_θ with parameters θ. In the *online phase*, we deploy F_θ across the M edge devices for inference serving. This involves partitioning the model's parameters (layers) into M shards, $\theta = \theta^{(1)}, \theta^{(2)}, \dots, \theta^{(M)}$, and placing shard $\theta^{(i)}$ on device i. The devices then collaboratively run the model: when an inference request arrives (e.g., a user prompt), the model computation is pipelined through devices 1 to M in sequence (or another scheduling strategy) and the final answer is produced.

Optionally, during deployment, the devices engage in periodic federated learning rounds to further fine-tune the model θ using local data. Each device uses whatever user data it has (e.g., device i has dataset D_i) to compute updates to its portion of the model. These updates are then aggregated (usually at the central server) to improve the global model. The updated global parameters are then redistributed to devices, keeping the distributed model in sync. This process can occur intermittently (e.g., at night or low-load times) to continuously improve the model's performance on edge-specific data distributions.

Our approach can be seen as a pipeline of optimizations: large model → compressed model → quantized model → distributed model across devices → federated fine-tuned model.

2.2. Model Compression Step

The objective was to build a framework that reduces the number of parameters and operations of the pretrained LLM while retaining as much of its original performance as possible. We combine *structured pruning* and *knowledge distillation* for this purpose.

Let the original model be denoted F_Θ with parameter set Θ (e.g., Θ are the weights of all layers of a 100 billion parameter model). We seek a significantly smaller model F_θ (with $\theta \subset \Theta$ or derived from $\Theta\$, *and* $\$|\theta| \ll |\Theta|$) such that $F_\theta(x) \approx F_\Theta(x)$ for inputs x in the tasks of interest.

2.2.1. Pruning formulation

We define a binary mask vector (or set) m of the same dimensionality as Θ (each element $m_w \in 0,1$ indicates whether weight w is kept (1) or pruned (0)). Applying m to the original weights yields pruned weights $\Theta \odot m$. Our goal could be formulated as an optimization: find m that minimizes the loss on the training data *and* satisfies a sparsity constraint (i.e. the number of ones in m is below a threshold K corresponding to desired model size). For example, one can set up: $\min_{m\in\{0,1\}^{|\Theta|}} L_{task}(x; \Theta \odot m)$ s.t. $|m|_0 \leq K$, where L_{task} is the task loss (e.g., cross-entropy on next-word prediction) and $|m|_0$ counts nonzero entries (the size of the pruned model). This optimization is generally combinatorial and intractable for large $|\Theta|$, so heuristic criteria are used (magnitude pruning, gradient-based pruning, etc., as discussed in Related Work). In our approach, we perform *magnitude-based pruning with structure*: we prune entire attention heads and feed-forward hidden units that have the smallest l_2 norm or importance scores, ensuring the pruned model remains a valid transformer. We also prune unimportant layers if needed (but we found it sufficient to prune within layers to reach our target compression ratio). The pruning ratio is chosen to reduce model size by roughly 50% initially.

2.2.2. Knowledge distillation formulation

After pruning, we employ distillation to further compress and to recover performance. We instantiate a *student model* S_θ (with parameters θ initialized from the pruned model or from scratch with smaller size) and use the original large model T_Θ (teacher) to guide its training. The student's architecture could be a scaled-down version of the teacher's (e.g., fewer layers or narrower layers). We train S_θ on a combination of the original training dataset (if available) and unlabeled data (which the teacher can label on the fly). The distillation loss is a weighted sum of the task loss and a distillation loss:

$$L_{distill}(\theta) = E_{x\sim\mathbb{D}\backslash\mathbb{B}ig}\left[\underbrace{\alpha \cdot \mathcal{L}_{ts}(S_\theta(x), y)}_{\text{supervised task loss}} + \underbrace{(1-\alpha)\cdot T^2 \cdot KL\backslash big(p_T(x) \mid p_S(x)\backslash big)}_{\text{distillation loss at temperature T}}\backslash Big\right],$$

where $\mathcal{D}$ is the training data distribution (with input x and ground-truth y when available), $p_T(x)$ is the teacher's output probability distribution (soft logits) for input x, $p_S(x)$ is the student's predicted distribution, KL is the Kullback–Leibler divergence, α is a weighting factor, and T is a temperature hyperparameter to soften the teacher probabilities [1]. We set $T > 1$ typically to give a smoother target distribution, which provides richer information than a one-hot label. We also experiment with *intermediate layer distillation*: matching the student's hidden states to the teacher's (projected to same dimension) for certain layers, as this has been shown to improve distillation of transformers.

Solving $\min_\theta L_{distill}(\theta)$ gives us the student parameters. This is done using stochastic gradient descent (SGD/Adam) on the student model, while the teacher is fixed. The end result is a compact model S_θ that tries to mimic T_Θ. In practice, we obtained a student model that is ~5× smaller than the teacher with only a minor drop in accuracy (consistent with earlier findings that, e.g., DistilBERT retained ~97% of BERT's accuracy, our student retained ~95–96% of the original LLM's performance on evaluation tasks, see Results).

