Tutorial: Becoming Full-Stack AI Researchers

Working Draft

Authors[1]

[1]Yale University

Version: `December 31, 2025`

1 Introduction

Large language models (LLMs) and their surrounding ecosystem have evolved rapidly, creating a landscape where the best researchers are expected to navigate multiple layers of the AI stack. From downloading sharded model checkpoints to deploying distributed reinforcement learning pipelines, a modern AI researcher must operate as both an engineer and a scientist.

Problem The traditional division of labor, where some focus only on mathematical modeling and others only on systems, has given way to a full-stack expectation: the ability to move fluidly between data handling, model training, evaluation, deployment, and interpretability. However, we cannot find resources that can introduce AI researchers to understand the entirety of the technical stack.

Contribution In the Fall 2025 semester at Yale University, we organised a **Becoming Full-Stack AI Researchers** working group. This tutorial paper and its accompanying GitHub repository emerged out of the working group. Our goal is not just to introduce individual frameworks in isolation, but to show how they fit together into an end-to-end research workflow. We emphasize hands-on, reproducible, and extensible practices. Each section is written by contributors who prepared minimal working examples (MWEs), demos, slides, and live presentations, thereby capturing both the technical details and the practical challenges encountered by new users.

Application This tutorial can be used as the starting point of a one-semester course on modern AI development as well as for self-study of interested learners who have basic familiarity with Python. Our intended audience includes graduate students, postdoctoral researchers, and faculties who want to broaden their AI engineering capabilities. By the end of this tutorial, readers should be able to replicate common workflows, adapt them to their own research problems, and understand the broader design space of tools and frameworks that constitute modern AI development.

Organization This tutorial is organized into modules covering the full AI development pipeline:

- **Foundations**: PyTorch, JAX, and TensorFlow fundamentals
- **LLMs as Black Boxes**: Inference, evaluation, deployment, and serving

- **Post-Training**: Supervised fine-tuning (SFT) and reinforcement learning (RL)
- **Agentic Systems**: Software and hardware agents, multi-step reasoning
- **Systems**: Distributed training (Ray), efficient serving (vLLM), memory optimization (DeepSpeed)

Each module includes minimal working examples (MWEs), detailed explanations, and practical exercises.

Acknowledgements This course is generously supported by the Wu Tsai Institute at Yale University. We thank Ping Luo, John Lafferty, Linjun Zhang, Anurag Kashyap, Theo Saarinen, and Yuxuan Zhu for helpful comments and suggestions.

2 PyTorch, JAX, and TensorFlow

2.1 Motivation and Background

Modern deep learning frameworks like TensorFlow, PyTorch, and JAX were created with the goals of

1. Efficient tensor (matrix) operations at scale, and
2. Automatic differentiation for optimization algorithms.

Consider a loss function $L(\theta)$ depending on model parameters $\theta \in \mathbb{R}^d$. Training typically happens via the gradient descent algorithm

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta L(\theta_t),$$

where η is the learning rate. Computing $\nabla_\theta L(\theta)$ by hand is infeasible for complex neural networks. These frameworks automate it by building a *computational graph* and applying reverse-mode differentiation (backpropagation).

They are written with GPU and TPU in mind for efficient parallel computation so that linera algebra operations and auto diff are executed faster than can be done on CPUs with numpy.

2.2 Historical Context and Ecosystem

In 2015, Google introduced its first large-scale public framework, TensorFlow. In 2018, Google introduced JAX, a functional, composable approach with transformations like `jit`, `grad`, and `pmap`, optimised for TPUs. TensorFlow was once popular but is now on its way to depreciation.

Meanwhile, in 2016, Facebook introduced PyTorch, which became the default for most academic work. Rumours have it that nowadays, Google and xAI work mostly with JAX, OpenAI and Meta/Facebook work mostly with PyTorch, and Anthropic uses both.

2.3 Strengths and Hardware Compatibility

PyTorch. Strengths: Vast ecosystem (Hugging Face, PyTorch Lightning) and greater community support. GPU-first.

JAX. Strengths: clean functional abstractions, XLA compiler, flexibility in creating customised kernels. TPU-first.

2.4 Core Programming Model and Syntax

Here is a simple example written in PyTorch and in Jax. With PyTorch, we can do

```
import torch

x = torch.tensor([2.0], requires_grad=True)
y = x**2 + 3*x + 1
y.backward()   # compute dy/dx
print(x.grad)  # prints tensor([7.])

```

The gradient follows

$$\begin{aligned} y &= x^2 + 3x + 1, \\ \frac{dy}{dx} &= 2x + 3 = 7 \quad \text{at } x = 2. \end{aligned}$$

Equivalently, in JAX, we can do

```
import jax
import jax.numpy as jnp

def f(x):
  return x**2 + 3*x + 1

grad_f = jax.grad(f)
print(grad_f(2.0))  # prints 7.0

```

2.5 Ecosystem and Tooling

Each framework is surrounded by a rich ecosystem of higher-level libraries.

