Compression and Distillation: Pruning, Quantization, and TinyLLM Edge Deployment

October 25, 2025

1 Pruning and Quantization

1.1 Compression roadmap

Figure **??** outlines a typical path from dense LLMs to compact deployable models: pruning removes redundant structure, quantization reduces numeric precision, and distillation produces student models tailored for constrained devices.

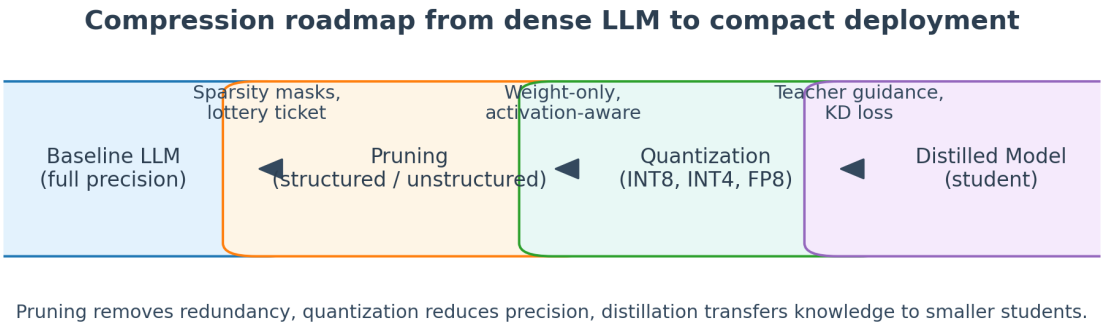

Figure 1: Compression roadmap combining pruning, quantization, and distillation.

1.2 Pruning strategies

- **Unstructured pruning:** Magnitude- or gradient-based removal of individual weights; flexible but hardware-unfriendly.
- **Structured pruning:** Drops entire channels, heads, or matrix columns, preserving dense matrix kernels.
- **Dynamic sparsity:** Methods like RigL or SNIP adjust sparsity masks during training for better convergence.
- **Lottery ticket subnetworks:** Identify sub-networks that can be retrained from scratch, guiding lightweight fine-tuning.

1.3 Quantization taxonomy

Method	Bit width	Highlights	Tooling
Dynamic quantization	INT8	On-the-fly activation stats during inference	PyTorch Dynamic Quantization
Static quantization	INT8	Calibration dataset for scale/zero-point	TensorRT, ONNX Runtime
Weight-only quantization	INT4/INT3	Compress weights while retaining FP16 activations	GPTQ, AWQ
Activation-aware quantization	INT8/INT4	Jointly scales weights and activations	SmoothQuant, AQLM
Mixed precision	FP8/INT8	Leverages modern accelerators (H100, Gaudi2)	TransformerEngine, DeepSpeed

1.4 GPTQ example

Listing 1: Applying GPTQ to quantize LLaMA weights

```python
from auto_gptq import AutoGPTQForCausalLM, BaseQuantizeConfig
from transformers import AutoTokenizer

model_name = "meta-llama/Llama-2-7b-hf"
quant_config = BaseQuantizeConfig(bits=4, group_size=128, desc_act=False)

model = AutoGPTQForCausalLM.from_pretrained(model_name, quantize_config=quant_config)
tokenizer = AutoTokenizer.from_pretrained(model_name, use_fast=True)

model.quantize(
    examples=["Compression saves memory.", "GPTQ performs post-training quantization."],
    batch_size=8,
    use_triton=True,
)

model.save_quantized("llama2-7b-gptq")
tokenizer.save_pretrained("llama2-7b-gptq")
```

2 Knowledge Distillation

2.1 Distillation workflow

Knowledge distillation (KD) transfers inductive biases and behaviors from a teacher to a student:

- **Soft targets:** Temperature-scaled logits guide the student via KL divergence.
- **Intermediate matching:** Align attention maps, hidden states, or gradients between teacher and student.
- **Task-specific KD:** Use teacher outputs on annotated or synthetic tasks to supervise student fine-tuning.

2.2 Loss formulation

$$\mathcal{L} = \alpha \mathcal{L}_{\text{KD}}(p_s, p_t) + \beta \mathcal{L}_{\text{task}}(y_s, y) + \gamma \mathcal{L}_{\text{feature}}(h_s, h_t), \tag{1}$$

where α, β, and γ regulate the balance between soft targets, ground-truth labels, and feature alignment.

2.3 Case highlights

- **TinyLlama:** 1.1B/3B students distilled from larger instruction-tuned teachers for mobile inference.
- **MiniLM:** Deep self-attention distillation yields compact BERT variants with competitive accuracy.
- **LLaDA:** Cross-lingual and multimodal knowledge distilled into smaller multilingual models.

