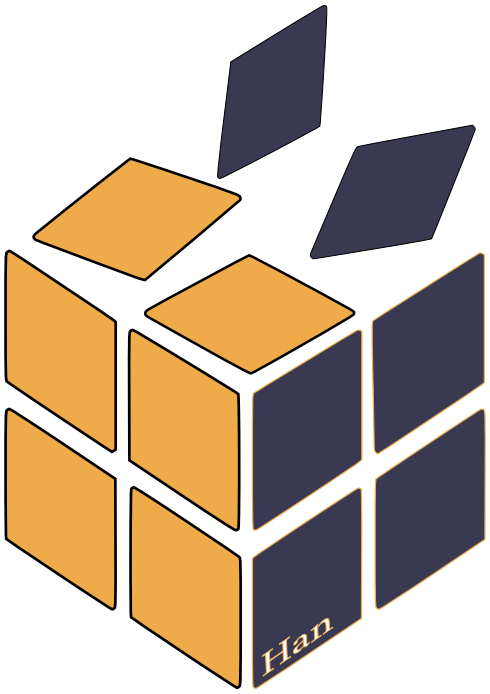
Deep Learning System Design



Han Cheol Moon
tabularasa8931@gmail.com

The logo depicts a cube puzzle gradually coming together, reflecting the journey of learning. Each piece represents a fragment of knowledge, and as they fall into place, they reveal the larger structure of understanding. It conveys the idea that growth is a process — knowledge is completed bit by bit.

Contents

Part I

Introduction

Chapter 1

Introduction

1.1 Operations challenges with LLMs

- **Long download times** (*e.g.,* Bloom LLM is 330GB).
- **Longer deploy times** (*e.g.,* Bloom takes $30 \sim 45$ mins to load the model into GPU).
- Along with increases in model size often come increases in **inference latency**.
- **Managing GPUs**
- **Peculiarities of text data**: unlike other fields, texts have ambiguities.
- **Token limits for a model** create bottlenecks
- **Hallucinations cause confusion**
- **Bias and ethical considerations**
- **Security concerns**
- **Controlling costs**: *e.g.,* GPUs, infra, storage, operational costs like energy consumption during both training and inference.

1.2 LLMOps Essentials

- **Compression** is the practice of making models smaller.
  - **Quantizing** is the process of reducing precision in preference of lowering the memory requirements.
  - **Pruning** is the process of weeding out and removing any parts of the model we deem unworthy.
  - **Knowledge distillation** takes the large LLM and train a smaller language model to copy it.
  - **Low-rank approximation** is a trick to simplify large matrices or tensors to find a lower dimensional representation.

  - **Mixture of Experts** (MoE) is a technique where we replace the feed-forward (FF) layers in a transformer with MoE layers instead. FF layers are notorious for being parameter-dense and computationally intensive, so replacing them with something better can often have a large effect. MoEs are a group of sparsely activated models. They differ from ensemble techniques in that typically only one or a few expert models will be run, rather than combining results from all models. The sparsity is often induced by a *gate mechanism* that learns which experts to use and/or a router mechanism that determines which experts should even be consulted.

- **Distributed computing** is a technique used in DL to parallelize and speed up large, complex neural networks by dividing the workload across multiple devices or nodes in a cluster. This approach significantly reduces training and inference times by enabling concurrent computation, data parallelism, and model parallelism.

  - **Data parallelism**: splitting up the data and running them through multiple copies of the model or pipeline.
  - **Tensor parallelism**: This approach takes advantage of matrix multiplication properties of split up the activations across multiple processors, running the data through and then combining them on the other side of the processors.
  - **Pipeline parallelism**: This creates a pipeline, as input data will go to the first GPU, process, then transfer to the next GPU, and so on until it's run through the entire model.
  - **3D parallelism**: We want to take advantages of all three parallelism practices as they can all be run together. This is known as 3D parallelism, which combines data, tensor and pipeline parallelism (DP+TP+PP) together. Since each technique and thus dimension will require at least two GPUs to run 3D parallelism, we will need at least eight GPUs to get started.

1.3 LLM Operations Infrastructures

- Data infrastructure is the foundation of DataOps. (*e.g.,* Airflow, Prefect, and Mage)
- Experiment trackers (*e.g.,* MLFlow and Weights & Biases)
- Model registry
- Feature stores (*e.g.,* Feast) is a centralized system for managing, storing, and serving features (the inputs to machine learning models) in a consistent and reliable way. It sits between your raw data sources and your ML models, making it easier to build, deploy, and maintain production ML systems. Why do we need it? In ML, the same features are used in two places:

  - Training – when building the model.
  - Serving/Inference – when the model is deployed to make predictions.

- Vector databases (*e.g.,* Qdrant, Pinecone, and Milvus) are specialized databases that store vectors along with some metadata around the vector, which makes them great for storing embeddings. The power of vector databases isn't in their storage but in the way that they search through the data.

- Monitoring systems (*e.g.,* whylogs and): ML models are often fail silently (*e.g.,* data drift [1]).
- GPU-enabled workstations (*e.g.,* H100 provides 80GB and NVL [2])
  - If you aren't sure which GPU you need to run which model, multiply the number of billions of parameters by two, since most models at inference will default to run at half precision, FP16 or BF16, which means we need at least 2 bytes for every parameter. You will need a little extra as well for embedding model, which will be about another gigabyte, and more for the actual toknes. One token is about 1MB. For 16 bathes of this size, you will need an extra 8GB of space.
  - For training you will need a lot more space (*e.g.,* full precision, optimizer tensors and gradients). Roughly you need 16 bytes for every parameter, so to train a 7B parameter model, you will need 112GB of memory.
- Deployment service (*e.g.,* NVIDIA Triton Inference Service, MLServer, Seldon, BentoML)

[1] Data drift in Machine Learning (ML) systems refers to changes in the statistical properties of input data (or the relationship between inputs and outputs) over time, which can degrade model performance if not monitored and addressed.

[2] NVIDIA Link is NVIDIA's high-speed interconnect technology. It allows GPUs (and sometimes CPUs) to communicate with each other much faster than PCIe. In multi-GPU systems (like deep learning servers, HPC, or AI supercomputers), GPUs need to share data frequently (*e.g.,* gradients and parameters). If they only use PCIe, communication is slower, creating a bottleneck.

Chapter 2

Preliminaries

2.1 Complexity of Matrix Multiplication

Matrix multiplication is a fundamental operation in many computational tasks, including neural networks. The complexity of multiplying two matrices depends on their dimensions. Let's dive into the specifics.

- Let A be a matrix of size $m \times k$.
- Let B be a matrix of size $k \times n$.
- The result C will be a matrix of size $m \times n$.

Standard Matrix Multiplication: For each element c_{ij} in the resulting matrix C:

$$c_{ij} = \sum_{l=1}^{k} a_{il} \cdot b_{lj}$$

This involves:

- Multiplications: k multiplications for each element c_{ij}.
- Additions: $k - 1$ additions for each element c_{ij}.

Complexity

- The total number of elements in C is $m \times n$.
- Therefore, the total number of multiplications is $m \times n \times k$.
- The total number of additions is $m \times n \times (k - 1)$.

Thus, the total complexity is $O(m \times n \times k)$.

Even though there are several advanced methods, the standard $O(m \times n \times k)$ complexity is often used in practice, due to the simplicity and efficiency of implementation on modern hardware. Optimized libraries (like BLAS, cuBLAS for GPUs) leverage hardware-specific optimizations to improve practical performance.

Part II

Data Engineering

Chapter 3

Data Engineering for LLMs

Data engineering is the development, implementation, and maintenance of systems and processes that take in raw data and produce high-quality, consistent information that supports downstream use cases, such as analysis and machine learning.

There isn't more valuable asset than your data. All successful AI and ML initiatives are built on a good data engineering foundation. It's important then that we acquire, clean, and curate our data.

3.1 Models and the Foundation

The most important dataset you will need to collect when training is the model weights of a pretrained model.

3.1.1 Evaluating LLMs

Evaluating a model requires two things: (*i*) a *metric* and (*ii*) a *dataset*.

Metrics

- ROUGE (Recall-Oriented Understudy for Gisting Evaluation)
- BLEU (BiLingual Evaluation Understudy)
- BPC (*e.g.,* Perplexity): The bits per character (BPC) evaluation is an example of an entropy-based evaluation for language models.

Industry benchmarks

- GLUE (General Language Understanding Evaluation) is essentially a standardized test for language models to measure performance versus humans and each other on language tasks meant to test understanding.

- SuperGLUE
- MMLU (Massive Multitask Language Understanding).

Responsible AI benchmarks

- HONEST evaluation metric compares how hurtful prompt completions are for different genders.
- Some datasets:
  - WinoBias dataset focuses on gender bias.
  - CALM
  - WinoQueer

Developing your own benchmark

- OpenAI's Evals library
- Huggingface's Evaluate

Evaluating code generators The basic setup looks like this:

1. Have your model generate code based on docstrings.
2. Run the generated code in a safe environment on prebuilt tests to ensure they work and that no errors are thrown
3. Run the generated code through a profiler and record the time it takes to complete.
4. Run the generated code through a security scanner and count the number of vulnerabilities.
5. Run the generated code against architectural fitness functions to determine artifacts like how much coupling, integrations, and internal dependencies there are.
6. Run steps 1 to 5 on another LLM.
7. Compare results.

Evaluating model parameters There's a lot you can learn by simply looking at the parameters of an ML model. For instance, an untrained model will have a completely random distribution.

```
import weightwatcher as ww
from transformers import GPT2Model

gpt2_model = GPT2Model.from_pretrained("gpt2")
gpt2_model.eval()

watcher = ww.WeightWatcher(model=gpt2_model)
details = watcher.analyze(plot=False)
print(details.head())
#    layer_id       name          D  ...       warning        xmax        xmin
```

```
#  0          2  Embedding  0.076190  ...  over-trained  3837.188332    0.003564
#  1          8     Conv1D  0.060738  ...                2002.124419  108.881419
#  2          9     Conv1D  0.037382  ...                 712.127195   46.092445
#  3         14     Conv1D  0.042383  ...                1772.850274   95.358278
#  4         15     Conv1D  0.062197  ...                 626.655218   23.727908
```

Each layer in your neural network has a weight matrix. When you analyze this matrix, you get a list of numbers called eigenvalues. These eigenvalues reveal how spread out or concentrated the layer's learning is.

The spectral analysis plots evaluate the frequencies of eigenvalues for each layer of a model. These plots tell you whether a model (or layer) looks well-trained and generalizes well or is unstable/poorly conditioned. Shape of the Spectrum (How eigenvalues are distributed)

- Power-law exponent (α): The Training Quality Score, the power-law exponent of the spectral tail.
  - Good if between 2 and 6: the layer is well-trained.
  - Bad if $\alpha > 6$: layer might be undertrained or over-regularized.
- Fit quality (Dks): The Reliability Score
  - Low Dks: spectrum matches the expected "heavy-tailed" shape, reliable (most eigenvalues are small, but some are very large, and they don't disappear quickly).
  - High Dks: poor fit, unstable or unstructured layer.

Note that

- In a light-tailed distribution, big values are rare and drop off fast.
- In a heavy-tailed distribution, big values are not that rare — you keep seeing occasional huge values.

3.2 Data for LLMs

It has been shown that data is the most important part of training an LLM.

Table 3.1: Summary of datasets

Dataset	Contents	Size	LastUpdate
WikiText	English Wikipedia	<1GB	2016
Wiki-40B	Multi-lingual Wikipedia	10GB	2020
Europarl	European Parliament proceedings	1.5GB	2011
Common Crawl	The internet	~ 300GB	Ongoing
OpenWebText	Curated internet using Reddit	55GB	2019
The Pile	Everything above plus specialty datasets (books, law, med)	825GB	2020
RedPajama	GitHub, arXiv, Books, Wikipedia, StackExchange , and multiple version of Common Crawl	5TB	2023
OSCAR	Highly curated multilingual dataset with 166 languages	9.4TB	Ongoing

3.2.1 Data cleaning and preparation

If you pulled any of the previously mentioned datasets, you might be surprised to realize most of them are just giant text dumps. There are no labels or annotations, and feature engineering hasn't been done at all.

LLMs are trained via self-supervised manner to predict the next word or a masked workd, so a lot of traditional data cleaning and preparation processes are unneeded. This fact leads many to believe that data cleaning as a whole is unnecessary.

Data cleaning and curation are difficult, time-consuming, and ultimately subjective tasks that are difficult to tie to key performance indicators (KPIs). Still, taking the time and resources to clean your data will create a more consistent and unparalleled user experience.

The right frame of mind when preparing your dataset:

1. Take your pie of data and determine a schema for the features
2. Make sure all the features conform to a distribution that makes sense for the outcome your're trying to get through normalization or scaling.
3. Check the data for bia/anomalies (most businesses skip this step by using automated checking instead of informed verification).
4. Convert the data into a format for the model to ingest (for LLMs, it's through tokenization and embedding).
5. Train, check, and retrain.

Note

For more information, check out Fundamentals of Data Engineering, WizardLM, and LIMA: Less Is More for Alignment.

Instruct Schema is one of the most effective and widely used data formats for fine-tuning models. Instruction tuning works on the principle that providing a model with explicit instructions for a task leads to better performance than simply giving it raw prompts and answers. In this approach, the data explicitly demonstrates what the model should do, making it clearer and more aligned with human intent. However, preparing such datasets is more demanding than assembling general web data, since each entry must be carefully constructed to match a structured format, typically including an instruction, optional input, and the expected output. You need to prepare your data to match a format that will look something like this:

```
###Instruction

{user input}

###Input

{meta info about the instruction}

###Response

{model output}
```

It is a structured way of formatting data so that each example clearly contains:

- An instruction (what the model should do).
- An input (optional context or data the model works on).
- An output (the desired response).

For instance,

```
{
  "instruction": "Translate the following English text into Korean.",
  "input": "The stock market saw significant volatility today due to global
    economic concerns.",
  "output": <Translations>
}
```

Note

- EvolInstruct: WizardLM
- Self-instruct format, Alpaca

Ensuring proficiency with speech acts When preparing a dataset for training a model, the most important factor is ensuring the data truly reflects the task you want the model to perform. Misaligned or overly generic data reduces performance and can cause unpredictable behavior.

Dataset alignment:

- Training data must match the intended task (e.g., don't train on Titanic survivors if you want to predict Boston housing prices).
- In real-world use cases (like fast-food ordering), interactions are more diverse and unpredictable than generic datasets suggest.

Robustness and Risks:

- Instruction datasets require intentional design: if a model is only trained on "helpful" responses, it might follow harmful instructions (e.g., "help me take over the world").
- With tool access (Google, HR docs), this becomes even riskier.

Understanding speech acts (directives, representatives, commissives, expressives, declarations, verdictives) helps design datasets that match realistic user interactions.

- In language learning, this means learners should not only know grammar/vocabulary but also how to perform speech acts appropriately:
  - How to make polite requests
  - How to refuse without sounding rude
  - How to apologize or thank in culturally acceptable ways

- In AI / LLM context, it means training the model to:
  - Generate outputs that correctly perform the intended communicative function (e.g., distinguish between an instruction, a suggestion, or a formal declaration).
  - Handle pragmatic nuances, politeness, indirectness, etc.

Speech acts refer to the various functions language can perform in communication beyond conveying information. They are a way of categorizing utterances based on their intended effect or purpose in a conversation. In short, it is an action performed through speaking. For example:

- Assertives → stating something true/false.
- Directives → requesting, commanding (*e.g.,* "Get it done in the next three days").
- Commissives → promising, committing (*e.g.,* "I swear").
- Expressives → greetings, apologizing (*e.g.,* "You are the best").
- Declarations → enacting something by saying it (*e.g.,* "I now pronounce you married").
- Questions (*e.g.,* "What is this?")

Annotating the data Annotation is labeling your data, usually in a positionally aware way. For speech recognition tasks, annotations would identify the different words as noun, verb, adjective, or adverb. Annotations essentially give us metadata that makes it easier to reason about and analyze our datasets.

There are tools to help with the task:

- Prodigy: multimodal annotation tool.
- doccano: Open-source web-based platform for data annotation.
- Praat: The audio annotation tool.
- Galileo: Galileo's LLM studio helps create prompt, evaluate and speed up annotation.

3.3 Text Processors

We need to transform our dataset into something that can be consumed by the LLM. Simply, we need to turn the text into numbers.

3.3.1 Tokenization

The importance of the tokenization is often ignored, but it plays a central role for every subsequent step for processing information by LLM.

Tokenization defines how raw text is decomposed into discrete units (tokens), which are then mapped to integer IDs and processed by the model. Because the model operates exclusively on

these token sequences, the tokenizer effectively determines what linguistic patterns the model can represent efficiently.

A well-designed tokenizer improves computational efficiency, maximizes effective context utilization, and reduces semantic fragmentation, thereby directly influencing both the quality and reliability of downstream model outputs.

Key concepts:

- Char v.s. Word v.s. Subword:
  - Character-level tokenizers split texts into single Unicode codepoints. It is simple but inefficient for long-range semantics.
  - Word-level tokenizers split on whitespace/punctuation; fast but poor for rare words and morphology.
  - Subword-level tokenizers (*e.g.,* BPE, WordPiece, SentencePiece) balance vocabulary size and OOV(Out of vocabulary) handling and are the most common for modern LLMs.
- Special tokens: reserve tokens for padding, beginning/end-of-sequence, unknowns, separators, and task-specific markers. Ensure consistency between tokenizer and model.
- Offsets and alignment: maintain offset mappings (token -> character span) when you need to map model outputs back to original text (useful for evaluation, annotation, highlighting).
- Token limits and chunking: handle long documents by sliding windows, overlapping chunks, or hierarchical encoders. Decide on truncation strategy (head, tail, or balanced) based on task.
- Determinism & reproducibility: fix preprocessing (lowercasing, stripping, normalization), and save tokenizer vocab + merges to reproduce training and inference behavior.