After this compression step, we have a model F_θ that is much smaller (e.g., if starting from a 6B parameter model, we might end with a 1.5B parameter model). We emphasize that compression is done *centrally* (offline) where we have computational resources to run the distillation process.

2.3. Quantization Step

Next, we quantize the compressed model F_θ to further optimize it for edge deployment. Quantization can be done *post hoc* on the trained weights (post-training quantization) and/or during fine-tuning (quantization-aware training). We primarily use post-training quantization for simplicity, with some calibration if needed.

2.3.1. Weight quantization

We represent each weight (a 32-bit float by default) with a lower precision representation. The simplest uniform quantization for an array of weights $W = w_i$ is:

Choose a bit-width b (e.g., $b = 8$ or 4). This gives 2^b quantization levels.

Determine a scale Δ (step size) and possibly zero-point Z for mapping real values to quantized integers. In symmetric uniform quantization (no zero-point), we have $q_i = \text{round}(w_i/\Delta)$ clipped to $\left[-2^{b-1}, 2^{b-1}-1\right]$.

The quantized weight is stored as q_i (an b-bit integer). The dequantization on the fly would compute $\widehat{w_i} = \Delta \cdot q_i$ as the approximation of w_i.

The challenge is to choose Δ (and Z if asymmetric) to minimize the quantization error $\left|\left|W - \widehat{W}\right|\right|$. Commonly, Δ is chosen based on the range of W (max-min) or some percentile of values to ignore outliers. In our implementation, we use an approach inspired by GPTQ for weight quantization: for each linear layer in the transformer, we sample a batch of representative inputs (from the training data or a held-out set) and measure the layer's output error as we quantize weights [19]. We then adjust the quantization scale or even do a one-step weight adjustment to reduce output error. We quantize all linear layer weights to 8-bit integers in our base deployment. We leave LayerNorm and embedding weights in higher precision (since they are small in size and quantizing them can sometimes hurt more). Our quantization procedure ensures that no layer's output divergence exceeds a small threshold (we set allowable degradation per layer to e.g. <0.5% in terms of output norm).

2.3.2. Activation quantization (for inference)

While weights are quantized and fixed, activations (the intermediate outputs) during inference can also be quantized to reduce runtime memory and computation. We use 8-bit activation quantization at inference time, implemented via the ONNX Runtime or PyTorch quantized kernels that essentially simulate int8 matrix multiplication and then dequantize results. To avoid range issues, we calibrate the activation range for each layer by running a few sample inferences and recording the distribution of activation values, then choosing appropriate scales.

2.4. Distributed Inference Strategy

Now we tackle how to deploy $\tilde{F}\theta$ across multiple edge devices for inference. The question is how to partition the model and coordinate the devices. We consider a straightforward partitioning by layers: each device gets a consecutive chunk of the transformer's layers. This respects the model's sequential nature and requires only that devices pass intermediate activations in sequence. Formally, suppose the model has L layers (numbered 1 to L, including attention and feed-forward sublayers, etc.). We choose cut points $1 = k_0 < k_1 < k_2 < \cdots < k_M = L$, and assign layers $[ki - 1, k_i)$ to device i (for $i = 1$ to M, where $k_M = L$ means device M has layers from k_{M-1} to L). Device 1 will take the input, run layers $1..k_1$, then send the output (the activations at layer k_1) to device 2. Device 2 continues until k_2, and so on, until device M produces the final output. This pipelined forward pass yields the result. (For generation tasks, this is done iteratively for each output token – which introduces some additional pipeline latency, but we ignore that detail for now and focus on a single forward pass.)

2.4.1. Partitioning optimization

We wanted to balance the computational load and consider communication. In general, the problem can be formulated as: *given compute times $t_1, \ldots, t_L$ for each layer and output sizes $s_1, \ldots, s_{L-1}$ for each potential cut (size of activation after layer i), choose $M - 1$ cut indices to minimize total latency*. If devices execute in pipeline, and if we allow streaming, the latency can approach the maximum of (compute time on device + communication to next) among the devices (since all can work in parallel on different micro-batches). However, for simplicity, we often consider a single request at a time, in which case the latency is the sum of all device compute plus comms (since each waits for previous to finish sending). In that case the objective is additive: minimize $\sum_{i=1}^{M}$ (compute on device i) + $\sum_{i=1}^{M-1}$ (communication of cut i). This is roughly: minimize $\sum_{j=1}^{L} t_j + \sum_{cuts}$ comm cut.

We want each device's compute time to be as equal as possible, so that they finish nearly simultaneously (so no one is idle waiting too long). We implement a simple heuristic: we measure (or estimate) each layer's execution time on each type of device. In our scenario, the devices were homogeneous (same hardware), so t_j is roughly constant for all devices, but if not, we would account for device speed differences. We then cumulate these times and choose cut points such that the cumulative compute time in each partition is approximately equal.

