Accelerating LLM Training: Best Practices & Cookbook

Compiled by Ethan Henley, see references for primary sources.

Optimizing Data Precision

Mixed Precision Training

- Combine float32 and float16/bfloat16 datatypes during training.
- Store model weights and optimizer states in float32 for stability.
- Perform passes in float16/bfloat16 to reduce memory and increase computation speed.
- Implement loss scaling to prevent underflow in gradients

Quantization for Inference

- Convert float32 models to INT8 for faster inference with minimal accuracy loss.
- Simulate quantization effects during training for better accuracy in quantized models.
- Consider hybrid approaches, keeping sensitive layers (e.g., embeddings) in higher precision.

Efficient Model Architectures

Flash Attention

- Replaces traditional attention mechanisms with a more memory-efficient algorithm.
- Reduces memory complexity from O(n^2) to O(n)
- Particularly effective for long sequences, enabling training on longer contexts.

Rotary Position Embeddings (RoPE)

- Encodes positional information directly into query and key vectors.
- Provides better relative positional encoding.
- Enables better generalization to sequences longer than those seen during training.

SwiGLU Activation

- A variant of the GLU (Gated Linear Unit) activation function.
- Combines the benefits of SiLU (Swish) activation and gating mechanisms.
- Often leads to faster convergence and better performance compared to ReLU or GELU.

Memory Optimization Techniques

Gradient Checkpointing

- Trade computation for memory by recomputing activations during the backward pass.
- Significantly reduces memory usage at the cost of increased computation time.

Activation Offloading

- Temporarily move activations to CPU memory during the forward pass
- Bring activations back to GPU only when needed for the backward pass.
- Enables training of larger models on limited GPU memory.

Fully Sharded Data Parallel (FSDP)

- Distribute model parameters, gradients, and optimizer states across multiple GPUs.
- Reduces memory requirements per GPU, enabling training of larger models.

Optimizing Training Loops

Gradient Accumulation

- Perform several forward and backward passes before updating model parameters.
- Simulates larger batch sizes without increasing memory requirements.

Dynamic Batch Sizing

- Adjust batch size on-the-fly based on available memory.
- Start with a large batch size and reduce if out-of-memory errors occur.

Curriculum Learning

- Start training on shorter or simpler sequences and gradually increase complexity.
- Can lead to faster convergence and better final performance.

Algorithmic Improvements

Group Query Attention (GQA)

- Reduce computational complexity of attention mechanism by grouping queries.
- Maintains model quality while significantly reducing memory and computation requirements.
- The more attention heads, the more effective.

Deep and Thin Architectures

- Increase model depth while keeping width (hidden size) relatively small.
- Can achieve similar or better performance than wider, shallower models with fewer total parameters.
- Requires careful tuning of learning rates and initialization to train effectively.

Embedding Sharing

- Use the same embedding matrix for input and output layers in encoder-decoder models.
- Reduces model size and can act as a regularizer.
- Particularly effective in language models where input and output vocabularies are the same.

Training Recipes for Common Scenarios

Training Large Models on Limited Hardware

1. Implement QLoRA for parameter efficiency
2. Use activation checkpointing to reduce memory footprint
3. Apply gradient accumulation to simulate larger batch sizes
4. Utilize CPU offloading for optimizer states

Scaling to Multi-GPU Training

1. Implement Fully Sharded Data Parallel (FSDP) training
2. Use mixed precision training (e.g., bfloat16) to reduce memory usage and increase speed
3. Apply Per-Parameter FSDP for flexible sharding with quantization techniques
4. Implement efficient data loading with prefetching and multiple worker

Optimizing For Inference Speed

1. Apply post-training quantization to INT8
2. Use knowledge distillation to create smaller, faster models
3. Implement efficient attention mechanisms like Flash Attention
4. Optimize model architecture (e.g., replace LayerNorm with RMSNorm)

Training on Very Long Sequences

1. Implement efficient attention mechanisms (e.g., Flash Attention, Sparse Attention)
2. Use gradient checkpointing to reduce memory usage
3. Apply curriculum learning, starting with shorter sequences
4. Implement sliding window attention or chunked cross-entropy for very long contexts

Fine-tuning Large Pre-trained Models

1. Use LoRA or QLoRA to reduce memory footprint and trainable parameters
2. Implement efficient optimizers like AdamW8bit
3. Apply mixed precision training
4. Use gradient accumulation for effective larger batch sizes

Training Models for Edge Devices

1. Use knowledge distillation to create a smaller, efficient model from a larger one
2. Simulate quantization effects to prepare the model for int8 or lower precision deployment
3. Implement efficient model architectures like MobileNet or EfficientNet as a base
4. Use pruning techniques to reduce model size while maintaining accuracy
5. Deploy the model for specific edge hardware (e.g., TorchChat, ExecuTorch)

References

"Efficient Transformers: A Survey" (Tay et al., 2020) | "Flash Attention: Fast and Memory-Efficient Exact Attention with IO-Awareness" (Dao et al., 2022) | "Training Compute-Optimal Large Language Models" (Hoffmann et al., 2022) | PyTorch Performance Tuning Guide | NVIDIA Deep Learning Performance Guide | Hugging Face Transformers Optimization Documentation