Practical tips:

- Use a byte-level BPE or SentencePiece for multilingual robustness and reproducibility.
- Tokenize and measure token distribution early to size your context windows and batch planning.
- Cache tokenized datasets (on disk or in memory) to speed up repeated training/evaluation runs.
- When fine-tuning, prefer the tokenizer that matches the pretrained weights to avoid embedding mismatches; if you must extend the vocab, initialize new embeddings carefully and validate downstream performance.

Common libraries:

- Hugging Face Tokenizers : A de facto standard tokenizer that offers a full-featured Python implementation for research workflows, alongside a highly optimized Rust implementation for production environments.
- sentencepiece : Google's standalone C++ based library for sentencepiece
- tiktoken : OpenAI's fast and efficient BPE tokenizer for GPT-family models

3.3.2 Embedding

While tokenization slices the text, embedding constructs the semantic bridge between human language and machine representation. For LLM practitioners, embeddings are not just about representing words; they are the backbone of Retrieval-Augmented Generation (RAG).

In a production environment, simply "turning text into numbers" is insufficient. We must consider the **architecture of the retrieval pipeline**, **inference latency**, and **storage costs**.

Contextualized Representations

Unlike static embeddings (*e.g.,* Word2Vec, GloVe) which map a unique token to a fixed vector, modern LLM-based embeddings are *contextual*. The vector representation of a token t_i depends on all other tokens $t_{1...n}$ in the sequence.

- **Pooling Strategies**: To represent a full sentence or document as a single vector (dense vector), we typically extract the output from the final layer of the Transformer.
  - `[CLS]` token: Using the vector of the special classification token (common in BERT-family models).
  - `Mean pooling`: Averaging the vectors of all tokens (often yields better semantic representations for clustering).
- **Instruction-Tuned Embeddings**: Modern SOTA models (*e.g.,* E5, Qwen3-Embedding series) require specific instructions or prefixes to distinguish between the query and the document (*e.g.,* adding `"query: "` or `"passage: "`).[1] Omitting these prompts can significantly degrade retrieval performance.

Optimization and Efficiency

As vector databases grow, storage and latency become bottlenecks. Two techniques are essential for production scaling:

1. **Matryoshka Representation Learning (MRL)**: Newer models (*e.g.,* OpenAI's `text-embedding-3` or Qwen3-Embedding series) are trained such that information is concentrated in lower-index dimensions. As a result, truncating a 4096-dimensional embedding to 2048 or even 1024 dimensions incurs minimal performance degradation, while significantly reducing storage and retrieval costs.
2. **Quantization**: Converting 32-bit floating-point vectors (FP32) to 8-bit integers (INT8) or even binary makes. While this introduces a small precision loss, it can reduce memory usage by 4× to 32× and significantly speed up similarity calculations.

[1] Providers often distinguish between symmetric tasks (like "text-matching" or "sentence similarity") and asymmetric tasks ("retrieval"), releasing separate model variants or prompts for each. This addresses the inherent asymmetry in RAG, where user queries are typically short and implicit, while documents are long and noisy.

Selecting the Right Model

Do not blindly use the default model provided by your cloud provider. Check the

- MTEB (Massive Text Embedding Benchmark) Leaderboard.

- **Task Specificity**: If you are building a semantic search engine, look at the *Retrieval* score. If you are doing intent classification, look at the *Clustering* or *Classification* scores.
- **Sequence Length**: Ensure the model's maximum context length (*e.g.,* 512 vs. 8192 tokens) matches your document chunking strategy. Long-context models are preferred for RAG to capture broader document semantics.

Part III

Training LLMs

Chapter 4

Training LLMs: How to generate the generator

4.1 Checkpointing

ef

During training, autograd needs forward activations (the intermediate tensors produced layer-by-layer) to compute gradients in the backward pass. Keeping all of them can blow up GPU memory—especially with long sequences, big batches, or deep networks.

Idea: don't keep everything. Save only a few checkpoints, and **recompute** the missing activations on the fly during backward. You trade extra compute for much lower memory.

4.1.1 Gradient/Activation Checkpointing

Checkpointing refers to a strategy that picks a few layers as checkpoints (say after $L_0, L_3, L_6, \dots$). Save only those. In backward, when you need activations inside ($L_3 \to L_6$), you re-run the forward from L_3 to L_6 to recreate them, then compute gradients.

4.2 Multi-GPU environments

Training is a resource-intensive endeavor. A model that only takes a single GPU to run inference on may take 10 times that many to train if, for nothing else, to parallelize your work and speed things up so you aren't waiting for a thousand years for it to finish training.

4.2.1 Setting up

It should be pointed out up front that while multi-GPU environments are powerful, they are also expensive. For the rest of us, setting up a virtual machine (VM) in Google's Compute Engine is one of the easiest methods.

Google Virtual Machine One of the easiest ways to create a multi-GPU environment is to set up a VM on Google's cloud.

1. Create a Google Cloud Project (GCP).
    - Set up billing
    - Download the gcloud CLI.
2. After setting up your account, GCP sets your GPU quotas to 0. Quotas are used to manage your costs. You need to increase to 2 or more, since we plan to use multiple GPUs.
3. Init by "`gcloud init`"
4.

4.2.2 Libraries

DeepSpeed: DeepSpeed is an optimization library for distributed deep learning. DeepSpeed is powered by Microsoft and implements various enhancements for speed in training and inference, like handling extremely long or multiple inputs in different modalities, quantization, caching weights and inputs, and, probably the hottest topic right now, scaling up to thousands of GPUs.

To install,

1. Install PyTorch
2. `pip install deepspeed`

Accelerate: From HuggingFace, Accelerate is made to help abstract the code for parallelizing and scaling to multiple GPUs away from you so that you can focus on the training and inference side.

To install,

1. `pip install accelerate`

PyTorch FSDP Install PyTorch with distributed support (CUDA version must match drivers):

```
pip install torch torchvision torchaudio

# (Optional) For speedups
pip install torchmetrics accelerate
```

Ensure passwordless SSH between nodes if you're on a bare-metal or HPC cluster.

Create `train_fsdp.py`:

```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.distributed as dist
from torch.distributed.fsdp import FullyShardedDataParallel as FSDP
```

```python
from torch.distributed.fsdp.wrap import transformer_auto_wrap_policy

# --- Simple Transformer block ---
class ToyBlock(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(1024, 4096)
        self.act = nn.ReLU()
        self.fc2 = nn.Linear(4096, 1024)

    def forward(self, x):
        return self.fc2(self.act(self.fc1(x)))

class ToyModel(nn.Module):
    def __init__(self, depth=6):
        super().__init__()
        self.layers = nn.Sequential(*[ToyBlock() for _ in range(depth)])

    def forward(self, x):
        return self.layers(x)

def main():
    dist.init_process_group("nccl")
    torch.cuda.set_device(dist.get_rank() % torch.cuda.device_count())

    model = ToyModel().cuda()

    # Auto-wrap large layers with FSDP
    auto_wrap_policy = transformer_auto_wrap_policy
    model = FSDP(model, auto_wrap_policy=auto_wrap_policy)

    optimizer = optim.AdamW(model.parameters(), lr=1e-4)

    for step in range(20):
        x = torch.randn(8, 1024).cuda()
        y = model(x).mean()
        y.backward()
        optimizer.step()
        optimizer.zero_grad()
        if dist.get_rank() == 0:
            print(f"Step {step} done.")

    dist.destroy_process_group()

if __name__ == "__main__":
    main()
```

If your node has 4 GPUs:

`torchrun -nproc_per_node=4 train_fsdp.py`

- 4 processes (one per GPU).
- NCCL backend handles GPU communication.

Let's try running multiple nodes

- Node 0: 10.0.0.1
- Node 1: 10.0.0.2

- 4 GPUs per node
- Total world size = 8 (2 nodes × 4 GPUs)

```
**On Node 0 (rank 0):**

torchrun --nnodes=2 --nproc_per_node=4 \
         --node_rank=0 \
         --master_addr=10.0.0.1 \
         --master_port=29500 \
         train_fsdp.py

**On Node 1 (rank 1):**

torchrun --nnodes=2 --nproc_per_node=4 \
         --node_rank=1 \
         --master_addr=10.0.0.1 \
         --master_port=29500 \
         train_fsdp.py
```

- `-nnodes=2`: total number of nodes.
- `-nproc_per_node=4`: GPUs per node.
- `-node_rank`: each node's unique index.
- `-master_addr`: IP/hostname of rank 0 node.
- `-master_port`: open port for coordination.

- Activation checkpointing: Saves memory by discarding intermediate activations during forward pass and recomputing them during backward pass.
- Mixed precision: Uses FP16 or BF16 for computations (faster, less memory) while keeping FP32 for stability in some ops.

Full Example with FSDP + Checkpointing + AMP:

```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.distributed as dist
from torch.distributed.fsdp import FullyShardedDataParallel as FSDP
from torch.distributed.fsdp.wrap import transformer_auto_wrap_policy
from torch.utils.checkpoint import checkpoint

# --- A block with activation checkpointing ---
class CheckpointedBlock(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(1024, 4096)
        self.act = nn.ReLU()
        self.fc2 = nn.Linear(4096, 1024)

    def forward(self, x):
        def forward_fn(x):
            return self.fc2(self.act(self.fc1(x)))
        # checkpoint will discard activations & recompute in backward
```

```
        return checkpoint(forward_fn, x)

# --- Toy Model ---
class ToyModel(nn.Module):
    def __init__(self, depth=6):
        super().__init__()
        self.layers = nn.Sequential(*[CheckpointedBlock() for _ in range(depth)
    ])

    def forward(self, x):
        return self.layers(x)

def main():
    dist.init_process_group("nccl")
    torch.cuda.set_device(dist.get_rank() % torch.cuda.device_count())

    # Build model
    model = ToyModel().cuda()
    auto_wrap_policy = transformer_auto_wrap_policy
    model = FSDP(model, auto_wrap_policy=auto_wrap_policy)

    optimizer = optim.AdamW(model.parameters(), lr=1e-4)

    # Use mixed precision autocast (bf16 preferred if hardware supports it)
    scaler = torch.cuda.amp.GradScaler(enabled=True)  # works for fp16

    for step in range(20):
        x = torch.randn(8, 1024).cuda()

        with torch.cuda.amp.autocast(dtype=torch.bfloat16):  # or torch.float16
            y = model(x).mean()

        # backward with gradient scaler
        scaler.scale(y).backward()
        scaler.step(optimizer)
        scaler.update()
        optimizer.zero_grad()

        if dist.get_rank() == 0:
            print(f"Step {step} done.")

    dist.destroy_process_group()

if __name__ == "__main__":
    main()
```

- Checkpointing: Wrap the forward function with `torch.utils.checkpoint.checkpoint`.
- AMP (Automatic Mixed Precision):
  - Use `torch.cuda.amp.autocast` for forward pass.
  - Use `torch.cuda.amp.GradScaler` for loss scaling (needed for FP16, not BF16).
- BF16 vs FP16:
  - Use BF16 if your GPUs are A100/H100 (more stable).
  - Use FP16 + GradScaler for V100 or older cards.

- Full State Dict: Gather the full parameters on rank 0 and save them. (simpler, larger files).

- Sharded State Dict: Each rank saves only its shard. (efficient, but needs all shards to reload).
- Rank 0 only writing: Typically only rank 0 writes to disk to avoid file conflicts.

```python
import os
import torch
from torch.distributed.fsdp import FullyShardedDataParallel as FSDP
from torch.distributed.fsdp import StateDictType, FullStateDictConfig

CHECKPOINT_DIR = "./checkpoints"

def save_checkpoint(model, optimizer, step):
    # Ensure only rank 0 writes the file
    rank = torch.distributed.get_rank()
    os.makedirs(CHECKPOINT_DIR, exist_ok=True)

    # Switch to FULL state dict (gathered on rank 0)
  # offload_to_cpu=True: keeps memory usage down when gathering full state.
    full_sd_config = FullStateDictConfig(offload_to_cpu=True, rank0_only=True)
    with FSDP.state_dict_type(model, StateDictType.FULL_STATE_DICT,
    full_sd_config):
        model_state = model.state_dict()
        optim_state = optimizer.state_dict()

    if rank == 0: # only rank 0 writes to disk.
        save_path = os.path.join(CHECKPOINT_DIR, f"step_{step}.pt")
        torch.save({"model": model_state, "optimizer": optim_state, "step":
    step}, save_path)
        print(f"[Rank 0] Saved checkpoint to {save_path}")
```

```python
def load_checkpoint(model, optimizer, load_path):
    # Load only on rank 0
    rank = torch.distributed.get_rank()
    map_location = "cpu" if rank == 0 else "meta"  # meta avoids OOM on other
    ranks
    checkpoint = torch.load(load_path, map_location=map_location)

    # Use FULL state dict context
    full_sd_config = FullStateDictConfig(offload_to_cpu=True, rank0_only=True)
    with FSDP.state_dict_type(model, StateDictType.FULL_STATE_DICT,
    full_sd_config):
        model.load_state_dict(checkpoint["model"])

    # Broadcast model weights to all ranks
    torch.distributed.barrier()
    optimizer.load_state_dict(checkpoint["optimizer"])
    print(f"[Rank {rank}] Loaded checkpoint from {load_path}")
```

- Up to 13B on few GPUs → FSDP (simpler, first-party).
- >30B or MoE, multi-node → Megatron-LM + (FSDP or DeepSpeed ZeRO).
- When GPU memory is tight → DeepSpeed ZeRO-3 (with offload).

4.3 Basic Training Techniques

Unlike traditional ML models, LLMs are often trained in stages:

Library	**Core Idea**	**Strengths / Use Cases**
DeepSpeed	Distributed training engine with ZeRO (Zero Redundancy Optimizer) sharding	- ZeRO-1/2/3: memory savings via parameter/gradient/optimizer sharding - Offloading to CPU/NVMe - Mixture of Experts (MoE) support - Great for very large dense or MoE models
FSDP (Fully Sharded Data Parallel)	Native PyTorch module for full parameter, gradient, optimizer sharding	- First-party, stable, integrated in PyTorch - ZeRO-3–like memory scaling - Easy to use with Hugging Face Accelerate/Lightning
Megatron-LM	Parallelism library from NVIDIA (TP, PP, SP, EP)	- Tensor Parallelism (TP) for matmuls - Pipeline Parallelism (PP) for layer distribution - Sequence/Expert Parallelism for long-context and MoE - Standard for 30B–100B+ scale

Table 4.1: Comparison of Core Libraries for Multi-GPU LLM Training

- Optuna: Open source hyperparameter optimization (HPO) framework for machine learning, including Large Language Models (LLMs).

Chapter 5

Parameter Efficient Fine-Tuning

5.1 LoRA

When fine-tuning a neural network for a new task, we adjust its weights W by learning an update ΔW, yielding

$$W' \;=\; W + \Delta W.$$

LoRA (Low-Rank Adaptation) avoids updating W directly. Motivated by the **intrinsic rank hypothesis**-the idea that task-critical changes lie in a low-dimensional subspace-LoRA parameterizes the update as a low-rank product:

$$\Delta W \;=\; BA, \qquad W' \;=\; W + BA,$$

while keeping W frozen. Here $B \in \mathbb{R}^{d\times r}$ and $A \in \mathbb{R}^{r\times d}$ with $r \ll d$, so BA is a low-rank approximation to the full update.

This factorization dramatically reduces trainable parameters. Instead of learning all d^2 entries of ΔW (for a $d \times d$ weight), LoRA learns only the factors B and A, totaling $2dr$ parameters—much smaller when $r \ll d$. (The same idea applies to non-square $W \in \mathbb{R}^{d\times k}$, using $B \in \mathbb{R}^{d\times r}$ and $A \in \mathbb{R}^{r\times k}$, for a cost of $r(d + k)$ instead of dk.)

Increasing r usually:

- Improves task performance (up to a point)
- Costs more memory + compute
- Can overfit more easily on small datasets

5.2 QLoRA

While LoRA reduces the number of *trainable* parameters, it still requires storing and using the full-precision base model W during fine-tuning. For very large LLMs (tens or hundreds of billions of parameters), this memory demand can exceed the capacity of a single GPU.

QLoRA (Quantized Low-Rank Adaptation) addresses this by combining LoRA with parameter *quantization*. Instead of keeping W in 16- or 32-bit precision, QLoRA stores it in a lower-bit format (typically 4-bit). During training:

- The frozen base weights W are kept in 4-bit quantized form, greatly reducing memory usage.
- On-the-fly, W is *dequantized* into higher precision (e.g., 16-bit) for computations.
- As in LoRA, trainable low-rank adapters A, B are introduced to capture task-specific updates.