2.4.2. Inference procedure

Once partitioning is set, inference is straightforward. We ensure all devices are running the same model code (just with different layers active). Pseudocode for the distributed inference process is given in Table 1. We also incorporate *quantized communication*: since our activations are in 8-bit on each device, we can transmit them as 8-bit values as well to reduce bandwidth.

Table 1 Algorithm 1: Distributed Inference Across Edge Devices (pseudocode)

Inputs: Model F_θ with layers 1..L partitioned across M devices.
Devices $D_1, \dots, D_M$ connected in sequence.
Input data (e.g., prompt tokens) x on device D_1.
Output: Model output (e.g., predicted text or logits) y on device D_M.
# On device D1: 	
a0 = x # initial input activations
for layer j = 1 to k1: # layers assigned to D1 a_j = layer_j(a_{j-1}) # compute forward pass through each layer end send a_{k1} to D2 (quantized 8-bit) # transmit output of layer k1 to next device
# On device Di (for i = 2 ... M-1): 	
receive a_{k_{i-1}} from D_{i-1} # input from previous device for layer j = k_{i-1}+1 to k_i: # layers assigned to Di a_j = layer_j(a_{j-1}) end send a_{k_i} to D_{i+1} # send output to next device
# On device DM (last device): receive a_{k_{M-1}} from D_{M-1} for layer j = k_{M-1}+1 to L: 	
a_j = layer_j(a_{j-1}) end y = a_L # final output return y # output is now on device DM (could send back to D1 or user if needed)

This procedure assumes a simple linear pipeline. In practice, one can parallelize by streaming multiple inputs (batch or tokens) so that while D1 processes input 2, D2 is processing input 1, etc., to better utilize all devices. We did not implement a full pipeline parallel scheduler, but the concept is similar. The above pseudocode suffices for functional correctness of single inference.

2.5. Federated Learning Integration

2.5.1. Federated learning models

Finally, we integrate federated learning to allow on-device training across the distributed edge deployment (Table 2). After the model is deployed and serving, we consider that over time, each device may accumulate local data (e.g., user interactions, sensor data, feedback). We leverage this data to improve the model via periodic federated learning rounds. Importantly, even though the model is partitioned for inference, for the purpose of learning, each device can *fine-tune its portion* of the model (the layers it holds). Since each device only has a part of the model, how do we do FL on the whole model? We consider two approaches: whole-model FL with full copies and split-model FL (layer-wise). In our implementation, we choose a middle ground: devices engage in FL for the full model, but sequentially training their part and receiving the rest. Concretely, we orchestrate a federated procedure where each client eventually gets to update all model weights, but not all at once to fit memory. This is done by rotating the model shards or using a coordinator that sends different parts of the model to clients in stages. However, to keep things simpler and consistent, we ended up using a form of federated knowledge distillation: instead of exchanging large weight updates, clients send distilled information (e.g., logits or gradients of outputs) to a server, which updates a global model.

For clarity, we present the FL process as if standard FedAvg is used, but note that in practice we applied some compression to updates. This requires each client to download and upload the full model. In our case, because θ is large, we: (a) use compression on the upload (we quantize the model update, or send only weights differences that are above a threshold, achieving >10× reduction in update size; this is crucial for edge network efficiency), and (b) possibly limit how frequently full model syncs happen (we might do more local epochs to reduce rounds).

Table 2 Algorithm 2: Federated Training Procedure (high-level pseudocode)

```
Initialize global model θ^0 = θ_compressed_quantized  (obtained from Steps 3.2 and 3.3) for each round t = 1 to T:
  Server selects a set of available devices S_t   for each device i in S_t (in parallel):
    Server sends θ^{t-1} (full model or its partition) to device i
    Device i loads θ^{t-1} into local model
    Device i performs Li local SGD updates on θ (for e.g. 1 epoch over local data)
    Device i obtains updated weights θ_i^t
    Device i sends Δθ_i^t (or θ_i^t) back to server (with compression)   Server aggregates updates:
    θ^t = θ^{t-1}  # start from previous       For each i in S_t:          θ^t = θ^t + (n_i / N_{S_t}) * (θ_i^t - θ^{t-1})
(This is equivalent to weighted average of θ_i^t) end for
Distribute final θ^T to all devices (final synchronized model)
```

3. Experiments

We conduct a series of experiments to evaluate the effectiveness of our proposed optimization strategies for LLM edge deployment.

3.1. Experimental Setup

3.1.1. Baseline Model

For our experiments, we use an LLM with an architecture similar to GPT-2 or GPT-J style decoder-only transformer, which we had available as a pretrained model. The model's size (before compression) is approximately 2.7 billion parameters (comparable to a GPT-2 XL). We choose a generative model to test both understanding tasks and generation tasks. This model was pretrained on a generic web corpus (similar to GPT-2 training data). We consider this our *starting large model*. Its size is too large for typical edge devices (requirse over 10GB memory in FP32).