2.4 Teacher-student loop

Listing 2: Simplified knowledge distillation training loop

```python
for batch in dataloader:
    with torch.no_grad():
        teach_out = teacher(**batch, output_hidden_states=True)

    stud_out = student(**batch, output_hidden_states=True)

    loss_kd = kl_divergence(
        F.log_softmax(stud_out.logits / T, dim=-1),
        F.softmax(teach_out.logits / T, dim=-1),
    ) * (T * T)

    loss_task = cross_entropy(stud_out.logits, batch["labels"])
    loss_hidden = mse_loss(
        stud_out.hidden_states[-1],
        projector(teach_out.hidden_states[-1]),
    )

    loss = alpha * loss_kd + beta * loss_task + gamma * loss_hidden
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

3 TinyLLM and Edge Deployment

3.1 Deployment stack

Figure **??** illustrates the TinyLLM deployment stack: a compression pipeline generates compact models, a quantized runtime executes on heterogeneous hardware, and orchestration plus monitoring close the loop.

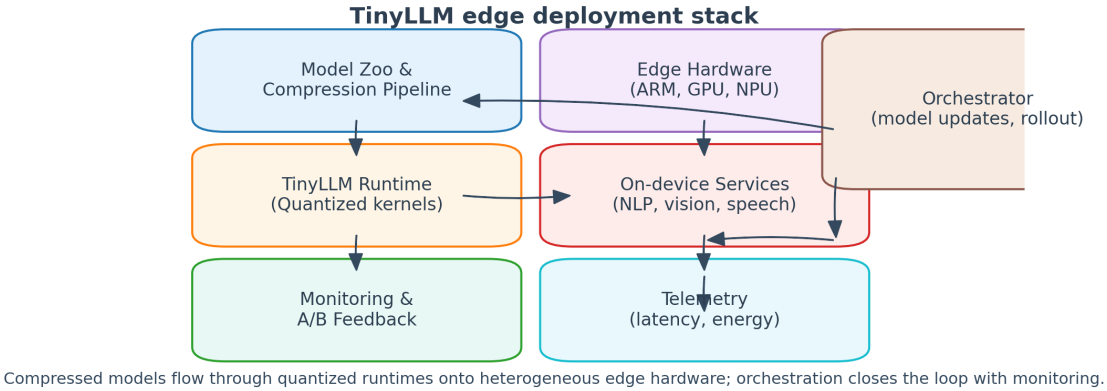

Figure 2: TinyLLM edge deployment stack with compression pipeline, runtime, hardware, and governance.

3.2 TinyLLM runtime features

- **Kernel optimization:** INT4/INT8 GEMMs, tensor slicing, pipelined attention, and paged KV caches.
- **Memory management:** Static pools, grouped-query attention, and streaming weights reduce footprint.
- **Scheduler:** Multi-tenant queueing, dynamic batching, latency-aware throttling for SLA compliance.

3.3 Edge hardware matrix

Hardware	Precision	Target workloads	Toolchain
ARM CPUs (Neon)	INT8/INT4	Offline assistants, smart speakers	MNN, NCNN, llama.cpp
NVIDIA Jetson	FP16/INT8	Robotics, industrial inspection	TensorRT, FasterTransformer
Apple Neural Engine	8-bit	On-device iOS assistants	Core ML, Metal Performance Shaders
Custom NPUs	INT4/INT2	Automotive, smart home, wearables	ONNX Runtime EPs, Apache TVM

3.4 Deployment workflow

1. Compress base models via pruning + quantization + distillation into student checkpoints.
2. Convert to target runtimes (ONNX, TensorRT, Core ML, GGUF).
3. Integrate with TinyLLM or other inference runtimes (FasterTransformer, MNN, llama.cpp).
4. Collect telemetry (latency, power, accuracy) and feed back into retraining or adaptive rollout.

3.5 Inference script

Listing 3: Running a quantized TinyLLM model with llama.cpp

```
import subprocess
from pathlib import Path

model_path = Path("models/tinyllm-q4_0.gguf")
prompt = "Provide a bilingual summary of today's incident reports."

cmd = [
    "./main",
    "-m", str(model_path),
    "-p", prompt,
    "-n", "160",
    "--temp", "0.7",
    "--batch-size", "48",
    "--threads", "6",
]

completed = subprocess.run(cmd, capture_output=True, text=True, check=True)
print(completed.stdout)
```

Operational recommendations

- Evaluate compression knobs jointly—quantization-aware pruning, sparse-aware distillation—to reach desired accuracy/fps goals.
- Establish regression suites covering perplexity, task accuracy, latency, and energy, plus red-team tests for safety.
- Instrument edge devices to gather anonymized telemetry; use data for continuous distillation and dynamic model selection.
- Harden edge deployments with encryption, secure boot, and policy enforcement despite local inference benefits.

Further reading

- Frantar et al. "GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers." NeurIPS, 2022.
- Dettmers et al. "QLoRA: Efficient Finetuning of Quantized LLMs." NeurIPS, 2023.
- Sanh et al. "DistilBERT, a distilled version of BERT." NeurIPS, 2019.
- Zhang et al. "TinyLlama: Tiny Language Models for Edge AI." arXiv, 2024.
- Li et al. "AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration." arXiv, 2023.