The update rule remains

$$W' \;=\; W + BA,$$

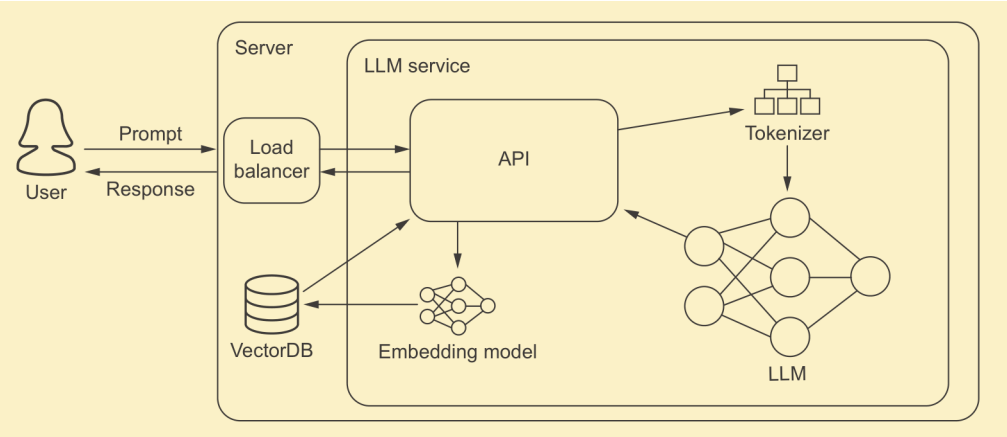
but now W is quantized, while A, B are small full-precision matrices.

Part IV

LLM Services

Chapter 6

LLM Services: A Practical Guide



Production refers to the phase where the model is integrated into live or operational environment to perform its intended tasks or provide services to end users. It's a crucial phase in making the model available for real-world applications and services. In this chapter, we will explore how to package up an LLM into a service or API so that it can take on-demand requests. Then, we will study how to set up a cluster in the cloud where you can deploy this service.

6.1 Creating an LLM service

6.1.1 Model Compilation

The success of any model in production is dependent on the hardware it runs on. Unfortunately, when programming in a high-level language like Python-based frameworks like PyTorch or TensorFlow, the model won't be optimized to take full advantage of the hardware. This is where compiling comes into play. Compiling is the process of taking code written in a high-level language and converting or lowering it to machine-level code that the computer can process quickly. Compiling your LLm can easily lead to major inference and cost improvements.

Kernel Tuning In DL, and high-performance computing, a kernel is a small program or function designed to run on a GPU or other similar processors. These routines are developed by the hardware vendor to maximize chip efficiency.

During kernel tuning, the most suitable kernels are chosen from a large collection of highly optimized kernels.

Kernel Fusion A GPU kernel is the tiny function that runs on each element. **Fusion** = do several elementwise ops in one kernel so each element is read from VRAM [1] once, processed in registers, then written back once.

The `y = ReLU(x + b)` example

Assume `x` and `b` are the same length (or `b` is broadcast on the last dim).
Without fusion (two kernels: add, then ReLU):

1. For every element i:
2. Read x[i] from VRAM
3. Read b[i] from VRAM
4. Compute t = x[i] + b[i]
5. Write t to VRAM ← intermediate array
6. Read t from VRAM
7. Compute y = max(t, 0)
8. Write y to VRAM

Global-memory ops per element: 5 (read x, read b, write t, read t, write y) Two kernel launches (one for add, one for ReLU).
With fusion (one kernel: add+ReLU together)

1. For every element i:
2. Read x[i] from VRAM
3. Read b[i] from VRAM
4. Compute t = x[i] + b[i] (kept in a register, not VRAM)
5. Compute y = max(t, 0)
6. Write y to VRAM

Global-memory ops per element: 3 (read x, read b, write y) One kernel launch.

ReLU Example:

```
import torch, time

B, C = 4096, 4096
x = torch.randn(B, C, device="cuda", dtype=torch.float32)
b = torch.randn(C,     device="cuda", dtype=torch.float32)

def add_relu_unfused(x, b):
```

[1] VRAM is technically a type of DRAM, but it is optimized for storing images and video data that the GPU (Graphics Processing Unit) requires for processing various graphics. It is primarily responsible for storing textures, frame buffers, image data, video data, and other graphics-related data [**?**].

```
    # Eager mode typically launches two kernels (add, then relu)
    return torch.relu(x + b)

# PyTorch 2.x compiler (Inductor) generates fused kernels automatically
# This fuses common elementwise chains (like  a d d ReLU ) into single kernels.
add_relu_fused = torch.compile(add_relu_unfused, backend="inductor")

# Correctness check
with torch.no_grad():
    y1 = add_relu_unfused(x, b)
    y2 = add_relu_fused(x, b)
    print("max |diff| =", (y1 - y2).abs().max().item())

# Simple timing
def bench(fn, iters=50, warmup=10):
    for _ in range(warmup):
        fn(x, b); torch.cuda.synchronize()
    t0 = time.perf_counter()
    for _ in range(iters):
        fn(x, b)
    torch.cuda.synchronize()
    return (time.perf_counter() - t0) * 1000 / iters

print("Unfused  :", bench(add_relu_unfused), "ms/iter")
print("Fused    :", bench(add_relu_fused),  "ms/iter")
```

Graph Optimization Graph optimization = semantics-preserving rewrites of the op graph (fold, fuse, simplify, relayout, retype, reschedule) so the lowered kernels do less work with less memory traffic

TensorRT NVIDIA TensorRT is an SDK that converts a trained model into an optimized "engine" and then runs that engine with very low latency/high throughput on NVIDIA GPUs. It includes an optimizer (compiler) and a lightweight runtime.

How it works (typical workflow):

1. Import your model (usually ONNX; there are parsers and framework bridges).
2. Build an optimized engine: TensorRT selects fast kernels ("tactics"), fuses layers, lowers precision (FP16/INT8) if allowed, and specializes to your shapes/hardware.
3. Serialize the engine to a .plan file (so you can load it instantly in production).
4. Run it via the TensorRT runtime (C++/Python).

```
import tensorrt as trt

logger  = trt.Logger(trt.Logger.WARNING)
builder = trt.Builder(logger)
network = builder.create_network(1 << int(trt.NetworkDefinitionCreationFlag.
    EXPLICIT_BATCH))
parser  = trt.OnnxParser(network, logger)

with open("model.onnx", "rb") as f:
    assert parser.parse(f.read()), parser.get_error(0)

config = builder.create_builder_config()
```

```
config.set_memory_pool_limit(trt.MemoryPoolType.WORKSPACE, 4 << 30)  # workspace budget
config.set_flag(trt.BuilderFlag.FP16)  # enable mixed precision if supported

# Dynamic-shape profile for input "input" (name must match your ONNX)
profile = builder.create_optimization_profile()
profile.set_shape("input", min=(1,3,224,224), opt=(8,3,224,224), max=(32,3,224,224))
config.add_optimization_profile(profile)

engine = builder.build_engine(network, config)
with open("model.plan", "wb") as f:
    f.write(engine.serialize())
```

ONNX Runtime **ONNX**, which stands for Open Neural Network Exchange, is an open source format and ecosystem designed for representing and interoperating between different deep learning frameworks. It was created to address the challenge of model portability and compatibility. ONNX is an IR (intermediate representation) and it allows you to represent models trained in some deep learning framework (*e.g.,* TensorFlow, PyTorch) in a standardized format easily consumed by other frameworks and it facilitates the exchange of models between different tools and environments. Unlike TensorRT, ONNX Runtime is intended to be hardware-agnostic, meaning it can be used with a variety of hardware accelerators, including CPUs, GPUs, and specialized hardware like TPUs.

6.1.2 LLM storage strategies

Now we have a compiled model, we need to think about how our service will access it.This step is critical, because boot times can be a nightmare when working with LLMs since it can take a long time to load such large assets into memory.

Object storage systems break up assets into small franctional bits called objects. They allow us to federate the entire asset across multiple machines and physical memory locations, a powerful tool that powers the clound, and to cheaply store large objects on hardware.

Fusing is the process of mounting a bucket to your machine as if it were an external hard drive.

Baking the Model Baking is the process of putting your model into the Docker image. It is considered an *anti-pattern*.

- Gigantic images: every update means pushing/pulling multi-GB layers → slow CI/CD, slow Kubernetes rollouts, higher storage costs.
- Tight coupling: a new model version requires a full image rebuild and redeploy (harder A/B tests, slower rollbacks, no "one image, many models").
- Poor caching: one byte change in weights invalidates a huge layer; your build cache won't help much.
- Security issue

Hybrid: download once, reuse many Download the model at boot time but store it in a volume that is mounted at boot time. While this doesn't help at all with the first deployment in a region, it does substantially help any new instances, as they can simply mount this same volume and have the model available to load without having to download.

- At boot (when the service starts up), your service (or an init step) pulls the model from S3/MinIO/HF/etc.
- It stores the files on a persistent volume that's mounted into the container.
- Subsequent pods/containers on the same node (or across nodes if using a shared RWX volume) just mount the volume—no re-download.
- You keep your app image small and decouple model updates from image builds.

6.1.3 Adaptive request batching

A typical API will accept and process requests in the order they are received, processing them immediately and as quickly as possible. However, anyone who's trained a ML model has come to realize that there are mathematical and computational advantages to running inference in batches of powers of 2 (16, 32, 64, etc), particularly when GPUs are involved, where we can take advantage of better memory alignment or vectorized instructions parallelizing computations across the GPU cores.

Why power of 2 is better Reference: CUDA C++ Best Practices Guide

https://datascience.stackexchange.com/questions/20179/what-is-the-advantage-of-keeping-batch-size-a-power-of-2

What adaptive batching does is essentially pool requests together over a certain period of time. Once the pool receives the configured maximum batch size or the timer runs out, it will run inference on the entire batch through the model, sending the results back to the individual clients that requested them. Essentially, it's a queue. Setting one up yourself can and will be a huge pain; thankfully, most ML inference services offer this out of the box, and almost all are easy to implement. For example, in **BentoML**, add `@bentoml.Runnable.method(batchable=True)` as a decorator to your predict function, and in **Triton Inference Server**, add `dynamic_batching{}` at the end of your model definition file.

If that sounds easy, it is. Typically, you don't need to do any further finessing, as the defaults tend to be very practical. That said, if you are looking to maximize every bit of efficiency possible in the system, you can often set a maximum batch size, which will tell the batcher to run once this limit is reached, or a batch delay, which does the same thing but for the timer. Increasing either will result in longer latency but likely better throughput, so typically these are only adjusted when your system has plenty of latency budget.

Overall, the benefits of adaptive batching include better use of resources and higher throughput at the cost of a bit of latency. This is a valuable trade-off, and we recommend giving your product the latency bandwidth to include this feature. In our experience, optimizing for throughput leads to better reliability and scalability and thus greater customer satisfaction. Of course, when latency times are extremely important or traffic is few and far between, you may rightly forgo this feature.

6.1.4 Flow Control

Rate limiters and access keys are critical protections for an API, especially one sitting in front of an expensive LLM. Rate limiters control the number of requests a client can make to an API within a specified time, which helps protect the API server from abuse, such as distributed denial of service (DDoS) attacks, where an attacker makes numerous requests simultaneously to overwhelm the system and hinder its function.

Rate limiters can also protect the server from bots that make numerous automated requests in a short span of time. This helps manage the server resources optimally so the server is not exhausted due to unnecessary or harmful traffic. They are also useful for managing quotas, thus ensuring all users have fair and equal access to the API's resources. By preventing any single user from using excessive resources, the rate limiter ensures the system functions smoothly for all users.

All in all, rate limiters are an important mechanism for controlling the flow of your LLM's system processes. They can play a critical role in dampening bursty workloads and preventing your system from getting overwhelmed during autoscaling and rolling updates, especially when you have a rather large LLM with longer deployment times. Rate limiters can take several forms, and the one you choose will be dependent on your use case.

Types of rate limiters

- Fixed window: allow a fixed number of requests in a set duration of time.
  - Problem: Burst at window boundaries:
    - User makes 100 requests at 12:59 PM
    - User makes 100 requests at 1:00 PM
    - Result: 200 requests in 1 minute! (violates the spirit of the limit)
- Sliding window log: Keep a log of all request timestamps. For each new request, remove timestamps older than the window duration, then check if we're under the limit.
  - Example: Allow 100 requests per hour
    - Maintain a list of timestamps: [12:05, 12:10, 12:15, ...]
    - New request at 1:20 PM $\rightarrow$ Remove all timestamps before 12:20 PM
    - If remaining count $<$ 100, allow the request
- Token bucket: Clients initially have a full bucket of tokens, and with each request, they spend tokens. When the bucket is empty, the requests are blocked.
- Leaky bucket: Requests enter a queue (bucket). Queue is processed at constant rate. If queue is full, new requests are rejected.

A rate limiter can be applied at multiple levels, from the entire API to individual client requests to specific function calls.

```
from typing import Optional

from fastapi import FastAPI, Depends, HTTPException, status, Request
from fastapi.security import APIKeyHeader
from slowapi import Limiter, _rate_limit_exceeded_handler
from slowapi.errors import RateLimitExceeded
```

```python
from slowapi.util import get_remote_address

# ----- Config -----------------------------------------------------------
# In real apps, load from a DB or env/secret manager.
VALID_KEYS = {"1234567abcdefg"}

# Displayed in Swagger UI as the auth "scheme" name
api_key_header = APIKeyHeader(name="X-API-Key", scheme_name="APIKey",
    auto_error=False)

# Prefer per-key limiting; fall back to client IP (useful when no key provided)
def rate_key(request: Request) -> str:
    key = request.headers.get("X-API-Key")
    return f"key:{key}" if key else f"ip:{request.client.host}"

# Use in-memory storage for single-process dev.
# For multi-workers/replicas, switch to: storage_uri="redis://redis:6379/0"
limiter = Limiter(key_func=rate_key, storage_uri="memory://")

# ----- App setup --------------------------------------------------------
app = FastAPI(title="API-Key + Rate Limit Example")
app.state.limiter = limiter
app.add_exception_handler(RateLimitExceeded, _rate_limit_exceeded_handler)

# If running behind a reverse proxy/load balancer, trust X-Forwarded-*.
# In production, replace ["*"] with your proxy host/IP(s).

# ----- Auth dependency --------------------------------------------------
async def require_api_key(api_key: Optional[str] = Depends(api_key_header)) ->
    str:
    """
    Validates the X-API-Key header against our allow-list.
    Returns the key (or user id associated with it) for downstream use.
    """
    if not api_key or api_key not in VALID_KEYS:
        # RFC-friendly header (helps some clients know how to auth)
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Invalid or missing API key",
            headers={"WWW-Authenticate": "APIKey"},
        )
    return api_key

# ----- Routes -----------------------------------------------------------
@app.get("/health", include_in_schema=False)
async def health() -> dict:
    return {"ok": True}

@app.get("/hello")
@limiter.limit("5/minute")            # per-key if present; else per-IP
async def hello(request: Request, api_key: str = Depends(require_api_key)):
    # SlowAPI needs `request`; dependency injects validated `api_key`.
    return {"message": "Hello World"}

# Optional: run directly with `python main.py`
if __name__ == "__main__":
    import uvicorn
    uvicorn.run("main:app", host="127.0.0.1", port=8000, reload=True)
```

- You (the API owner) issue a unique key (*i.e.*, `VALID_KEYS`) to each user/app.
- The client keeps that key and sends it with every request.

- Because the client passes the key in an HTTP header. Your example uses an `X-API-Key` header via `APIKeyHeader`.
- The dependency `require_api_key(...)` reads the header, checks it against `VALID_KEYS`, and returns 401 if it's missing/invalid.
- The rate limiter's `key_func` (`rate_key`) also reads the same header to bucket requests per key.

To test in SwaggerUI:

- Click Authorize.
- You'll see a field for APIKey (because APIKeyHeader is used).
- Paste "1234567abcdefg" and authorize

LiteLLM LLM proxy server

6.1.5 Streaming responses

Streaming allows us to return the generated text to the user as it is being generated versus all at once at the end.

6.1.6 Feature store

When it comes to running ML models in production, feature stores really simplify the inference process. Feature stores establish a centralized source of truth. They answer crucial questions about your data:

Who is responsible for the feature? What is its definition? Who can access it? Let's take a look at setting one up and querying the data to get a feel for how they work. We'll be using Feast, which is open source and supports a variety of backends. To get started, let us `pip install feast` and then run the `init` command in your terminal to set up a project, like so:

The app we are building is a question-and-answer service. Q&A services can greatly benefit from a feature store's data governance tooling. For example, point-in-time joins help us answer questions like "Who is the president of x?" where the answer is expected to change over time. Instead of querying just the question, we *query the question with a timestamp*, and the point-in-time join will return whatever the answer to the question was in our database at that point in time. In the next listing, we pull a Q&A dataset and store it in a parquet format in the data directory of our Feast project.

```
import pandas as pd
from datasets import load_dataset
import datetime

from sentence_transformers import SentenceTransformer
model = SentenceTransformer("all-MiniLM-L6-v2")

def save_qa_to_parquet(path):
```

```
    squad = load_dataset("squad", split="train[:5000]")
    ids = squad["id"]
    questions = squad["question"]
    answers = [answer["text"][0] for answer in squad["answers"]]
    qa = pd.DataFrame(
        zip(ids, questions, answers),
        columns=["question_id", "questions", "answers"],
    )
    qa["embeddings"] = model.encode(questions) # feature
    qa["created"] = utcnow() # ingest time
    qa["datetime"] = qa["created"].dt.floor("h") # Event time (when the new
    fact becomes true)
    qa.to_parquet(path) # offline store file

if __name__ == "__main__":
    path = "./data/qa.parquet"
    save_qa_to_parquet(path)
```

- `DataFrame.to_parquet` in pandas is a method used to write a pandas DataFrame to the binary Parquet file format.
  - Parquet is a columnar storage format optimized for efficient data storage and retrieval, especially in big data environments.
- Creating a tiny feature table with:
  - Entity key: question_id
  - Features: embeddings (and you could treat answers, questions as features for the demo)
  - Time columns: datetime (event time), created (ingest time)

```
from feast import Entity, FeatureView, Field, FileSource, ValueType
from feast.types import Array, Float32, String
from datetime import timedelta

path = "./data/qa.parquet"

question = Entity(name="question_id", value_type=ValueType.STRING)

question_feature = Field(name="questions", dtype=String)

answer_feature = Field(name="answers", dtype=String)

embedding_feature = Field(name="embeddings", dtype=Array(Float32))

questions_view = FeatureView(
    name="qa",
    entities=[question],
    ttl=timedelta(days=1),
    schema=[question_feature, answer_feature, embedding_feature],
    source=FileSource(
        path=path,
        event_timestamp_column="datetime",
        created_timestamp_column="created",
        timestamp_field="datetime",
    ),
    tags={},
    online=True,
)
```

6.1.7 Retrieval-augmented generation

6.1.8 LLM service libraries

If you are starting to feel a bit overwhelmed about all the tooling and features you need to implement to create an LLM service, there are several libraries aim to do all of this for you! Some open source libraries of note are **vLLM** and **OpenLLM** (by BentoML).