3.1.2. Edge Hardware Simulation

We simulate edge device hardware using two setups: (a) a single-machine with multiple CPU processes to emulate multiple devices (with bandwidth limits enforced to simulate network), and (b) an actual pair of edge devices for a subset of tests (two NVIDIA Jetson AGX Orin boards, each with 32GB memory and an 8-core ARM CPU and a small GPU). The Jetson represents a high-end edge device (like an advanced IoT gateway or a onboard computer in a vehicle). We cap the available memory per device artificially in some tests to simulate less-capable devices (e.g., 8GB). The devices are connected via an Ethernet network (1 Gbps), and we also emulate slower wireless (100 Mbps) in some tests to see effect of comm bandwidth.

3.1.3. Frameworks and Tools

We implement the model in PyTorch 2.0 for both training and inference. For quantization, we use PyTorch's native static post-training quantization tooling for 8-bit, and a custom script for 4-bit (since PyTorch doesn't natively support 4-bit at the time of writing – we integrated an open-source GPTQ implementation). All experiments maintain consistency in using PyTorch on all devices (no mix of TensorFlow or other frameworks) to avoid any inconsistent behaviors. For distributed execution, we use Python's multiprocessing for coordinating the pipeline in simulation, and a lightweight RPC for the real device test. Federated learning is coordinated using Flower 1.2 library, which easily integrates with PyTorch models and allows us to simulate federated rounds on our devices.

3.1.4. Datasets

We evaluate on benchmarks of both natural language understanding (NLU) and generation:

- **GLUE Benchmark** [18]: We select three representative tasks from GLUE to test NLU performance: (i) MNLI (multi-genre natural language inference, a text classification task to predict if a hypothesis is entailed by a premise, which tests language understanding and reasoning), (ii) QNLI (question-natural language inference, similar to QA/NLI task), and (iii) SST-2 (Stanford Sentiment Treebank, sentiment classification). These tasks allow us to measure if our model retains understanding capabilities after compression. We use the standard

train/dev sets from GLUE, but note our model was pretrained (not specifically fine-tuned on these), so we *fine-tune* on these tasks under different model settings (baseline vs compressed vs compressed+federated, etc.) to compare the achievable accuracy.

- **SQuAD v1.1:** The Stanford Question Answering Dataset, a reading comprehension benchmark where a model must answer questions from given passages. We evaluate our model's F1 score on SQuAD to see if the compressed model can still perform extractive QA. We fine-tune the model on the SQuAD training set for a few epochs in each scenario and measure dev set score.
- **WikiText-103 (WT103):** A large corpus of Wikipedia articles commonly used to evaluate language modeling (measuring perplexity). We use WT103 as a test for generative performance: we compute perplexity of the model on this corpus (without fine-tuning on it, just as a zero-shot evaluation of language modeling ability). Perplexity is sensitive to the model's overall distributional knowledge, which can be affected by quantization and pruning. We expect some increase in perplexity after optimization, and we measure how much.
- **Edge-specific dataset (Federated Setting):** To test federated learning in a realistic setting, we construct a scenario with non-IID user data. We use the Reddit Comments dataset (a large corpus of Reddit discussions) as a proxy for personal conversational data. We partition this dataset by author to simulate each user's device having its own conversation history. We sample 50 users, each with a few thousand comments. The task here is next sentence prediction or response generation. We don't have a straightforward accuracy metric for generation, so we use perplexity on each user's data and qualitative evaluation. The idea is to see if federated fine-tuning on this decentralized data helps the model adapt (for example, using FL to teach the model slang or topics specific to those users).

3.1.5. Baselines for Comparison:

- **Cloud-only (no optimization) Baseline:** The original 2.7B model running on a single high-end GPU (NVIDIA A100) with full precision. This represents the upper-bound accuracy and the scenario with no edge optimizations (but obviously not feasible on real edge).
- **Edge Single-Device Baseline:** The model after compression & quantization but running entirely on one edge device. This will highlight the benefit of distribution by comparing to our multi-device deployment.
- **Smaller Model Baseline:** We also compare to a totally different smaller model: DistilBERT (66M params) fine-tuned on the same tasks. While DistilBERT is not an LLM for generation, it provides a point of reference for NLU tasks to see if our method's compressed LLM can outperform a purpose-built small model.
- **Ablations:** We compare variants: (a) *Compression+Quantization only* (no distributed, no FL), (b) *+Distributed inference* (distributed vs single device), (c) *+Federated* (with FL fine-tuning vs without, on the Reddit user data scenario). This isolates the contribution of each component.

3.1.6. We measure a variety of metrics

- *Accuracy/F1:* For GLUE classification tasks (accuracy) and SQuAD (F1 score).
- *Perplexity:* For WT103 language modeling (lower is better).
- *Model size and memory:* Model file size on disk, and peak runtime memory usage on device.
- *Latency:* Time to generate an output (for a fixed length input) on the edge devices. For distributed, we measure end-to-end latency including communication; for single device, just local compute time.
- *Throughput:* If applicable, e.g., queries per second the system can handle (though for simplicity we focus on single query latency due to the sequential nature of LLM output).
- *Communication overhead:* amount of data transferred per inference (to ensure it's not huge).
- *Federated training rounds:* we track the model's validation accuracy on a global validation set after each FL round, to see the improvement.

We run each experiment 3 times (with different random seeds or data shuffles) and report averaged results to account for variance in fine-tuning.

4. Results

4.1. Model Compression & Quantization Effect

After applying our compression (pruning + distillation) pipeline, the model size (in parameters) was reduced by ~4× from 2.7B to about 700M. In terms of storage, the 2.7B model (fp32) was ~10.8 GB, the compressed 700M model (fp32) is ~2.8 GB. Further quantizing to int8, the model size on disk becomes ~0.75 GB (a 14× reduction from original). Table 3 shows the accuracy of the model on GLUE and SQuAD before and after these steps.

Table 3 Performance of models on downstream tasks. Original model is fine-tuned on each task (or evaluated zero-shot for perplexity). Compressed model is our pruned+distilled version and fine-tuned per task. Quantized model is int8 quantized (post-training) version of compressed model, fine-tuned with quantization-aware training for GLUE/SQuAD. DistilBERT is fine-tuned on tasks for reference

Model	MNLI, Acc.	QNLI, Acc.	SST-2, Acc.	SQuAD, F1	WT103, Perplexity
Original 2.7B (cloud)	87.4	92.1	94.0	88.5	18.2
Compressed 700M (fp32)	85.5	90.0	92.7	85.0	20.5
Compressed+Quant 8-bit	85.2	89.6	92.5	84.6	21.0
DistilBERT (66M, reference)	82.2	87.5	91.0	80.8	–

As seen, the compressed model retains most of the performance: e.g., MNLI accuracy went from 87.4 to 85.5 (just a 1.9 drop), SQuAD F1 from 88.5 to 85.0 (3.5 drop). The quantization to 8-bit incurred a very small additional drop (within 0.3-0.5 on GLUE, and 0.4 on SQuAD F1). These differences are small relative to the gap between our model and a much smaller model like DistilBERT. Our 700M quantized model significantly outperforms DistilBERT on MNLI (+3% absolute) and SQuAD (+3.8 F1), showing that *our compression preserved the advantages of a larger model's representational power*. WT103 perplexity did worsen from 18.2 to ~21 after compression+quant, indicating some loss in generative fidelity (expected because pruning and distillation can prune out some of the less frequent knowledge). However, a perplexity of 21 is still quite good for a model of this size on WT103, and is far better than, say, a 117M GPT-2 (which has perplexity around 30-40 on WT103). This indicates our compressed model is still a strong language model.

Memory usage on the Jetson device: the original model could not even be loaded, the compressed quantized model uses ~6 GB of RAM when loaded (including PyTorch overheads and some activations for a batch of size 8). This fits within the 8 GB limit we set.

4.2. Distributed Inference Performance

We deployed the 700M int8 model on two Jetson devices (350M params each split). Each Jetson had ~4 GB of the model. We measured the time to process a single input of 128 tokens (typical sentence pair length for MNLI) through the model.

Single-device (one Jetson running entire 700M model): Latency = 520 ms per inference.

Two-device pipeline (each roughly half model): Latency = 330 ms per inference (including communication overhead for 128 tokens of size 768 each, which is ~0.4 MB transfer each way, negligible on 1 Gbps LAN).

Thus, using two devices gave a ~1.57× speedup, which is close to ideal scaling (1.8–2× would be ideal if there were no comm and perfect balance). The balance was indeed good (each Jetson had ~260 ms of compute, and ~70 ms was spent in transferring data). On a simulated slower network (100 Mbps), the latency increased to ~450 ms, indicating comm became the bottleneck (in that case ~80 ms to send 0.4 MB * 2 directions became ~320 ms, dominating). This shows that distributed inference is effective *if network is reasonably fast*. In a local wired or 5G context, it works well; on slower networks, one might instead not partition as much.

We also tried with 3 devices (splitting into ~233M params each) on the simulated setup and got further reduction to ~250 ms latency, but given diminishing returns and that most real deployments might have at most a couple of nearby devices, our main focus was 2.

In terms of throughput (queries per second), if multiple requests are pipelined, the 2-device system can sustain ~6-7 QPS (since both devices can be utilized in parallel on different queries) whereas one device did ~2 QPS, roughly scaling with number of devices.

4.3. Federated Learning Results

For the Reddit-based federated scenario, we measure perplexity on each user's comments (the model's ability to predict text similar to that user). Initially, before any fine-tuning, the global model (trained on generic data) had an average perplexity of 30 on the users' data, but varied: for some users who write in very domain-specific jargon, perplexity was as high as 50. We then simulate federated fine-tuning: each of 50 users (on 50 simulated devices) fine-tunes the model on their own comments (we actually used our two physical devices to iterate through user data, due to limited physical

devices we simulated sequentially, but each user's update is as if on their own device). We ran 10 rounds of FedAvg, with 10 users participating per round (randomly chosen), each performing 1 epoch on their local data (around 1000-2000 text samples each). We used a smaller learning rate for local fine-tuning (1e-5) to prevent divergence.