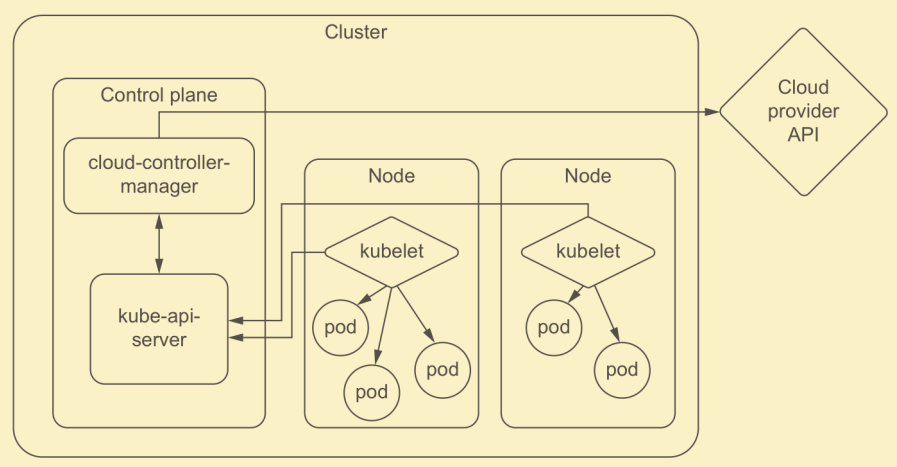
Most of these toolings are still relatively new and under active development, and they are far from feature parity with each other, so pay attention to what they offer.

6.2 Setting up infrastructure

Setting up infrastructure is a critical aspect of modern software development, and we shouldn't expect machine learning to be any different. To ensure scalability, reliability, and efficient deployment of our applications, we need to plan a robust infrastruc- ture that can handle the demands of a growing user base. This is where **Kubernetes** comes into play.

Kubernetes, often referred to as k8s, is an open source container orchestration platform that helps automate and manage the deployment, scaling, and management of containerized applications. It is designed to simplify the process of running and coordinating multiple containers across a cluster of servers, making it easier to scale applications and ensure high availability. We are going to talk a lot about k8s in this chapter, and while you don't need to be an expert, it will be useful to cover some basics to ensure we are all on the same page.

At its core, k8s works by grouping containers into logical units called *pods*, which are the smallest deployable units in the k8s ecosystem. These **pods are then scheduled and managed by the k8s control plane** (*i.e.,* the brain of Kubernetes), which oversees their deployment, scaling, and updates. This control plane consists of several components that collectively handle the orchestration and management of containers.



Control Plane Node vs Worker Node:

- Control plane nodes are dedicated to running Kubernetes system components that manage the cluster

  - API server: this is the gate to talk to k8s.
  - Scheduler: Decides which worker node will run your pod.
  - Controller manager: Watches everything and fixes it if it drifts from your request.
- Worker nodes run your application workloads. Each node is a computer (physical or virtual). It runs:
  - Kubelet: The assistant on each node. This talks to the control plane and makes sure the right containers (*i.e.,* pod) are running.

How it works?

- You tell Kubernetes: "Run 3 pods of my web app."
- API Server receives the request.
- Scheduler decides where to place pods.
- Controller Manager keeps checking until 3 pods are alive.
- Kubelet on each worker starts the containers.
- Kube-Proxy sets up networking so users can reach your app.

6.2.1 Provisioning clusters

The first thing to do when starting any project is to set up a *cluster*. A cluster is a collective of worker machines or nodes where we will host our applications.

- On GCP:

```
gcloud container clusters create <NAME>
```

- On AWS

```
eksctl create cluster
```

- Provisioning a cluster is like building an "empty factory": control room (control plane), power, network, and a small set of machines (nodes) to start with. In short, it creates and prepares the cluster.
- Node Auto-Provisioning (NAP) is an auto-hire manager that watches your job queue (unschedulable Pods) and, when there aren't enough machines, buys and installs the right new machines automatically—and removes them later when work slows down. In short, it dynamically adds new nodes/pools to an existing cluster (not create the cluster itself).

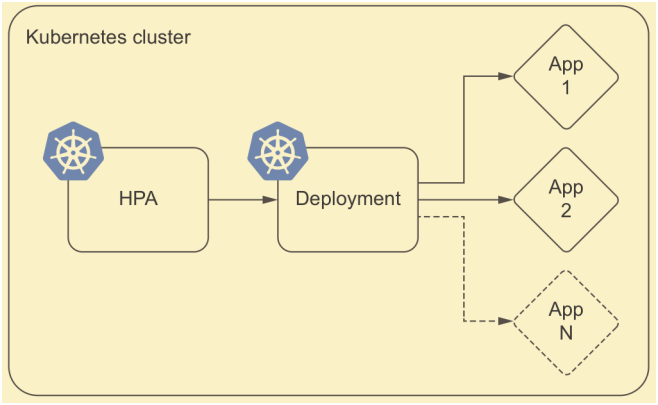


Figure 6.1: Basic autoscaling using the in-built k8s HPA. The HPA watches CPU and memory resources and will tell the deployment service to increase or decrease the number of replicas.

6.2.2 Autoscaling

One of the big selling points to setting up a k8s cluster is *autoscaling*. Autoscaling is an important ingredient in creating robust production-grade services. The main reason is that **we never expect any service to receive static request volume**.

Autoscaling and NAP

Autoscaling can be considered as a family of mechanisms at different layers, and NAP is one specific (infrastructure) member of that family.

The HPA (horizontal pod autoscaler) watches CPU and memory resources and will tell the deployment service to increase or decrease the number of replicas.

The first service we'll need is one that can **collect the GPU metrics** for autoscaling. We can use *NVIDIA's Data Center GPU Manager (DCGM)*, which provides a metrics exporter that can export GPU metrics. DCGM exposes a host of GPU metrics, including temperature and power usage, which can create some fun dashboards, but the most useful metrics for autoscaling are utilization and memory utilization.

From here, the data will go to a service like *Prometheus*. Prometheus is an open source monitoring system used to monitor Kubernetes clusters and the applications running on them. Prometheus collects metrics from various sources and stores them in a time-series database, where they can be analyzed and queried. Prometheus can collect metrics directly from Kubernetes APIs and from applications running on the cluster using a variety of collection mechanisms such as exporters, agents, and sidecar containers. **It's essentially an aggregator of services like DCGM**, including features like alerting and notification. It also exposes an HTTP API for service for external tooling like Grafana to query and create graphs and dashboards with.

While Prometheus provides a way to store metrics and monitor our service, the metrics aren't exposed to the internals of Kubernetes. For an HPA to gain access, we will need to register yet another service to either the custom metrics API or external metrics API. By default, Kubernetes comes with the *metrics.k8s.io* endpoint that exposes resource metrics, CPU, and memory utilization. To accommodate the need to scale deployments and pods on custom metrics, two additional APIs were introduced: *custom.metrics.k9s.io* and *external.metrics.k8s.io*. There are some limitations to this setup, as currently, only one "adapter" API service can be registered at a time for either one. This limitation mostly becomes a problem if you ever decide to change

this endpoint from one provider to another.

- Prometheus is a time-series database that collects and stores metrics.
- Grafana is a dashboarding and visualization tool that queries Prometheus (and other data sources) to create visual representations.

For this service, Prometheus provides the *Prometheus Adapter*, which works well, but from our experience, it wasn't designed for production workloads. Alternatively, we would recommend *KEDA*. KEDA (Kubernetes Event-Driven Autoscaling) is an open source project that provides event-driven autoscaling for Kubernetes. It offers more flexibility in terms of the types of custom metrics that can be used for autoscaling.

While Prometheus Adapter requires configuring metrics inside a ConfigMap, any metric already exposed through the Prometheus API can be used in KEDA, providing a more streamlined and friendly user experience. It also offers scaling to and from 0, which isn't available through HPAs, allowing you to turn off a service completely if there is no traffic. That said, you can't scale from 0 on resource metrics like CPU and memory and, by extension, GPU metrics, but it is useful when you are using traffic metrics or a queue to scale.

Putting this all together, you'll end up with the architecture shown in the following figure. You'll notice at the bottom that DCGM is managing our GPU metrics and feeding them into Prometheus Operator. From Prometheus, we can set up external dashboards with tools like Grafana. Internal to k8s, we'll use KEDA to set up a custom.metrics.k9s.io API to return these metrics so we can autoscale based on the GPU metrics. KEDA has several CRDs, one of which is a ScaledObject, which creates the HPA and provides the additional features.

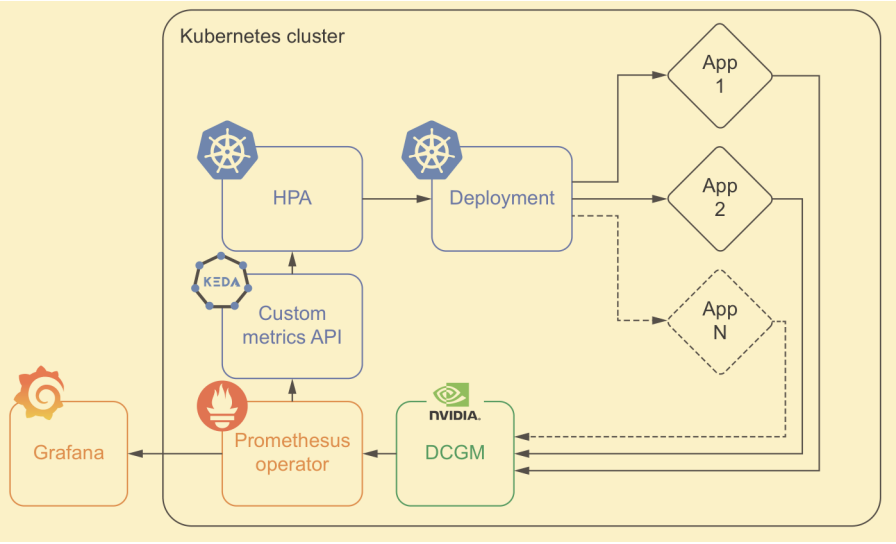


Figure 6.2: Autoscaling based on a custom metric like GPU utilization.

While autoscaling provides many benefits, it's important to be aware of its limitations and potential problems, which are only exacerbated by LLM inference services. Proper configuration of the HPA is often an afterthought for many applications, but it becomes mission-critical when dealing with LLMs. LLMs take longer to become fully operational, as the GPUs need to be initialized and model weights loaded into memory; these aren't services that can turn on a dime, which often can cause problems when scaling up if not properly prepared for. Additionally, if the system scales down too aggressively, it may result in instances being terminated before

completing their assigned tasks, leading to data loss or other problems. Lastly, flapping is just such a concern that can arise from incorrect autoscaling configurations. Flapping happens when the number of replicas keeps oscillating, booting up a new service only to terminate it before it can serve any inferences.

There are essentially five parameters to tune when setting up an HPA:

- Target parameter
- Target threshold
- Min pod replicas
- Max pod replicas
- Scaling policies

Summary You want Kubernetes to **add or remove Pods (and maybe nodes) based on GPU load** (*e.g.,* how busy the GPU is, how much GPU memory is used).

- DCGM (NVIDIA): the thermometer for your GPUs. It reads GPU stats (utilization, memory used, temperature, power) and exposes them as metrics.
- Prometheus: the notebook + calculator. It scrapes (collects) those DCGM metrics and stores them so you can query/graph/alert.
- Adapter (KEDA or Prometheus Adapter): the translator for Kubernetes. It takes Prometheus metrics and exposes them to Kubernetes' autoscaling APIs so HPAs can use them.
  - By default, Kubernetes only knows basic resource metrics (CPU & memory) via metrics.k8s.io.
  - To scale on custom metrics (like GPU utilization), you must expose them via: custom.metrics.k8s.io (object-scoped) or external.metrics.k8s.io (external/global), through one adapter per API (limitation of the design).
- HPA (Horizontal Pod Autoscaler): the replica dial. It looks at a metric (e.g., GPU utilization) and scales Pods up/down.
- Cluster-level scaling (CA/NAP/Karpenter): the machine buyer. If more Pods need room, these can add nodes (and later remove them). NAP can even create new node *types* if needed.

The flow with GPUs:

- DCGM exports GPU metrics.
- Prometheus scrapes and stores them.
- KEDA (or Prometheus Adapter) publishes those metrics to Kubernetes as autoscaling metrics.
- HPA reads that metric and scales Pod replicas.
- If Pods don't fit on current nodes, Cluster Autoscaler / NAP may add nodes (and later remove them).

Target Threshold The target parameter is the most important metric to consider when ensuring your system is properly configured. Scaling based on GPU utilization is going to be the most common and straightforward path, but the first thing we need to do is ensure the GPU is the actual bottleneck in our service. It's pretty common to see eager young engineers throw a lot of expensive GPUs onto a service but **forget to include adequate CPU and memory capacity**. CPU and memory will still be needed to handle the API layer, such as taking in requests, handling multiple threads, and communicating with the GPUs. If there aren't enough resources, these layers can quickly become a bottleneck, and your application will be throttled way before the GPU utilization is ever affected, ensuring the system will never actually autoscale. While you could switch the target parameter on the autoscaler, CPU and memory are cheap compared to GPU resources, so it'd be better to allocate more of them for your application.

In addition, there are cases where other metrics make more sense. If your LLM application takes most of its requests from a streaming or batch service, it can be more prudent to scale based on metrics that tell you a DAG is running or an upstream queue is filling up-especially if these metrics give you an early signal and allow you more time to scale up in advance.

Another concern when selecting the metric is its stability. For example, an indi- vidual GPU's utilization tends to be close to either 0% or 100%. This can cause problems for the autoscaler, as the metric oscillates between an on and off state, as will its recommendation to add or remove replicas, causing flapping. Generally, flapping is avoided by taking the average utilization across all GPUs running the service. Using the average will stabilize the metric when you have a lot of GPUs, but it could still be a problem when the service has scaled down. If you are still running into problems, you'll want to use an average-over-time aggregation, which will tell you the utilization for each GPU over a time frame—say, the last 5 minutes. For CPU utilization, average-over-time aggregation is built into the Kubernetes HPA and can be set with the horizontal-pod-autoscaler-cpu-initialization-period flag. For custom metrics, you'll need to set it in your metric query (for Prometheus, it would be the `avg_over_time` aggregation function).

6.2.3 Rolling Updates

Rolling updates or rolling upgrades is a strategy that gradually implements the new version of an application to reduce downtime and maximize agility. It works by **gradually creating new instances and turning off the old ones**, replacing them in a methodical manner. This update approach allows the system to remain functional and accessible to users even during the update process, otherwise known as zero downtime. Rolling updates also make it easier to catch bugs before they have too much effect and rollback faulty deployments.

Rolling updates is a feature built into k8s and another major reason for its wide-spread use and popularity. Kubernetes provides an automated and simplified way to carry out rolling updates.

6.2.4 Inference Graphs

Inference graphs are the crème filling of a donut, the muffin top of a muffin, and the toppings on a pizza: they are just phenomenal. Inference graphs allow us to create sophisticated flow diagrams at inference in a resource-saving way. Consider figure 6.8, which shows us the building blocks for any inference graph.

Generally, any time you have more than one model, it's useful to consider an infer- ence graph architecture. Your standard LLM setup is usually already at least two models: an encoder and

the language model itself.

Usually, when we see LLMs deployed in the wild, these two models are deployed together. You send text data to your system, and it returns generated text. It's often no big deal, but when deployed as a sequential inference graph instead of a packaged service, we get some added bonuses. First, the encoder is usually much faster than the LLM, so we can split them up since you may only need one encoder instance for every two to three LLM instances. Encoders are so small that this doesn't necessarily help us out that much, but it saves the hassle of redeploying the entire LLM if we decide to deploy a new encoder model version. In addition, an inference graph will set up an individual API for each model, which allows us to hit the LLM and encoder separately.

This is really useful if we have a bunch of data we'd like to preprocess and save in a VectorDB; we can use the same encoder we already have deployed. We can then pull this data and send it directly into the LLM.

The biggest benefit of an inference graph is that it allows us to separate the API and the LLM. The API sitting in front of the LLM is likely to change much more often as you tweak prompts, add features, and fix bugs. The ability to update the API with out having to deploy the LLM will save your team a lot of effort.

Let's now consider figure 6.9, which provides an example inference graph deploy- ment using Seldon. In this example, we have an encoder model, an LLM, a classifier model, and a simple API that combines the results. Whereas we would have to build a container and the interface for each of these models, Seldon creates an orchestrator that handles communication between a user's request and each node in the graph.

Summary Don't ship your entire AI pipeline as one big black-box service. Instead, split it into small model services and wire them together as a graph at inference time.

6.2.5 Monitoring

6.3 Production Challenges

6.3.1 Model updates and retraining

Watching your model for silent failures and data drift, but what do you do when you notice the model starts failing?

Many traditional ML implementations that the answer is to simply retrain the model on the latest data and redeploy. And that works well when you are working with a small model.

we aren't going to retrain from scratch, and we likely need to finetune our model, but the reason it doesn't make sense is seen when we ask ourselves just what exactly the latest data is.

The data we need to finetune the model is extremely important, and so it becomes necessary for us to take a step back and really diagnose the problem. What are the edge cases our model is failing on? What is it still doing well? How exactly have incoming prompts changed? Depending on the answers, we might not need to finetune at all.

For example, consider a Q&A bot that is no longer effective at answering current event questions as time goes on. We probably don't want to retrain a model on a large corpus of the latest news articles. Instead, we would get much better results by ensuring our RAG system is up to date. Similarly, there are likely plenty of times that simply tweaking prompts will do the trick.

6.3.2 Load testing

6.3.3 Troubleshooting poor latency

One of the biggest bottlenecks when it comes to your model's performance in terms of latency and throughput has nothing to do with the model itself but comes from **data transmission of the network**. One of the simplest methods to improve this I/O constraint is to **serialize the data before sending it across the wire**, which can have a large effect on ML workloads where the payloads tend to be larger, including LLMs where prompts tend to be long.

To serialize the data, we utilize a framework known as *Google Remote Procedure Call (gRPC)*. gRPC is an API protocol similar to REST, but instead of sending JSON objects, we **compress the payloads into a binary serialized format using Protocol Buffers**, also known as *protobufs*. By doing this, we can send more information in fewer bytes, which can easily give us orders of magnitude improvements in latency. Luckily, most inference services will implement gRPC along with their REST counterparts right out of the box, which is extremely convenient since the major hurdle to using gRPC is setting it up.

A major reason for this convenience is the Seldon V2 Inference Protocol, which is widely implemented. The only hurdle, then, is ensuring our client can serialize and deserialize messages to take advantage of the protocol. In listing 6.13, we show an example client using MLServer to do this. It's a little bit more in depth than your typi- cal curl request, but a closer inspection shows the majority of the complexity is simply converting the data from different types as we serialize and deserialize it.

6.3.4 Resource management

6.3.5 Cost engineering

It is often paid through longer inference times and extra compute time. In fact, it's been suggested that simply adding "be concise" to your prompt can save you up to 90% of your costs.

You'll also save a lot by using text embeddings.

You also need to consider which model you should use for which task.

You also need to consider which model you should use for which task. Generally, bigger models are better at a wider variety of tasks, but if a smaller model is good enough for a specific job, you'll save a lot by using it.

You'll need to consider the costs of GPUs to use when running. In general, you'll want to use the least amount of GPUs needed to fit the model into memory to reduce the cost of idling caused by bubbles

Part V

RAG

Chapter 7

Retrieval Augmented Generation

7.1 Introduction

As large language models (LLMs) began to see serious commercial adoption around 2023, interest in the concept of Retrieval Augmented Generation (RAG) grew alongside them. This period marked one of the first moments when LLMs moved beyond simply imitating existing styles—such as writing poems—or serving as somewhat limited conversational partners, and instead led companies to believe that these models could be applied in genuinely useful, real-world scenarios.

Following the release of ChatGPT, people—particularly those in industry—started to continuously explore how this technology might contribute to creating business value. Encountering well-trained (pretrained) LLMs, many were impressed, and at times even unsettled, by responses that felt strikingly "human." Yet this impression did not last long before the limitations became clear. Models lack awareness of information beyond their training cutoff, cannot access internal documents or organizational policies, and have no inherent understanding of the most recent data. On top of that, they frequently suffer from hallucinations, generating outputs that sound convincing but are not factually correct.

From a corporate perspective, these shortcomings are more than academic concerns—they are critical constraints. For deployment in real business environments, systems must ensure accuracy, timeliness, and the reliability of their information sources. A model that merely "sounds articulate" carries significant risk when used for customer-facing applications or internal decision-support tools. Inevitably, this led to a fundamental question: How can we safely make use of information that the model itself does not know?

Retrieval Augmented Generation (RAG) emerged as a practical and compelling answer. At its core, RAG is built on a simple idea: instead of forcing the model to internalize all possible knowledge during training, retrieve the relevant external information at inference time and supply it alongside the prompt. In this setup, the LLM is not treated as a vast knowledge store, but rather as a reasoning and language-generation engine, while factual knowledge and data are managed by external systems.

In the following sections, we will examine the key components of a RAG system—document collection and preprocessing, embeddings and vector stores, retrieval strategies, and prompt design during generation. Along the way, we will explore why this architecture works so well in practice, and where its limitations and failure modes tend to appear.

Part VI

Prompt Engineering

Chapter 8

Prompt Engineering

8.1 Prompting your model

8.1.1 Few-shot prompting

8.1.2 One-shot prompting

8.1.3 Zero-shot prompting

8.2 Prompt engineering basics

Part VII

A Systems View for LLMs

Chapter 9

Understanding GPUs

Chapter 10

Model Efficiency

You can understand efficiency of your deep learning regime as consisting of 3 different components.

- **Computation**: Time spent on your GPU computing actual *floating point operations per second (FLOPS)*
- **Bandwidth**: Time spent transferring tensors within a GPU
- **Overhead**: Everything else

10.1 Computation

You can't really shrink the number of operations like multiplies/adds without changing the model/algorithm (*e.g.,* smaller layers, pruning, low-rank tricks).

But you can often shrink time by reducing fewer memory trips, better packing, fusing steps, batching requests, and so on.

Hardware keeps getting faster than it speeds up IO operations. FLOPs (chef speed) roughly double quicker than memory bandwidth.

Suppose your GPU peaks at 300 TFLOPs and 1 TB/s memory bandwidth.

- To fully use 300 TFLOPs, your *kernels*[1] need $\geq$ 300 FLOPs per byte (roofline rule: performance $\leq$ min(peak FLOPs, bandwidth $\times$ FLOPs/byte)).
- If your workload only does 50 FLOPs/byte, memory limits you to $\sim$ 50 TFLOPs no matter how beefy the chefs are-the pantry can't feed them fast enough.

What this means for LLMs (simple takeaways):

- **Prefill** (big matmuls): often compute-bound $\rightarrow$ use mixed precision (FP8/16), fuse ops, and batch to keep math units hot.

[1] A kernel is a single function the GPU runs in parallel (*e.g.,* "do this matmul," "add two tensors," "apply GELU"). Your model is a chain of many kernels.

- **Decode** (one token at a time): often memory-bound due to KV-cache reads → shrink bytes (KV quantization, MQA/GQA), fuse tiny ops, batch tokens across requests, speculative decoding to raise work per memory fetch.
- Across the board: arrange data contiguously, prefetch, use CUDA graphs/compilers, and avoid tons of micro-kernels.

10.2 Bandwidth

Bandwidth cost is the time and energy spent moving data. In deep-learning kernels, that can mean:

- CPU ↔ GPU transfers,
- Node ↔ node transfers (network),
- GPU DRAM ↔ on-chip compute (global memory ↔ registers/shared memory (SRAM)).

Our focus is on the last one. It dominates many GPU operations: moving tensors in and out of GPU DRAM [2] can be far more expensive than the math itself. That's why memory-bound kernels (little math per byte moved) often run much slower than you'd expect.

Memory bandwidth cost

Each time you launch a GPU kernel, intermediate results are typically written back to global memory and read again by the next kernel and it is called memory bandwidth cost. If each step does only a tiny bit of math, you're mostly paying for reads/writes and these operations are called *memory-bound operations*.

10.2.1 Kernel Fusion (Operator Fusion)

Kernel fusion (or Operator fusion) combines multiple ops so intermediates stay on-chip instead of bouncing to DRAM. That cuts global reads/writes and lifts you out of the memory-bound regime.

[2] `nvidia-smi` shows up GPU's DRAM, and DRAM is the primary resource of causing CUDA Out of Memory errors.

Point-wise operation example

Point-wise operation refers to an operation that each output element depends only on the corresponding input element, with no interaction across elements.
There are 4 global accesses.

```
x1 = x.cos()   # read x,   write x1
x2 = x1.cos()  # read x1,  write x2
```

With fusion (2 global accesses total):

```
x2 = x.cos().cos()  # read x once, write x2 once
```

Cutting global traffic like this is often close to a 2× speedup for simple pointwise chains. In `x.cos()`, for every index `i`, the kernel computes `y[i] = cos(x[i])`. There's no neighborhood, no reduction, no matrix multiply - each element is handled in isolation. Let's consider a simple example:

In short, kernel fusion do several steps in one pass. Instead of: "do op A → write result to memory → read it back → do op B", we combine them so data stays on-chip while we keep computing. That avoids slow trips to (global) memory.

Most kernels are limited by memory bandwidth, not raw FLOPs. If we don't have to write/read intermediates, we save bandwidth and time. That's why two separate PyTorch ops are a chance to fuse and speed things up.

One of counterintuitive observation is that the computational costs of `x.cos().cos()` and `x.cos()` are similar once fused. This is because, the expensive part is often moving x through memory. If you keep the value in registers and apply cos twice before writing, the extra math is tiny compared to the avoided memory traffic. This is also why GELU vs ReLU are closer in runtime than you'd expect: after fusion, both mostly cost the same memory movement.

There are a couple of caveats that make this a bit tricky.

1. The GPU needs to know what's going to happen next when performing the current operation. So, you can't do this optimization in eager-mode, where PyTorch runs operators one operation at a time.
2. We actually need to generate CUDA code for this, which opens up a whole new can of worms.

Example

Think of two "speed limits" on a GPU:

1. Memory bandwidth — how fast you can move data to/from global memory.
2. Compute throughput — how fast you can do math.

On an A100:

- Bandwidth ≈ 1.5 TB/s $= 1.5 \times 10^{12}$ bytes/s.
- Compute ≈ 19.5 TFLOP/s $= 19.5 \times 10^{12}$ float ops/s.
- A float32 is 4 bytes.

We can only read: 1.5×10^{12} bytes/s /4 bytes $\approx$ 375B numbers/s.
For something like $y = 2 \times x$, each element:

- read 4 bytes (x),
- write 4 bytes (y).
- Total 8 bytes per element.

So max elements/s $\approx 1.5 \times 10^{12}/8 = 187.5$B elems/s.
This operations is memory-bound. Each element does ≈ 1 flop (a multiply). If you can process at most 187.5B elems/s, that's $\approx$ 187.5GFLOP/s. This is tiny compared to 19.5 TFLOP/s. The math units are mostly waiting on memory.
To use the GPU's compute fully, you need high *arithmetic intensity* [a] (many flops per byte), *e.g.,* matrix multiplications.

[a] how much math you do per byte of data moved from main memory.

10.3 Overhead

Overhead is everything that isn't real math or moving tensors. For example, time spent in the Python interpreter? Overhead. Time spent in the PyTorch framework? Overhead. Time spent launching CUDA kernels.

This is a problem since, GPUs are insanely fast. If the GPU can do hundreds of trillions of FLOPs per second but Python can only do tens of millions, then any time you spend in Python (or framework layers) is like idling a rocket engine at a red light. For tiny ops, the setup time dominates; the math itself is over in a blink.

If you do lots of small ops (*e.g.,* add a few numbers many times), you pay the overhead each time. The GPU's compute is barely used, so your speed is capped by Python/framework/kernal-launch overhead, not by hardware.

Practical fixes (what to do):

- Batch work / make tensors bigger. Fewer, larger ops amortize overhead.
- Fuse ops. Do more math per read/write (*e.g.,* fuse activation, bias, scale).

- Vectorize in Python. Replace Python loops with tensor ops; avoid per-element Python.
- Use torch.compile (PyTorch 2+). Lets the compiler capture graphs, fuse, and lower overhead.
- Prefetch/async where possible. Overlap transfers and compute.
- If CPU-bound and tiny arrays: consider NumPy or even C++ for critical loops.
- Custom kernels (*e.g.,* Triton/CUDA) when patterns are regular and hot.

Part VIII

LLM Inference

Chapter 11

Understanding LLM Inference

Most widely used decoder-only LLMs (*e.g.,* GPT) are trained with a causal language modeling objective-effectively, next-token prediction. Given an input token sequence, they generate the continuation autoregressively until a stopping criterion is reached (*e.g.,* a maximum length) or a special <END> token is produced. The inference process naturally splits into two stages: (*i*) the *prefill* phase, which reads users inputs and (*ii*) the *decode* phase, which generates responses.

Tokens. Tokens are the smallest units the model consumes. A common rule of thumb is that one token corresponds to roughly four English characters (the exact ratio depends on the tokenizer). Text is always tokenized before entering the model.

11.1 Prefill Phase

During prefill, the model ingests the entire input sequence (*e.g.,* user/system prompt) and constructs the attention memory—namely the keys and values, often called the *KV cache*—to be reused during generation. Because the full input is available from the start, this stage parallelizes well: it resembles a matrix–matrix workload and typically drives the GPU close to peak utilization.

11.2 Decoding Phase

Decoding then emits output one token at a time. Each new token must attend to all tokens seen so far (the original prompt plus previously generated outputs), which makes this step inherently sequential. At each step, the computation behaves more like matrix–vector multiplication and generally uses the GPU less efficiently than prefill.

For every token produced, the model must *fetch* a substantial portion of the KV cache from GPU memory. As the context grows, the data transferred per step increases, so latency is governed more by memory bandwidth than by raw compute—decoding is thus *memory-bound*.

Intuition. Imagine typing an answer character by character, but before each keystroke you skim all prior notes to remain consistent. The typing (compute) is quick; the skimming (memory

access) is what slows you down.

For instance, with a 1,000-token prompt, generating 200 tokens means token #1 attends over roughly 1,000 tokens, while token #200 attends over roughly 1,200 tokens.

Because decoding is the usual bottleneck, many inference optimizations focus here: more efficient attention mechanisms, improved KV-cache handling (*e.g.,* paging, quantization, head sharing), and strategies such as speculative or look-ahead decoding that reduce sequential work.

11.3 Batching

The simplest way to improve GPU utilization, and effectively throughput, is through batching. Since multiple requests use the same model, the memory cost of the weights is spread out. Larger batches getting transferred to the GPU to be processed all at once will leverage more of the compute available.

Batch sizes, however, can only be increased up to a certain limit, at which point they may lead to a memory overflow. To better understand why this happens requires looking at key-value (KV) caching and LLM memory requirements.

Traditional batching (also called static batching) is suboptimal. This is because for each request in a batch, the LLM may generate a different number of completion tokens, and subsequently they have different execution times. As a result, all requests in the batch must wait until the longest request is finished, which can be exacerbated by a large variance in the generation lengths. There are methods to mitigate this, such as in-flight batching, which will be discussed later.

11.4 KV-Caching

One common optimization for the decode phase is *KV caching*. The decode phase generates a single token at each time step, but each token depends on the key and value tensors of all previous tokens (including the input tokens' KV tensors computed at prefill, and any new KV tensors computed until the current time step).

To avoid recomputing all these tensors for all tokens at each time step, it's possible to cache them in GPU memory. Every iteration, when new elements are computed, they are simply added to the running cache to be used in the next iteration. In some implementations, there is one KV cache for each layer of the model.

11.5 LLM memory requirement

In effect, the two main contributors to the GPU LLM memory requirement are model weights and the KV cache:

- **Model weights**: Memory is occupied by the model parameters. As an example, a model with 7 billion parameters (such as Llama 2 7B), loaded in 16-bit precision (FP16 or BF16) would take roughly 7B $\times$ sizeof(FP16) $\approx$ 14 GB in memory.

- **KV caching**: Memory is occupied by the caching of self-attention tensors to avoid redundant computation.

With batching, the KV cache of each of the requests in the batch must still be allocated separately, and can have a large memory footprint. The formula below delineates the size of the KV cache, applicable to most common LLM architectures today.

$$\text{Size of KV cache per token in bytes} = 2 \times (\text{n\_layers}) \times (\text{n\_heads} \times \text{dim\_head}) \times (\text{precision\_in\_bytes})$$

- The first factor of 2 accounts for the K and V matrices.
- Commonly, the value of (num_heads $\times$ dim_head) is equal the hidden_size (or dimension of the model, d_model) of the transformer. These model attributes are commonly found in model cards or associated config files.

This memory size is required for each token in the input sequence, across the batch of inputs.

For example, with a Llama2 7B model in 16-bit precision and a batch size of 1, the size of the KV cache will be $1 \times 4096 \times 2 \times 32 \times 4096 \times 2$ bytes, where 4096 is the sequence length. In sum, it takes around 2 GB.

Managing this KV cache efficiently is a challenging endeavor. Growing linearly with batch size and sequence length, the memory requirement can quickly scale. Consequently, it limits the throughput that can be served, and poses challenges for long-context inputs. This is the motivation behind several optimizations featured.

11.6 Scaling up LLMs with model parallelization

One way to reduce the per-device memory footprint of the model weights is to distribute the model over several GPUs. Spreading the memory and compute footprint enables running larger models, or larger batches of inputs. Model parallelization is a necessity to train or infer on a model requiring more memory than available on a single device, and to make training times and inference measures (latency or throughput) suitable for certain use cases. There are several ways of parallelizing the model based on how the model weights are split.

- Pipeline parallelism
- Tensor parallelism
- Sequence parallelism

11.6.1 Pipeline parallelism

Pipeline parallelism involves sharding the model (vertically) into chunks, where each chunk comprises **a subset of layers that is executed on a separate device**.

The main limitation of this method is that, due to the sequential nature of the processing, some devices or layers may remain idle while waiting for the output (activations, gradients) of previous layers. This results in inefficiencies or *pipeline bubbles* in both the forward and backward passes.