After 10 rounds, the global model's perplexity on an aggregate validation set of Reddit data (not seen in training) dropped from 28 to 24, indicating a gain in overall modeling of that style of data. More interestingly, the perplexity on each user's own data decreased significantly: the average per-user perplexity went from 30 to 20 (a 33% reduction), and even the worst-case user dropped from 50 to about 30. This demonstrates that *federated learning successfully personalized the model for the users while creating a better global model for that data distribution*. In contrast, if we fine-tuned the model centrally on the union of all data (which we did as an oracle baseline), perplexity reached ~22 – so FL got reasonably close to centralized training, showing only a small hit due to federated updates (which is expected due to fewer epochs and perhaps non-IID issues). We also observed that certain tokens and slang from specific subreddits became much better predicted by the model after FL – qualitatively, the model could generate responses with more *community-specific language* after federated training, which it couldn't do before (for privacy, we don't list those explicitly, but e.g. a user who often mentioned a particular video game character name saw that name's prediction probability increase in the model after FL).

For GLUE/SQuAD tasks, we did not need FL as those are centralized datasets; FL is more for the decentralzied scenario. But one could imagine federated fine-tuning on say distributed Q&A data. That is outside our current scope.

4.4. Overall System vs Baselines

Bringing it all together, we consider an *edge deployment scenario where we want to maximize accuracy under strict resource constraints.* Our system (700M quantized model on 2 edge devices with FL fine-tuning) achieved the following:

MNLI accuracy ~85%, QNLI ~90%, SST-2 ~92% (after fine-tuning, same as in Table 1 for compressed model). In contrast, a single DistilBERT on device (66M) gave 82/87/91. And a *TinyBERT* (a 25M parameter student of BERT) we tested gave even lower (79% MNLI).

Latency ~330 ms as mentioned, within acceptable range for many interactive applications (a third of a second).

Privacy: user data never left devices in the FL scenario.

The main competitor could be sending requests to a cloud LLM: e.g., an API call to a large model in the cloud might have a 100 ms network latency + 50 ms processing on a big GPU (just hypothetical). That could be faster and more accurate (since cloud model could be 10x bigger). However, in scenarios where privacy and offline capability are required, our edge solution shines. Also, note that with our optimization, the edge model's accuracy is not too far behind a 2.7B cloud model. Depending on the application, that might be an acceptable trade-off.

5. Discussion

5.1. Effectiveness of Combined Techniques

One clear takeaway is that no single technique was sufficient; it is the *combination* that enabled success. Compression alone got model size down, but inference was still slow on one device – distributed inference addressed that. Quantization alone could let the model fit, but without compression the model was still too large to be quantized on device. Federated learning alone can't even start unless the model can be deployed on device, which required the other optimizations. In our framework, each component addressed a different bottleneck: compression tackled memory, quantization tackled both memory and speed, distributed inference tackled compute speed, and federated learning tackled data scarcity and privacy. We found interesting interactions: for example, quantization sometimes exacerbated the effect of pruning on accuracy (because both introduce errors). However, doing a bit of *quantization-aware fine-tuning after pruning* recovered the loss. Also, distributed inference doesn't affect accuracy but introduces a hyper-parameter (the cut location). We initially cut evenly, but one could imagine learning the optimal cut via reinforcement learning or so.

5.2. Trade-offs

There are trade-offs to consider. First, Model size vs accuracy. We compressed 4× with ~2% accuracy loss. We could have compressed more aggressively (say 10×) by pruning more and using a smaller student, but we expect the accuracy

loss to grow – possibly non-linearly. There's likely a sweet spot depending on application requirements. Secondly, quantization bits. In our case, 8-bit was safe; 4-bit gave more compression but we did notice a ~1 point further drop on SQuAD when using 4-bit for some layers (we did not show in table, but e.g., SQuAD F1 went from 85 to 83 when using 4-bit for all linear layers). Techniques like AWQ mitigate that, but 4-bit is close to the edge of what the model can handle without fine-tuning. We decided 8-bit was a good balance given hardware support.

5.3. Scalability

How would this approach scale to even larger models or more devices? For instance, deploying a 10B or 70B model on edges. In principle, the techniques scale: one could compress a 70B to maybe 10B, quantize to 8-bit (2.5B params effective), and distribute across, say, 8 devices with 16GB each (fits ~2.5B in 8-bit). It's not outrageous, but the complexity of coordination and the latency might increase (more hops). Our scheduling method might need to be smarter (maybe partition both layers and sequence among devices to reduce activation sizes). Also, the federated learning of very large models might face issues: network strain and client memory – one might need to rely more on federated distillation or partial model updates. The PriSM approach could be integrated: each client only trains a subset of the model (the part it holds) and maybe occasionally the parts rotate or merge. This is a promising direction to allow FL of huge models.

5.4. Heterogeneity

In a real-world edge scenario, devices are often heterogeneous – some powerful, some weak. Our method can accommodate that by uneven partitioning (give more layers to stronger devices). Also some devices might not participate in FL due to connectivity or power issues; FL algorithms exist to handle dropouts and unbalanced updates (e.g., FedAvg can still work if some clients only occasionally contribute). In our experiments we simulated dropouts by randomly selecting clients each round; in practice one would implement mechanisms to handle stragglers.

5.5. Privacy and Security

Federated learning is not foolproof for privacy – model updates can potentially leak information. In a deployment, one might add differential privacy (DP) or secure aggregation to the FL process to further protect users [20]. That usually comes at the cost of some accuracy. Since our focus was not on DP, we didn't implement it, but it's worth noting as an important extension.

5.6. Comparison with Alternative Approaches

An alternative to our approach could be *on-device distillation at runtime* – e.g., have a small model running on device that is continuously distilled from a large model running in the cloud (sending queries to cloud occasionally to teach the small model). This could reduce the need for federated training. However, it reintroduces dependency on the cloud and latency, which we aimed to avoid. Another approach is *MoE (Mixture-of-Experts) models*, which keep a large capacity but activate only a small part for each input (thus could be sparse and maybe distributed). MoEs are an interesting direction: if each device holds a subset of experts, one could route tokens to the appropriate device.

5.7. Limitations and Reproducibility

Our experiments, while extensive, were limited to certain model sizes and tasks. The generalization of conclusions to all LLMs should be cautious. Very large LLMs (like GPT-3 175B) have emergent abilities that might degrade more noticeably when compressed heavily. We didn't test tasks like coding or logical reasoning – it's possible a compressed model loses more on those complex tasks relative to simpler GLUE tasks. Another limitation is that our evaluation of federated learning was on a proxy dataset (Reddit) for personalization. In a real application (say a personal assistant on phones), the distribution might be different, and evaluation is tricky because it involves user experience.

We took care to detail our procedures. One potential hiccup is that distillation sometimes requires careful hyperparameter tuning (temperature, learning rate, etc.) to succeed – we spent some effort tuning those for our model. If one compresses a model without such tuning, results could vary.

5.8. Theoretical considerations

In theory, one might ask how close the compressed model can approach the original. There are some theoretical works on model compression bounds, but for LLMs it's largely empirical. We note that if the model has a lot of redundancy, as it seems, even a 10× smaller model can do well (DistilBERT case). But eventually compressing too much will hit a sharp drop-off, which from literature often happens when student model has <10% of teacher's parameters. We stayed at

~25% with success. Future work could explore pushing this boundary with multi-stage compression or clever teacher assistant training (where you compress in steps, teacher->mid->smaller).

5.9. Algorithmic complexity

The complexity of our distributed inference algorithm is linear in number of layers (since each layer forwards once). Communication complexity is linear in number of devices (each activation vector sent once). So overall O(L + M) per inference. The federated training complexity is more significant: if we have N devices each with data size n, a round costs each device O(n * model_size) and communication O(model_size). But since our model is now smaller and quantized, and we might not use all devices every round, it was manageable. In our case, training 700M model with 50 devices for 10 rounds took a few hours on 2 physical devices sequentially – scaled out it would be faster.

5.10. Consistent framework usage

Using PyTorch throughout was helpful, especially with Flower for FL, and ONNX for deploying quantized model if needed. We ensure that any custom kernels (like our GPTQ quantization) were carefully validated to produce same results across devices.

In conclusion, the discussion reaffirms that optimizing LLMs for edge involves a multifaceted approach. The success of our experiments suggests that relatively large models can indeed move to the edge with clever optimization, opening doors for more private, low-latency AI applications.

6. Conclusion

In conclusion, Optimizing LLMs for the edge is a critical step towards ubiquitous and user-centric AI. Our research demonstrates that through a combination of cutting-edge techniques, one can significantly bend the cost-accuracy curve, bringing large-model intelligence to devices at the edge of the network. We hope this work serves as a blueprint for future systems and inspires further innovations in efficient AI deployment.

References

[1] Y. Zheng, Y. Chen, B. Qian, X. Shi, Y. Shu, and J. Chen, 'A Review on Edge Large Language Models: Design, Execution, and Applications', Feb. 23, 2025, arXiv: arXiv:2410.11845. doi: 10.48550/arXiv.2410.11845.

[2] M. Xia, Z. Zhong, and D. Chen, 'Structured Pruning Learns Compact and Accurate Models', in Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), S. Muresan, P. Nakov, and A. Villavicencio, Eds., Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 1513–1528. doi: 10.18653/v1/2022.acl-long.107.