11.6.2 Tensor Parallelism

Tensor parallelism involves sharding (horizontally) individual layers of the model into smaller, independent blocks of computation that can be executed on different devices. Attention blocks and multi-layer perceptron (MLP) layers are major components of transformers that can take advantage of tensor parallelism. In multi-head attention blocks, each head or group of heads can be assigned to a different device so they can be computed independently and in parallel.

11.6.3 Sequence Parallelism

Tensor parallelism has limitations, as it requires layers to be divided into independent, manageable blocks. It's not applicable to operations like LayerNorm and Dropout, which are instead replicated across the tensor-parallel group. While LayerNorm and Dropout are computationally inexpensive, they do require a considerable amount of memory to store (redundant) activations.

Sequence parallelism (SP) splits work along the sequence length dimension across multiple GPUs. Instead of every GPU holding all tokens of each sequence, each GPU holds a slice of tokens (*e.g.,* tokens 0–255 on GPU0, 256–511 on GPU1). It's usually used together with tensor/model parallelism to cut activation memory and enable longer context or larger batches.

11.7 Optimizing the attention mechanism

- Multi-head attention
- Multi-query attention
- Grouped-query attention
- Flash attention

11.8 Model optimization techniques

11.8.1 Quantization

Quantization is the process of reducing the precision of a model's weights and activations. Most models are trained with 32 or 16 bits of precision, where each parameter and activation element takes up 32 or 16 bits of memory—a single-precision floating point. However, most deep learning models can be effectively represented with eight or even fewer bits per value.

Reducing the precision of a model can yield several benefits. If the model takes up less space in memory, you can fit larger models on the same amount of hardware. Quantization also means you can transfer more parameters over the same amount of bandwidth, which can help to accelerate models that are bandwidth-limited.

There are many different quantization techniques for LLMs involving reduced precision on either the activations, the weights, or both. It's much more straightforward to quantize the weights because they are fixed after training. However, this can leave some performance on the table because the activations remain at higher precisions. GPUs don't have dedicated hardware for

multiplying INT8 and FP16 numbers, so the weights must be converted back into a higher precision for the actual operations.

It's also possible to quantize the activations, the inputs of transformer blocks and network layers, but this comes with its own challenges. Activation vectors often contain outliers, effectively increasing their dynamic range and making it more challenging to represent these values at a lower precision than with the weights.

One option is to find out where those outliers are likely to show up by passing a representative dataset through the model, and choosing to represent certain activations at a higher precision than others (LLM.int8()). Another option is to borrow the dynamic range of the weights, which are easy to quantize, and reuse that range in the activations.

11.8.2 Sparsity

Similar to quantization, it's been shown that many deep learning models are robust to pruning, or replacing certain values that are close to 0 with 0 itself. Sparse matrices are matrices where many of the elements are 0. These can be expressed in a condensed form that takes up less space than a full, dense matrix.

GPUs in particular have hardware acceleration for a certain kind of structured sparsity, where two out of every four values are represented by zeros. Sparse representations can also be combined with quantization to achieve even greater speedups in execution. Finding the best way to represent large language models in a sparse format is still an active area of research, and offers a promising direction for future improvements to inference speeds.

11.8.3 Distillation

11.9 Model Serving Techniques

11.9.1 In-Flight Batching

11.9.2 Speculative inference

Chapter 12

Flash Attention

"*Many approximate attention methods have aimed to reduce the compute and memory requirements of attention. These methods range from sparse-approximation to low-rank approximation, and their combinations. Although these methods reduce the compute requirements to linear or near-linear in sequence length, many of them do not display wall-clock speedup against standard attention and have not gained wide adoption. One main reason is that they focus on FLOP reduction (which may not correlate with wall-clock speed) and tend to ignore overheads from memory access (IO)*" [**?**].

- Fast — excerpt from the paper: "We train BERT-large (seq. length 512) 15% faster than the training speed record in MLPerf 1.1, GPT2 (seq. length 1K) 3x faster than baseline implementations from HuggingFace and Megatron-LM, and long-range arena (seq. length 1K-4K) 2.4x faster than baselines."
- Memory-efficient — compared to vanilla attention, which is quadratic in sequence length, $O(N^2)$, this method is sub-quadratic/linear in $O(N)$. We'll see later why & how.
- Exact — meaning it's not an approximation of the attention mechanism (like *e.g.,* sparse, or low-rank matrix approximation methods) — its outputs are the same as in the "vanilla" attention mechanism.
- IO aware — compared to vanilla attention, flash attention is sentient.

Over the years GPUs have been adding compute capacity (FLOPS) at a faster pace than increasing the memory throughput (TB/s).

It doesn't matter if you can compute at exaFLOPS speeds if there is no data to be processed. These 2 need to be closely aligned, and since the hardware lost that balance we have to make our software compensate for it.

It turns out attention is (on current AI accelerators) memory-bound.

- Softmax
- Dropout
- Masking

- Matmul

Modern GPUs have a strict memory hierarchy:

- SRAM – fast, on-chip, small
- HBM – slower than SRAM, large size. That's what we usually address as GPU memory.

Every operation that "does math" must move data from HBM up to SRAM and then write results back. Those transfers are not free. For attention, the cost of reading and writing dominates; the kernels often become bandwidth/IO-bound rather than compute-bound.

The problem with the vanilla attention is that attention forms the full score matrix:

$$S = \frac{QK^\top}{\sqrt{d_k}}$$

There are several issues:

- Materialization of S: S is a $N \times N$ matrix. For long sequences, just storing and moving S overwhelms SRAM capacity and HBM bandwidth.
- SRAM limits: To compute the softmax for one query token i, you'd like all scores on chip.

In short: the IO pattern is wasteful.

FlashAttention makes attention IO-aware by restructuring computation to minimize reads/writes between HBM and SRAM while staying exact:

- Tile the sequence: Load small blocks (tiles) of Q, K, and V that fit in SRAM. Compute partial QK for that tile only.
- Online/streaming softmax: For each query row i, keep tiny running statistics in SRAM
- Write once: When a row is complete (all tiles processed), write the final output to HBM. You avoid the ($O(N^2)$) intermediate traffic.

However, this matrix is necessary for transformer training as it is a part of backpropagation and gradient calculation. The authors propose that it's better to recalculate this matrix during the backward pass (again without explicit materialization). Not only does this saves lots of memory, but it also provides huge speedups as we don't need to transfer this enormous matrix between different GPU memory types.

Overall, such an approach did not only speed up calculations by taking GPU I/O specifics into account, but also allowed processing huge sequence lengths as memory complexity drops to $O(n)$.

In sum,

- Load a small block of Q and K into SRAM.
- Compute just that block of scores ($Q \cdot K^T$).

- Do a streaming softmax: keep a running max and sum so softmax stays numerically stable without needing all tokens at once.
- Immediately apply that softmax block to the matching V block and accumulate partial outputs.
- Move to the next tile. When all tiles are processed, you already have the final output—no big attention matrix ever stored.

This is called an IO-aware algorithm: it minimizes slow memory traffic and maximizes use of the GPU's fast memory.

What attention normally does (and why it's slow)

Given per-head matrices:

- $Q \in \mathbb{R}^{N\times d}, K \in \mathbb{R}^{N\times d}, V \in \mathbb{R}^{N\times d_v}$
- Scores: $S = \frac{QK^\top}{\sqrt{d}}$(size $N \times N$)
- Output: $O = \text{softmax}(S)V$

Naive kernels materialize S (size (N^2)), apply softmax row-wise, then multiply by V. Problem: writing/reading (N^2) scores to HBM (GPU DRAM) is *memory-bound* and explodes memory as N grows (*e.g.,* $4k$ tokens $\rightarrow 16M(4k^2)$ scores per head).

The core idea of FlashAttention is to compute over tiles. If we compute matrix multiplications over small blocks or tiles, we can significantly reduce the amount of memory access.

However, to compute the attention scores, we still have to compute the softmax. To compute the softmax over tiles, we can adopt online softmax and never form S in HBM. In other words, do attention in tiles that fit in on-chip SRAM to minimize HBC traffic, and keep only tiny per-row summaries in HBM. For each tile:

1. Load a block of Q and a block of K, V into SRAM.
2. Compute partial scores $S_{\text{blk}} = Q_{\text{blk}}K_{\text{blk}}^\top/\sqrt{d}$.
3. Apply a streaming softmax update so you don't need the whole row at once.
4. Immediately multiply by V_{blk} and accumulate partial outputs for that row block.
5. Move to the next block of K/V and repeat.

- $\frac{M}{4d}$: Note that query, key, and value vectors are d-dimensional. We also need to combine them into the d-dimensional output vector. To load all four vectors, it has to be $4d$.

Algorithm 1 FlashAttention

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM, on-chip SRAM of size M.

1: Set block sizes $B_c = \lceil \frac{M}{4d} \rceil$, $B_r = \min\left(\lceil \frac{M}{4d} \rceil, d\right)$.
2: Initialize $\mathbf{O} = (0)_{N \times d} \in \mathbb{R}^{N \times d}$, $\ell = (0)_N \in \mathbb{R}^N$, $m = (-\infty)_N \in \mathbb{R}^N$ in HBM.
3: Divide $\mathbf{Q}$ into $T_r = \left\lceil \frac{N}{B_r} \right\rceil$ blocks $\mathbf{Q}_1, \ldots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide $\mathbf{K}, \mathbf{V}$ in to $T_c = \left\lceil \frac{N}{B_c} \right\rceil$ blocks $\mathbf{K}_1, \ldots, \mathbf{K}_{T_c}$ and $\mathbf{V}_1, \ldots, \mathbf{V}_{T_c}$, of size $B_c \times d$ each.
4: Divide $\mathbf{O}$ into T_r blocks $\mathbf{O}_i, \ldots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide ℓ into T_r blocks $\ell_i, \ldots, \ell_{T_r}$ of size B_r each, divide m into T_r blocks $m_1, \ldots, m_{T_r}$ of size B_r each.
5: **for** $1 \le j \le T_c$ **do**
6: Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.
7: **for** $1 \le i \le T_r$ **do**
8: Load $\mathbf{Q}_i, \mathbf{O}_i, \ell_i, m_i$ from HBM to on-chip SRAM.
9: On chip, compute $\mathbf{S}_{ij} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.
10: On chip, compute $\tilde{m}_{ij} = \text{rowmax}(\mathbf{S}_{ij}) \in \mathbb{R}^{B_r}$, $\tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij} - \tilde{m}_{ij}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\tilde{\ell}_{ij} = \text{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.
11: On chip, compute $m_i^{\text{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}$, $\ell_i^{\text{new}} = e^{m_i - m_i^{\text{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.
12: Write $\mathbf{O}_i \leftarrow \text{diag}(\ell_i^{\text{new}})^{-1}(\text{diag}(\ell_i) e^{m_i - m_i^{\text{new}}} \mathbf{O}_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\mathbf{P}}_{ij} \mathbf{V}_j)$ to HBM.
13: Write $\ell_i \leftarrow \ell_i^{\text{new}}$, $m_i \leftarrow m_i^{\text{new}}$ to HBM.
14: **end for**
15: **end for**
16: Return $\mathbf{O}$.

12.1 The streaming (online) softmax trick

Note that a naive implementation of softmax can suffer from overflow issue, since softmax uses exponentials, and exponentials blow up (or vanish) fast. The standard way to fix this issue is to shift by the max (*i.e.,* log-sum-exp trick).

Softmax is shift-invariant:

$$\text{softmax}(x)_i = \text{softmax}(x - c)_i \quad \text{for any constant } c.$$

Choose $c = m = \max_j x_j$. Define $\tilde{x}_i = x_i - m \le 0$. Then

$$\text{softmax}(x)_i = \frac{e^{x_i - m}}{\sum_j e^{x_j - m}} = \frac{e^{\tilde{x}_i}}{\sum_j e^{\tilde{x}_j}}.$$

For one output row i (i-th query token), the softmax over all keys $j = 1 \ldots N$ is:

$$o_i = \sum_{j=1}^{N} \frac{e^{s_{ij}}}{\sum_{k=1}^{N} e^{s_{ik}}} v_j, \quad s_{ij} = \frac{q_i \cdot k_j}{\sqrt{d}} + \text{mask/bias}.$$

To compute how much a particular i-th token from the input sequence pays attention to other tokens in the sequence, you'd need to have all of those scores (*i.e.,* denominator) readily available (denoted here by s_{ij}) in SRAM. However, SRAM is limited in its capacity. Thus, we process keys in chunks (*i.e.,* tiles). Maintain per-row running stats. In other woreds, We process keys $j = 1, \ldots, N$ in chunks (tiles). For each row i keep running statistics:

- m_i: running max score seen so far (for numerical stability)
- ℓ_i: running sum of exp-shifted scores, *i.e.,* $\ell_i = \sum e^{s_{ij} - m_i}$

- z_i: running weighted sum of values, $z_i = \sum e^{s_{ij} - m_i} v_j$

When you see a new block with per-row block max $m_i^{\text{blk}} = \max_j s_{ij}^{\text{blk}}$ and sums

- $\ell_i^{\text{blk}} = \sum_j e^{s_{ij}^{\text{blk}} - m_i^{\text{blk}}}$,
- $z_i^{\text{blk}} = \sum_j e^{s_{ij}^{\text{blk}} - m_i^{\text{blk}}} v_j$,

update with running stats:

$$\begin{aligned} m_i^{\text{new}} &= \max(m_i, m_i^{\text{blk}}), \\ \ell_i^{\text{new}} &= \ell_i e^{m_i - m_i^{\text{new}}} + \ell_i^{\text{blk}} e^{m_i^{\text{blk}} - m_i^{\text{new}}}, \\ z_i^{\text{new}} &= z_i e^{m_i - m_i^{\text{new}}} + z_i^{\text{blk}} e^{m_i^{\text{blk}} - m_i^{\text{new}}}. \end{aligned}$$

At the end of all blocks:

$$o_i = \frac{z_i}{\ell_i}.$$

classic log-sum-exp fusion with dynamic max shifting keeps numbers well-scaled without needing all s_{ij} at once.

Example

- scores (two tiles): $s = [1, 3|0, 2]$
- 2-D values:
  - $(v_1 = [1, 0])$
  - $(v_2 = [0, 2])$
  - $(v_3 = [-1, 1])$
  - $(v_4 = [3, 1])$

Define $a = e^{-2} \approx 0.135335, b = e^{-1} \approx 0.367879$.

- **Tile 1**: $[1, 3]$ with$[v_1, v_2]$
  - block max: $m^{\text{blk}} = 3$
  - block sums (shift by m^{blk}):

$$\ell^{\text{blk}} = e^{1-3} + e^{3-3} = a + 1$$
$$z^{\text{blk}} = av_1 + 1 \cdot v_2 = a[1, 0] + [0, 2] = [a, , 2]$$

  - merge into running stats (start from $m = -\infty, \ell = 0, z = 0$):

$$m \leftarrow 3, \quad \ell \leftarrow a + 1, \quad z \leftarrow [a, 2].$$

- **Tile 2**: $[0, 2]$ with$[v_3, v_4]$
  - block max: $m^{\text{blk}} = 2$
  - block sums:

$$\ell^{\text{blk}} = a + 1, \quad z^{\text{blk}} = av_3 + 1 \cdot v_4 = a[-1, 1] + [3, 1] = [3 - a, 1 + a].$$

  - merge (global max stays $m^{\text{new}} = \max(3, 2) = 3$):

$$\ell \leftarrow \ell \cdot e^{3-3} + \ell^{\text{blk}} \cdot e^{2-3} = (a + 1) + (a + 1), b \approx \boxed{1.55300179}$$
$$z \leftarrow z \cdot e^{3-3} + z^{\text{blk}} \cdot e^{2-3} = [a, 2] + b[3 - a, 1 + a] \approx \boxed{[1.18918654, 2.41766651]}.$$

- **Finalize** (elementwise divide by scalar ℓ)

$$o = \frac{z}{\ell} \approx \left[\frac{1.18918654}{1.55300179}, \frac{2.41766651}{1.55300179}\right] = \boxed{[0.76573417, 1.55676994]}.$$

- **Sanity check** (plain softmax on all 4 keys) Weights from $s = [1, 3, 0, 2]$ are $\approx [0.08714, , 0.64391, , 0.03206, , 0.23688]$.

$$\sum_j w_j v_j = 0.08714[1, 0] + 0.64391[0, 2] + 0.03206[-1, 1] + 0.23688[3, 1]$$
$$= [0.76573417, 1.55676994],$$

  which matches the streaming result exactly.

Part IX

Parallelism

Chapter 13

Data Parallelism

13.1 Data Parallel

The first step of the typical training loop for deep learning models is to split a dataset into batches so that we can feed them into the model and compute gradients corresponding to them. As the model size grows up, we couldn't fit the model into a single GPU. The *data parallelism* tries to tackle the issue by clone the model across multiple GPUs so that each GPU can take a small portion of the batches for each iteration. Data Parallel (sometimes referred to as "single-node data parallel") is typically used when you have **multiple GPUs on a single machine**.

Let's say the batch size is 10 and we have 5 GPUs. Then, each GPU takes 2 batches and calculate gradients by on its own. The calculated gradients are then synchronized across the GPUs pretending they are computed on a single GPU. Finally, the synchronized gradient information is going to be distributed to them.

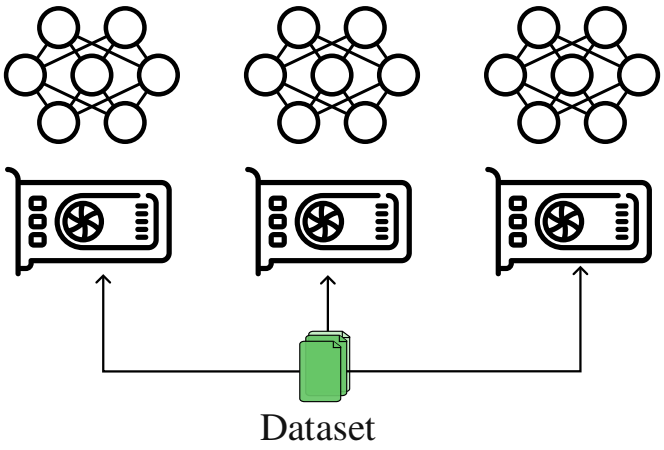
There are some important things to mention:

1. One process (or master thread) becomes a bottleneck for gradient aggregation and parameter updates.
2. As you increase the number of GPUs, or try to involve multiple machines, communication overhead grows significantly and can slow down training.
3. Each GPU holds a copy of the entire model, which can be large.

13.2 Distributed Data Parallel

To alleviate such issues, we can adopt an approach called *Distributed Data Parallel* (DDP), which is designed to scale training across many GPUs, potentially across multiple machines (nodes). Modern deep learning frameworks (like PyTorch torch.nn.parallel.DistributedDataParallel) typically recommend DDP as the best practice for multi-GPU/multi-node training due to better performance and scalability. During backpropagation, gradients are shared among GPUs through efficient communication primitives, resulting in synchronized model parameters across all GPUs.

Key benefits:



- Scalability: You can increase the number of GPUs (and even add more machines) to handle large datasets and bigger models.
- Performance: DDP typically provides better performance than older methods like NN.DATAPARALLEL (in PyTorch) because it uses *all-reduce* and eliminates the single "master" bottleneck.
- Flexibility: You can combine DDP with other parallelization strategies (*e.g.,* model parallel, sharded data parallel, pipeline parallel) if needed.

13.2.1 Concepts and Terminology

All-Reduce is a collective communication operation commonly used in distributed computing (especially in high-performance computing and deep learning). In simple terms:

- Each process (or GPU) starts with its own data (*e.g.,* local gradients).
- These data are combined (usually via a reduction operation like sum, mean, min, or max) across all processes.
- The result of that reduction (*e.g.,* the summed gradients) is then shared back so that every process receives the same reduced value.
- Hence the name: "all" (everyone gets the result) + "reduce" (combine data).

Basic Terms:

- World Size: The total number of processes engaged in the distributed job. Often, we run one process per GPU, so world size is the number of GPUs.
- Rank: A unique integer ID assigned to each process. Ranks typically range from 0 to world_size − 1. Rank 0 is often referred to as the "leader" or "master" process, but in DDP, every process does roughly the same work.
- Local Rank: When multiple GPUs reside on a single node, local rank identifies which GPU a specific process is mapped to on that local machine (*e.g.,* 0 for the first GPU, 1 for the second, etc.).

- Backend: The communication backend used for synchronization (*e.g.,* nccl). For GPU training, NCCL is typically recommended because it's optimized for high-performance GPU-to-GPU communication.
- Initialization Method: Describes how processes connect with each other (*e.g.,* a TCP store, a file-based store). This allows all processes to know who's who in the cluster.

13.2.2 How DDP Works Under the Hood

1. Process Per GPU: Each GPU runs the same script in its own process.
2. Data Subset: A DistributedSampler ensures that each process sees a unique subset of data. This prevents overlap in data usage among GPUs.
3. Full Model Copy: Each GPU has a full replica of the model in memory.
    - For massive models, consider *Sharded DDP* (*e.g.,* PyTorch's FSDP or DeepSpeed ZeRO) to split parameters across GPUs.
4. All-Reduce Gradient Sync: After backprop, gradients are summed (or averaged) across processes with an all-reduce operation. This keeps all models in sync.

13.3 DDP vs DataParallel

In PyTorch there are two common ways to use multiple GPUs:

- `nn.DataParallel`: "Single process, many GPUs, GPU0 is the boss."
- `DistributedDataParallel` **(DDP)**: "One process per GPU, talk via all-reduce (NCCL)."

In practice, `nn.DataParallel` is easy to use but inefficient and considered outdated for serious training. `DistributedDataParallel` is the recommended approach, especially for large models and LLMs.

13.3.1 Mental Model

`nn.DataParallel`

- There is **one Python process** and one "master" model copy on GPU0.
- For each forward pass:
    1. The input batch is split across GPUs.
    2. The model on GPU0 is *replicated* onto the other GPUs.
    3. Forward and backward are run on each GPU.
    4. Gradients are gathered back to GPU0, summed, and `optimizer.step()` is applied on GPU0's parameters.
- GPU0 becomes a bottleneck:

  - It hosts the master model.
  - It handles scatter/gather and the optimizer step.
- Everything happens inside a single Python process, so the Python GIL can also limit scaling.

`DistributedDataParallel` **(DDP)**

- There is **one process per GPU** (or per GPU per node).
- Each process:
  - Owns a model replica on its GPU.
  - Receives a shard of the global batch.
- After `loss.backward()`:
  - Gradients are **averaged across processes** via an all-reduce operation (typically NCCL).
  - Each process then calls `optimizer.step()` locally using the synchronized gradients.
- There is no single central bottleneck; synchronization is done in a distributed manner.

13.3.2 Practical Differences

Aspect	`nn.DataParallel`	`DistributedDataParallel`
Processes	1	1 per GPU
Model copies	Replicated each forward	One static replica per process
Gradient sync	Gather on GPU0	All-reduce (NCCL)
Python GIL bottleneck	Yes	No (multi-process)
Multi-node support	No	Yes
Performance / scaling	Poor for large models / many GPUs	Good, recommended
Recommended for training	No	Yes

For large-scale ML and LLM training/fine-tuning, `DistributedDataParallel` is the default choice.

Chapter 14

Pipeline Parallelism

14.1 Introduction

The basic idea of the data parallel is to distribute the model across GPUs. However, if the model size is bigger than the VRAM of GPU, the model wouldn't fit in a single GPU. To resolve the issue, we have to split the model across GPUs. For instance, we can put the half of the model into the fist GPU and the remaining half into the second GPU. This approach is often called *model parallelism*. Let's closely look at one of the model parallelism approaches, called *pipeline parallelism*.

Pipeline Parallelism is a strategy for distributing large deep learning models across multiple devices (GPUs) by splitting the model layers into sequential stages. Rather than replicating the entire model on each GPU or sharding the parameters themselves, pipeline parallelism assigns a subset of layers to each device in a pipeline-like fashion. This technique is especially helpful when:

- The model is too large to fit on a single GPU, but it can be split into chunks (layers/stages).
- You want to keep multiple GPUs actively working on different portions (stages) of the forward and backward pass concurrently.

14.1.1 Illustration of the Pipeline

In pipeline parallelism, the model is divided into N sub-networks, and each sub-network is placed on a different GPU (or sometimes on multiple GPUs if you have many layers). Think of it like an assembly line:

- Sub-Network 1: Layers $1 \sim k$
- Sub-Network 2: Layers $(k+1) \sim m$
- Sub-Network 3: Layers$(m+1) \sim \ldots$
- and so on.

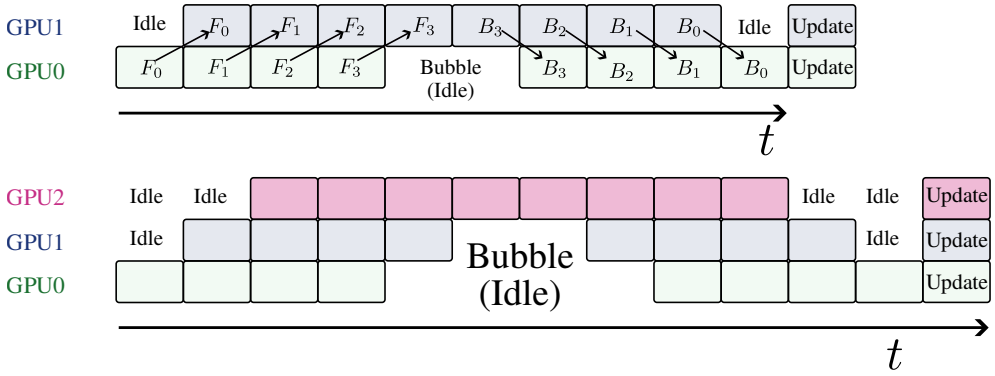


Figure 14.1: The Illustration of the pipeline parallel on two GPUs. As you can see the *bubble* (*i.e.,* underutilization) tends to grow as we increase the number of GPUs. It updates once at the end.

The input minibatch is then split into smaller micro-batches (smaller pieces of data), which flow sequentially through these sub-networks. In other words, the micro-batch is the basic unit of the input to the pipeline parallelism.

- While Stage 1 is processing the next micro-batch, Stage 2 can concurrently work on the intermediate outputs from Stage 1's previous micro-batch.

On each stage, for each microbatch that backpropagates through that stage's layers, the stage adds the microbatch gradients into its local grad buffers:

$$\text{grads\_stage} \mathrel{+}= \text{dLoss/dParam (microbatch k)}$$

Mixed precision usually applies loss scaling before accumulation.

Example: Imagine a 2-stage pipeline parallel setup (for simplicity):

- GPU 0: Holds Layers 1–3
- GPU 1: Holds Layers 4–6

If you have a batch of data with 32 samples, you might split it into 4 micro-batches of size 8 each. Then, forward Pass can be processed as follows:

1. Micro-Batch 1
   (a) Step A: GPU 0 processes layers 1–3 for micro-batch #1.
   (b) Step B: Once GPU 0 is done with those layers, it sends the activations for micro-batch #1 over to GPU 1.
   (c) Step C: GPU 1 then processes layers 4–6 for micro-batch #1.
2. Micro-Batch 2

   (a) As soon as GPU 0 finishes Step A for micro-batch #1 and passes the data to GPU 1, GPU 0 is free to start micro-batch #2 (layers 1–3).
   (b) Meanwhile, GPU 1 is busy processing micro-batch #1 (layers 4–6).
   (c) Once GPU 0 finishes its part for micro-batch #2, it sends those activations to GPU 1-which will be ready to handle them as soon as it's done with micro-batch #1.

3. Micro-Batch 3 and 4
   (a) This pattern continues in an overlapping fashion: while GPU 1 is busy with micro-batch #2, GPU 0 can start on micro-batch #3, and so on.

The key benefit is concurrency:

- While GPU 0 is processing micro-batch 2, GPU 1 can process micro-batch 1.
- This overlap leads to higher GPU utilization.

Backward pass is a bit more complex because:

- You need gradient signals to flow in the reverse order of the forward pipeline.
- Each stage waits until it receives the gradient from the next stage before it can compute its own local gradients and pass them back to the previous stage.

However, the overall concept is similar-multiple stages can run backprop (on different micro-batches) in parallel, thereby keeping all GPUs busy.

14.1.2 Pipeline Bubbles

When using pipeline parallelism, you often hear about *pipeline bubbles* (or underutilization). This refers to idle times on some GPUs before the assembly line is fully loaded or after it starts to wind down.

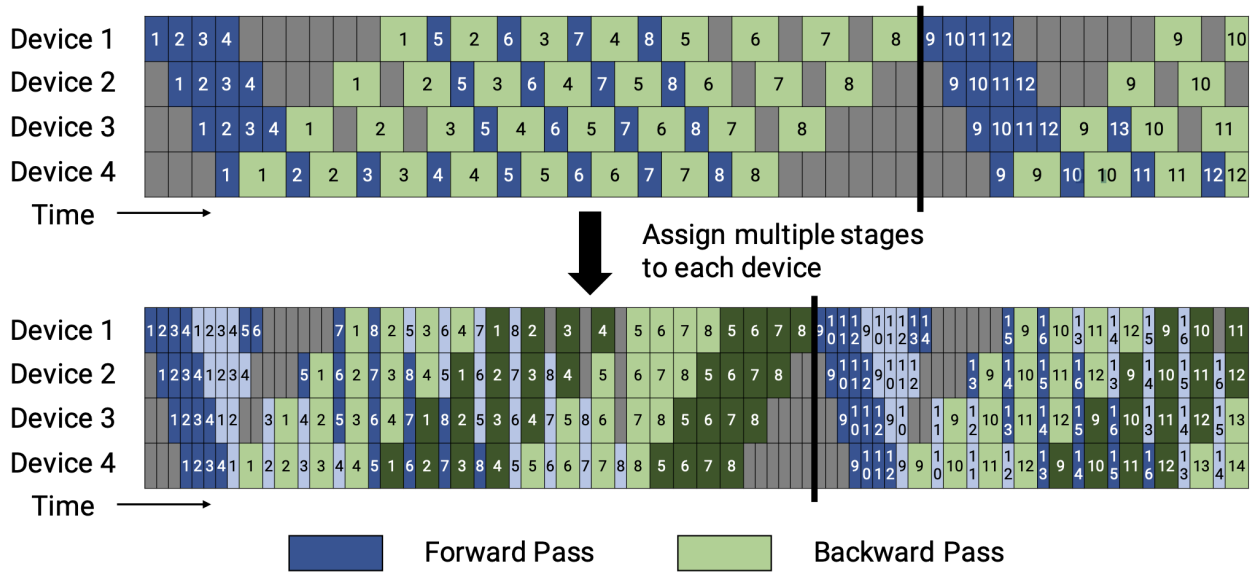
- Start-up Bubble: In the very beginning, GPU 1 must wait until GPU 0 finishes the first forward pass for micro-batch 1. GPU 1 sits idle during that initial delay.
- Wind-down Bubble: After the last micro-batch enters GPU 0, GPU 1 continues to process the pipeline while GPU 0 is idle.

The percentage of idle can be computed as follows:

$$\frac{1-m}{m+n-1},$$

where m is the number of microbatches and n is the number of GPUs.

These bubbles can lead to less-than-ideal speedups, but you can mitigate them by using enough micro-batches to keep the pipeline busy most of the time.



14.1.3 Combining Pipeline Parallelism with Other Forms of Parallelism

In practice, pipeline parallelism is often combined with:

- Data Parallelism: You still replicate each stage across multiple GPUs to handle separate shards of data.
- Tensor Parallelism / Model Parallelism: Instead of giving entire layers to one GPU, you split the parameters or compute of a single layer across multiple GPUs (common in large language model setups, *e.g.,* Megatron-LM).
- Sharded Optimizer Approaches (*e.g.,* ZeRO, FSDP): Distribute optimizer states and gradients to reduce memory overhead.

14.2 1F1B

One of the issues is that the model parameters keep changing while processing the forward passes. This means at every time step, minibatches are going to be forwarded through different weights. Thus, it is necessary to keep different states of the model parameters. Thus, 1F1B increases the memory requirements while increasing the processing speed.

14.2.1 Non-interleaved Schedule

The non-interleaved schedule can be divided into two states. The first state is the startup state (or warm-up state). In the startup state, After completing the forward pass for the first minibatch, it performs the backward pass for the same minibatch, and then starts alternating between performing forward and backward passes for subsequent minibatches. As the backward pass starts propagating to earlier stages in the pipeline, every stage starts alternating between forward and backward pass for different minibatches. As shown in the above figure, in the steady state, every machine is busy either doing the forward pass or backward pass for a minibatch.

14.2.2 Interleaved Schedule

This schedule requires the number of microbatches to be an integer multiple of the stage of pipeline. In this schedule, each device can perform computation for multiple subsets of layers(called a model chunk) instead of a single contiguous set of layers. *i.e.,* Before device 1 had layer 1-4; device 2 had layer 5-8; and so on. But now device 1 has layer 1,2,9,10; device 2 has layer 3,4,11,12; and so on. With this scheme, each device in the pipeline is assigned multiple pipeline stages and each pipeline stage has less computation. This mode is both memory-efficient and time-efficient.

14.3 Zero Bubble

Chapter 15

Tensor Parallelism

15.1 Introduction

Let's go over an example:

- x is a row vector of shape $[1, d_{\text{in}}]$ (the input).
- W is a weight matrix of shape $[d_{\text{in}}, d_{\text{out}}]$.
- output is $[1, d_{\text{out}}]$.

We have two GPUs, GPU 0 and GPU 1. We want to split (shard) the weight matrix W across two GPUs. One common approach is column parallelism:

- GPU 0 holds columns $[0, 1]$
- GPU 1 holds columns $[2, 3]$

This means each GPU stores some columns of W. Let's denote:

$$W = \left[W_{\text{left}} \mid W_{\text{right}}\right]$$

where

- W_{left} is a 4×2 matrix on GPU 0,
- W_{right} is a 4×2 matrix on GPU 1.