[3] X. Ma, G. Fang, and X. Wang, 'LLM-Pruner: On the Structural Pruning of Large Language Models', presented at the Thirty-seventh Conference on Neural Information Processing Systems, Nov. 2023. Accessed: Mar. 19, 2025. [Online]. Available: https://openreview.net/forum?id=J8Ajf9WfXP

[4] V. Sanh, T. Wolf, and A. Rush, 'Movement Pruning: Adaptive Sparsity by Fine-Tuning', in Advances in Neural Information Processing Systems, Curran Associates, Inc., 2020, pp. 20378–20389. Accessed: Mar. 19, 2025. [Online]. Available: https://proceedings.neurips.cc/paper/2020/hash/eae15aabaa768ae4a5993a8a4f4fa6e4-Abstract.html

[5] E. Frantar and D. Alistarh, 'SparseGPT: Massive Language Models Can be Accurately Pruned in One-Shot', in Proceedings of the 40th International Conference on Machine Learning, PMLR, Jul. 2023, pp. 10323–10337. Accessed: Mar. 19, 2025. [Online]. Available: https://proceedings.mlr.press/v202/frantar23a.html

[6] S. Shen et al., 'Q-BERT: Hessian Based Ultra Low Precision Quantization of BERT', Sep. 25, 2019, arXiv: arXiv:1909.05840. doi: 10.48550/arXiv.1909.05840.

[7] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, 'OPTQ: Accurate Quantization for Generative Pre-trained Transformers', presented at the The Eleventh International Conference on Learning Representations, Sep. 2022. Accessed: Mar. 19, 2025. [Online]. Available: https://openreview.net/forum?id=tcbBPnfwxS

[8] T. Dettmers, M. Lewis, Y. Belkada, and L. Zettlemoyer, 'LLM.int8(): 8-bit matrix multiplication for transformers at scale', in Proceedings of the 36th International Conference on Neural Information Processing Systems, in NIPS '22. Red Hook, NY, USA: Curran Associates Inc., Nov. 2022, pp. 30318–30332.

[9] J. Lin et al., 'AWQ: Activation-aware Weight Quantization for On-Device LLM Compression and Acceleration', Proceedings of Machine Learning and Systems, vol. 6, pp. 87–100, May 2024.

[10] Y. Luo and L. Chen, 'DAQ: Density-Aware Post-Training Weight-Only Quantization For LLMs', Oct. 17, 2024, arXiv: arXiv:2410.12187. doi: 10.48550/arXiv.2410.12187.

[11] Z. Yao, R. Yazdani Aminabadi, M. Zhang, X. Wu, C. Li, and Y. He, 'ZeroQuant: Efficient and Affordable Post-Training Quantization for Large-Scale Transformers', Advances in Neural Information Processing Systems, vol. 35, pp. 27168–27183, Dec. 2022.

[12] G. Xiao, J. Lin, M. Seznec, H. Wu, J. Demouth, and S. Han, 'SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models', in Proceedings of the 40th International Conference on Machine Learning, PMLR, Jul. 2023, pp. 38087–38099. Accessed: Mar. 19, 2025. [Online]. Available: https://proceedings.mlr.press/v202/xiao23c.html

[13] Y. Kang et al., 'Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge', in Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, in ASPLOS '17. New York, NY, USA: Association for Computing Machinery, Apr. 2017, pp. 615–629. doi: 10.1145/3037697.3037698.

[14] X. Zhang, J. Liu, Z. Xiong, Y. Huang, G. Xie, and R. Zhang, 'Edge Intelligence Optimization for Large Language Model Inference with Batching and Quantization', in 2024 IEEE Wireless Communications and Networking Conference (WCNC), Apr. 2024, pp. 1–6. doi: 10.1109/WCNC57260.2024.10571127.

[15] A. Borzunov et al., 'Distributed Inference and Fine-tuning of Large Language Models Over The Internet', 2023.

[16] Y. Niu, S. Prakash, S. Kundu, S. Lee, and S. Avestimehr, 'Federated Learning of Large Models at the Edge via Principal Sub-Model Training', presented at the Workshop on Federated Learning: Recent Advances and New Challenges (in Conjunction with NeurIPS 2022), Oct. 2022. Accessed: Mar. 19, 2025. [Online]. Available: https://openreview.net/forum?id=e97uuEXkSii&noteId=e0fzUuoTuN7

[17] H. Woisetschläger, A. Erben, S. Wang, R. Mayer, and H.-A. Jacobsen, 'Federated Fine-Tuning of LLMs on the Very Edge: The Good, the Bad, the Ugly', in Proceedings of the Eighth Workshop on Data Management for End-to-End Machine Learning, Santiago AA Chile: ACM, Jun. 2024, pp. 39–50. doi: 10.1145/3650203.3663331.

[18] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, 'GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding', presented at the International Conference on Learning Representations, Sep. 2018. Accessed: Mar. 19, 2025. [Online]. Available: https://openreview.net/forum?id=rJ4km2R5t7

[19] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, 'GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers', Mar. 22, 2023, arXiv: arXiv:2210.17323. doi: 10.48550/arXiv.2210.17323.

[20] M. Chen, N. Shlezinger, H. V. Poor, Y. C. Eldar, and S. Cui, 'Communication-efficient federated learning', Proceedings of the National Academy of Sciences, vol. 118, no. 17, p. e2024789118, Apr. 2021, doi: 10.1073/pnas.2024789118.