In numeric form, suppose

$$W = \begin{bmatrix} 1 & 2 & 5 & 6 \\ 3 & 4 & 7 & 8 \\ 2 & 0 & 3 & 1 \\ -1 & 4 & 8 & 2 \end{bmatrix}.$$

Then, for column parallel:

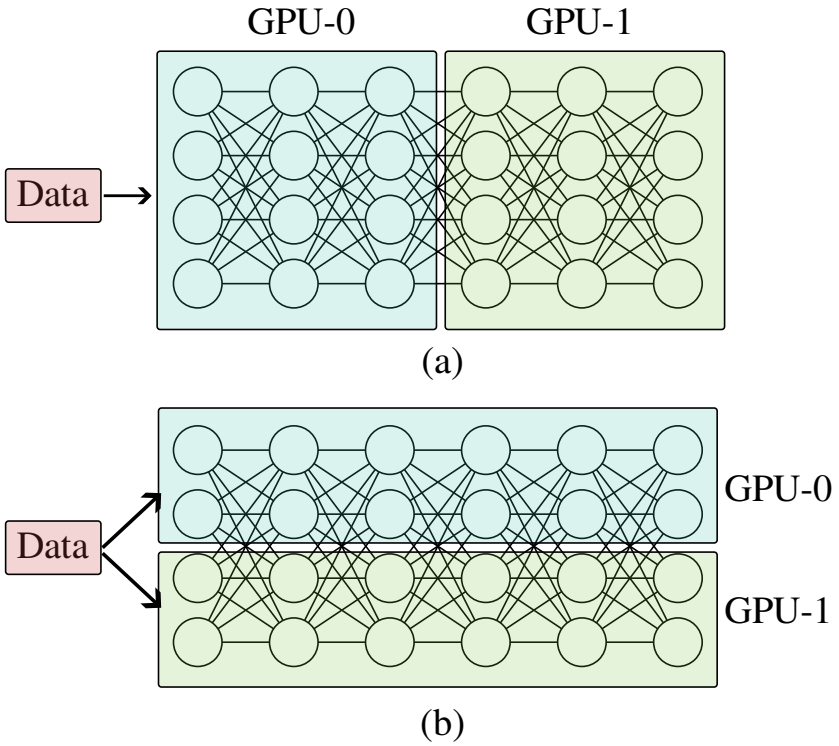


Figure 15.1: (a): Pipeline parallelism. (b) Tensor parallelism.

- GPU 0:

$$W_{\text{left}} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 2 & 0 \\ -1 & 4 \end{bmatrix}.$$

- GPU 1:

$$W_{\text{right}} = \begin{bmatrix} 5 & 6 \\ 7 & 8 \\ 3 & 1 \\ 8 & 2 \end{bmatrix}.$$

Given the input

$$x = [\,1,\ 2,\ 0,\ 1\,].$$

We can treat x as a row vector $[1, 4]$. For column parallelism, each GPU needs the entire input x so it can multiply by its subset of columns:

- We copy the x to both GPU 0 and GPU 1.
  - This is typically a small overhead compared to storing large weight matrices.
- Then, compute the matrix multiplications for each matrix.
- Finally, concatenate the outputs.

$$\text{output} = [\,\text{partial}_0 \,\big|\, \text{partial}_1\,] = [\,6,\ 14,\ 27,\ 24\,].$$

- Some frameworks do a ring-all-gather, or they might place this final output on one GPU if needed, etc.

When we do backprop, we can update the model's parameters in the opposite direction.

In Megatron-LM, all Transformer layers, except normalization layer, are using row or column parallelism.

Tensor parallelism can be costly primarily due to the significant communication overhead involved when distributing large model layers across multiple GPUs, requiring frequent data exchange between devices which can become a bottleneck, especially when dealing with very large models and limited network bandwidth; this communication cost often outweighs the benefits of parallel computation, making it a major drawback of tensor parallelism.

Chapter 16

N-Dim Parallelism

Chapter 17

DualPipe

17.1 Introduction

17.1.1 All-to-All vs Point-to-Point

When orchestrating multiple GPUs, we need them to communicate with each other to share information like gradients and model parameters. There are two main types of communication patterns:

1. All-to-all communication.
2. Point-to-point communication.

All-to-all communication involves every GPU in the system simultaneously exchanging data with all other GPUs. The canonical analogy is a group chat where everyone needs to share their updates with everyone else. All-to-all communication is expensive and involves a ton of communication overhead. There are several clever algorithms like ring-AllReduce that can reduce this overhead, but it's still often a bottleneck.

Point-to-point communication, on the other hand, is a communication between just two GPUs (the analogy here is a private conversation). One GPU sends data directly to another specific GPU without involving the rest of the system. This is much more efficient in terms of network bandwidth and latency. In practice, point-to-point communication is strongly preferred when possible because it's significantly cheaper in terms of computational resources.

17.2 DualPipe

DualPipe = bidirectional pipeline + full overlap of forward & backward + extra model copies.

Key pieces:

Bidirectional pipeline scheduling

- In classic pipeline, micro-batches only enter from one end.

Figure 17.1: An illustration of dualpipe.

- In DualPipe, micro-batches are injected from both ends of the pipeline.
- So each GPU can:
  - Run forward for some micro-batches coming from the left, and
  - Run backward for other micro-batches coming from the right at the same time (interleaved in time).

Each GPU is juggling two streams of work: forward for some micro-batches and backward for others, from opposite ends of the pipeline, while communications happen in the background. That's the dual pipeline.

Finer-Grained stages: divide each chunk into 4 components:

- Attention,
- All-to-all dispatch(Handles communication between devices),
- MLP(Multi-Layer Perceptron),
- All-to-all combine(merge output across devices).

For a backward chunk, the attention and MLP split further into two parts: backward for input(B) and backward for weights(W) like Zero Bubble.

Bidirectional pipeline scheduling which feeds micro-batches from both ends of the pipeline simultaneously and a significant portion of communications can be fully overlapped (See the 2 black arrows in following diagram). In order to support bidirectional pipeline scheduling, DualPipe requires keeping two copies of the model parameters. If we have 8 devices with a 8 layers model, in the Zero Bubble Schedule, each device has a corresponding layer. But in the DualPipe Schedule, in order to handle bidirectional pipeline, the device 0 should have model's layer0 and layer7, and the device 7 should have model's layer7 and layer0.

Part X

On-Device AI

Chapter 18

Introduction to On-Device AI

18.1 Introduction

Traditionally, AI models have relied on powerful cloud computing resources for training and inference [49]. However, with the proliferation of the IoT, edge computing, and mobile devices, an increasing number of AI models are being deployed on-device [42, 193]. This shift not only enhances the real-time processing and efficiency of data handling but also reduces reliance on network bandwidth and strengthens data privacy protection [60]. Specifically, Gartner projects that by 2025, approximately 75% of all enterprise-generated data will be produced outside traditional data centers [60]. Transmitting and processing this data in centralized cloud systems introduces significant system and latency overhead, along with substantial bandwidth requirements [43]. This also underscores the importance of deploying AI models on-device.

Edge intelligence enhances the concept of localized data processing by deploying AI algorithms directly on edge devices [1], thereby reducing reliance on cloud infrastructure [284]. This approach not only facilitates faster data processing but also addresses important privacy and security concerns, as sensitive data remains within the local environment [57, 122, 203], with on-device AI models finding application in various scenarios, such as smartphones, smart home systems, autonomous vehicles, and medical devices [43].

However, the effective implementation of AI models on edge devices poses significant challenges. The reliance of these models on large parameter counts and powerful processing capabilities necessitates the development of innovative strategies for

1. Model compression
2. Model optimization
3. Model adaptation to specific operational environments

On-device AI models refer to AI models that are designed, trained, and deployed on edge or terminal devices. These models can perform data processing and inference locally without the need to transmit data to the cloud for processing [44, 248]. On-device AI models typically possess the following characteristics:

[1] Edge devices encompass a wide range of hardware, from high-performance edge servers capable of executing complex computational tasks to resource-constrained IoT sensors designed for specific applications [190].

Table 18.1: Comparison of On-Device AI Models and Cloud-based AI Models

Aspect	**On-Device**	**Cloud-based**
Resources	Limited	Powerful
Latency	Low / Aim for real-time	high
Privacy	Enhanced	High security risks
Scalability	Limited	Good
Maintenance	Complex	Centralized

- **Real-time Performance**: They can quickly respond to user requests, making them suitable for applications that require immediate feedback [11].
- **Resource Constraints**: They are limited in computational power, storage, and energy consumption, necessitating optimization to fit the hardware environment of the device [20].
- **Data Privacy**: By processing data locally, they reduce the risks associated with data transmission, thereby enhancing user privacy protection [284].

Memory and speed are the keys for successful on-device AI models.

Research questions:

- What are the **applications of on-device AI models in daily life?**
- What are the main **technical challenges** for deploying on-device AI models?
- What are the most effective optimization and implementation methods for enhancing the performance of on-device AI models
- What are the future trends of on-device AI models?

18.2 Comparison of On-Device AI Models and Cloud-Based AI Models

The comparison between on-device AI models and cloud-based AI models highlights several critical aspects that influence their deployment and effectiveness (see Table 3). On-device AI models are constrained by the computational resources available on the device, necessitating optimization to function efficiently; however, they offer lower latency, making them suitable for real-time applications [261]. In contrast, cloud-based AI models leverage powerful cloud infrastructure, enabling the support of complex models but often resulting in higher latency, which can be detrimental in time-sensitive scenarios [248]. Data privacy is another significant consideration, as on-device models enhance privacy by processing data locally, thereby reducing the risks of data breaches, while cloud-based models face higher security risks due to the transmission of data to external servers [215]. Scalability also differs markedly; on- device models have limited scalability due to hardware constraints, whereas cloud-based models can dynamically adjust resources to accommodate varying demands [190]. Finally, maintenance presents a contrasting challenge: on-device models require complex updates and maintenance, particularly in large deployments, while cloud-based models benefit from centralized management, simplifying the update and maintenance processes [43].

18.2.1 Applications of On-Device AI Models

- Smartphones and Mobile Devices
  - Voice assistants
  - Image recognition
  - Personalized recommendations
  - Health monitoring
- IoT Devices
  - Smart Homes: bulbs, thermostats, security cameras and so on
  - Environmental Monitoring: temperature, humidity, air quality
  - Industrial automation: equipment failures, optimize production processes
  - Smart agriculture: AI models assist farmers in optimizing irrigation and fertilization practices to improve crop yields, promoting sustainable agricultural practices
- Edge computing
  - Real-time data processing: facial recognition, surveillance.
  - Traffic management
  - Smart manufacturing:
  - AR / VR
- Autonomous Driving and Intelligent Transportation Systems
  - Environmental perception: AI models analyze sensor data to identify surrounding objects
  - Path planning: analyzing traffic conditions and map data in real-time
  - Decision making: AI models assist autonomous systems in making quick decisions
  - Vehicle to everything: Through communication with other vehicles and infrastructure, AI models optimize traffic flow and enhance road safety, contributing to smarter transportation systems
- Medical Devices and Health Cares
  - Diseasee disagnosis
  - Personalized treatment
  - Remote monitoring
  - Drug development

18.3 Technical Challenges

18.3.1 Limited computational resources (CPU / GPU)

- Processing power: The limited performance of CPUs and GPUs in these devices may not suffice to meet the real-time processing demands of complex models. To address this challenge, optimizing algorithms to enhance computational efficiency is crucial. Techniques such as model pruning, quantization, and the use of specialized hardware accelerators can help improve processing capabilities without requiring significant increases in power consumption or hardware costs

- Model complexity: Complex models generally demand more computational resources, resulting in increased latency during execution on edge devices, which can adversely affect user experience.

18.3.2 Storage and Memory Limitations

- Storage Space: pruning and quantization
- Memory: Memory limitations on edge devices may hinder the ability to load all necessary data during inference, adversely affecting performance and response speed
- Data management:

18.3.3 Energy Consumption

- Battery: Many edge devices rely on battery power, and the high energy demands of AI models can lead to rapid battery depletion, adversely affecting user experience.
- Dynamic Energy Management: To balance performance and energy use, devices must dynamically adjust their energy consumption strategies in response to varying workloads and environmental conditions [142, 240]. Researchers are focusing on developing intelligent energy management algorithms, including AI-based controllers, to optimize energy usage in edge devices [195] [205].
- Hardware Optimization: Designing dedicated hardware accelerators, such as Tensor Processing Units (TPUs) and Field Programmable Gate Arrays (FPGAs), can significantly enhance the computational efficiency of AI models while reducing energy consumption

18.3.4 Communication Bandwidth Limitations

Edge devices typically face significant communication bandwidth limitations compared to servers, making it challenging to transfer large volumes of data between the edge and the cloud.

- Data Preprocessing: One effective approach to reducing data transmission is through data preprocessing algorithms. These algorithms can filter and compress data, ensuring that only relevant information is transmitted during communication
- Edge Caching: Edge caching technology is another valuable strategy that allows for the storage of frequently accessed data and models directly on edge devices [246]. By reducing the frequency of communication with the cloud, edge caching minimizes the amount of data transmitted [74].
- On-Device Computation: performing computations directly on the edge device, only relevant data needs to be transmitted to the cloud

18.3.5 Data Privacy and Security

- Data Protection: AI models deployed on edge devices frequently handle personal data, including health information and location data, making the security of this information during processing and storage paramount to preventing data breaches

  - Homomorphic encryption enables computations to be performed on encrypted data without the need for decryption, thereby preserving confidentiality during processing

- Compliance: AI models on edge devices must comply with relevant laws to ensure the lawful use and protection of user data
- Security Attacks: Edge devices are susceptible to a range of security threats, including malware and network attacks, prompting researchers to actively develop security mechanisms to safeguard these devices and the sensitive data they process

18.3.6 Model Transferability and Adaptability

The transferability and adaptability of AI models on edge devices are crucial for ensuring effective operation across diverse environments:

- Cross-Device Migration: AI models must be capable of running on various types of devices, including migrating from high-performance servers to resource-constrained mobile devices.
- Environmental Adaptability: Edge devices operate in diverse environmental conditions, including variations in lighting, temperature, and network connectivity
- Continuous Learning: AI models on edge devices must possess the capability for continuous learning and updating during use to adapt to changing user needs and behavior patterns

18.4 Optimization and implementation of AI models on devices

18.4.1 Data Optimization

In ML, the principle of "garbage in, garbage out" underscores the importance of high-quality data inputs for achieving reliable results

- Data Filtering: Data filtering is essential for maintaining data quality by eliminating irrelevant or noisy data prior to further analysis
  - robust filtering techniques
- Active label cleaning
- Ensemble methods also play a significant role in effectively managing varying noise levels across datasets
- Feature Extraction
- Data Aggregation: Data aggregation involves synthesizing information from multiple sources to minimize redundancy and enhance coherence, which is particularly beneficial in IoT networks with interconnected devices. Techniques like federated learning enable data privacy while facilitating the combination of data from distributed sources, thus providing efficient solutions for processing large datasets. However, while aggregation can improve data coherence, it may also introduce latency issues if the methods employed are overly complex or centralized

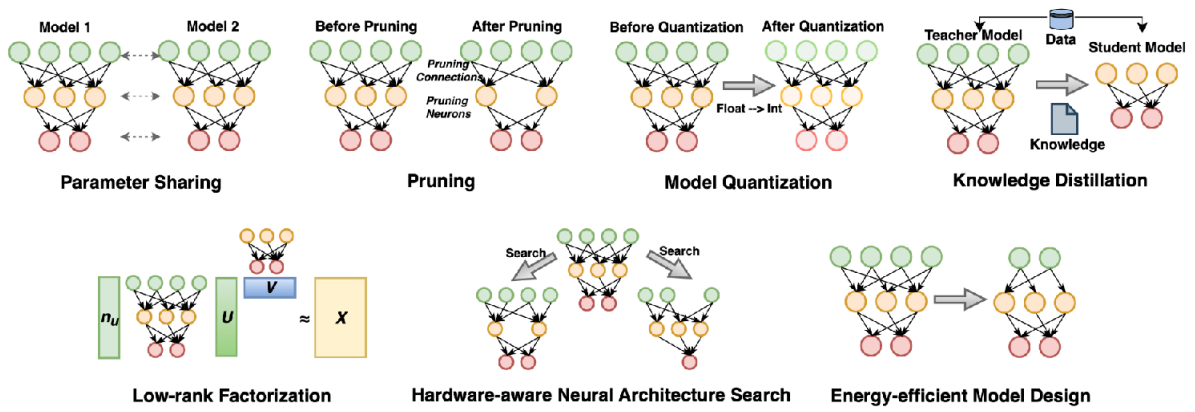


Figure 18.1: An overview of model optimization techniques.

- Data Quantization: Data quantization refers to the process of reducing the precision of data representation, commonly applied in scenarios that require efficient processing of sensor data on edge devices

18.4.2 Model Optimization

To effectively deploy AI models on edge devices, a variety of model optimization techniques have been developed [248]. These approaches aim to reduce the size and complexity of AI models while preserving their performance levels. Key techniques include parameter sharing, pruning, model quantization, knowledge distillation, low-rank factorization, hardware-aware neural architecture search, and energy-efficient model design [20]. A notable example of integrating multiple optimization strategies is Deep Compression, which synergistically combines pruning, quantization, and Huffman coding to achieve substantial reductions in the size of deep neural networks (DNNs) [71].

Table 18.2: Model Optimization Techniques in On-Device AI

Technique	Desc	Pros
Parameter Sharing	Sharing parameters	Decrease memory usage and improve inference
Pruning	Eliminate insignificant weights	
Model Quantization	Lowers the precision of weights	
Knowledge Distillation	Train a smaller student model	
Low-Rank Factorization	Decomposes weight matrices	
Neural Architecture Search	Search optimized architecture	
Energy-Efficient Model	Design efficient architecture	

- Traditional ML (before DL) Compression Methods:
- Parameter Sharing: Using the same learnable weights in multiple places of a model so one set of parameters serves many computations.
- Pruning: By systematically removing redundant parameters or entire layers, pruning techniques decrease model complexity, enabling faster inference and lower memory consumption while often maintaining competitive accuracy [30].

- Model Quantization: By decreasing the precision of model parameters and activations, quantization achieves substantial reductions in model size while minimizing accuracy degradation
- Knowledge Distillation
- Low-rank factorization
- Hardware-aware Neural Architecture Search
- Energy-efficient Model Design

18.5 System Optimization Techniques

As the demand for real-time performance and resource-efficient deep learning models continues to rise, optimizing systems for on-device AI deployment has become a critical area of research. Successfully deploying deep learning models on edge devices necessitates a combination of software and hardware-based approaches to enhance computational efficiency.

Part XI

Compression

Chapter 19

Model Compression

Chapter 20

Model Pruning

Chapter 21

Quantization