- **PyTorch:** PyTorch Lightning provides structured training loops and experiment organization. The Hugging Face Transformers library, while framework-agnostic, has its deepest integration with PyTorch.

- **JAX:** Flax and Haiku are neural network libraries built specifically for JAX, offering functional-style model definitions.

For distributed and large-scale training, each framework offers its own primitives.

- **PyTorch:** the `torch.distributed` package implements data-parallel and model-parallel training across multiple GPUs and nodes.
- **JAX:** the `pmap` transformation allows parallel execution of functions across multiple devices (TPUs or GPUs), with automatic sharding of arrays.

For the compatibility with the computational hardware,

- **PyTorch:** Built directly on top of CUDA/cuDNN for GPU acceleration.
- **JAX:** Relies on XLA, which in turn generates CUDA kernels when targeting NVIDIA GPUs. Compatible with both GPUs and TUPs.

Finally, orchestration frameworks such as **Ray** integrate with all three, providing a higher-level interface for scaling training jobs beyond a single machine.

2.6 Installation and Getting Started

Installation depends on hardware

- PyTorch: `pip install torch torchvision torchaudio`. Select CUDA version if using GPUs.
- JAX: `pip install jax jaxlib`, or TPU runtime via Colab/Cloud.

Typical pitfalls include

- CUDA version mismatches between PyTorch and system drivers.
- Installing JAX with the wrong CUDA wheel.
- Mixing conda and pip environments without care.

3 Scaling Laws for Neural Language Models

3.1 Motivation: Predictable Improvement

One of the most remarkable discoveries in modern AI is that model performance scales predictably with key factors: model size, dataset size, and compute budget. Understanding these **scaling laws** enables:

- **Resource planning:** Predict performance before training expensive models
- **Optimal allocation:** Balance parameters, data, and compute efficiently
- **Research prioritization:** Identify bottlenecks and high-leverage improvements
- **Capability forecasting:** Anticipate future model capabilities

The power of scaling laws lies in their predictability across orders of magnitude, allowing researchers to extrapolate from smaller experiments to larger deployments.

3.2 Power Law Foundations

Scaling laws typically follow **power law** relationships of the form:

$$L = \frac{\alpha_0}{p^{\alpha}} \tag{1}$$

where:

- L is the loss (or error metric)
- p is a scaling parameter (e.g., number of parameters, dataset size)
- α_0 is a constant
- α is the scaling exponent

3.2.1 Why Power Laws?

Power laws arise from **scale invariance**: scaling the input p by a factor k produces a proportional response $f(kp) = k^{\beta} f(p)$. This property necessarily leads to relationships of the form $L = cp^{\alpha}$.

3.2.2 Log-Log Linearity

Taking logarithms of Equation **??**:

$$\log L = \log \alpha_0 - \alpha \log p \tag{2}$$

This linearity in log-log space is the signature of power laws and enables easy fitting and extrapolation.

3.3 Kaplan Scaling Laws (2020)

Kaplan et al. [**kaplan2020**] conducted extensive experiments to characterize how language model loss scales with:

- N = Number of model parameters
- D = Dataset size (number of tokens)
- C = Compute budget (FLOPs)

3.3.1 Key Findings

Loss vs. Model Size.

$$L(N) \propto N^{-\alpha_N} \quad \text{where } \alpha_N \approx 0.076 \tag{3}$$

Larger models achieve lower loss, with diminishing returns following a power law.

Loss vs. Dataset Size.

$$L(D) \propto D^{-\alpha_D} \quad \text{where } \alpha_D \approx 0.095 \tag{4}$$

More training data improves performance, but with diminishing returns similar to model size.

Loss vs. Compute.

$$L(C) \propto C^{-\alpha_C} \quad \text{where } \alpha_C \approx 0.050 \tag{5}$$

Given fixed compute, how should one allocate between model size and training tokens?

Optimal Allocation (Kaplan). Kaplan et al. recommended:

- Most compute should go toward larger models
- Dataset size can remain relatively modest
- This led to models like GPT-3 (175B parameters, 300B tokens)

3.4 Chinchilla Scaling Laws (2022)

Hoffmann et al. [**chinchilla2022**] revisited Kaplan's experiments and found different optimal allocations.

3.4.1 Key Revision

Main Finding: For compute-optimal training, model size (N) and dataset size (D) should scale equally with compute budget (C):

$$N \propto C^{0.50}, \quad D \propto C^{0.50} \tag{6}$$

3.4.2 Implications

Previous Models Were Over-Parameterized. Models like GPT-3 used too many parameters for their training data:

- GPT-3: 175B parameters, 300B tokens
- Chinchilla: 70B parameters, 1.4T tokens
- Result: Chinchilla outperformed GPT-3 despite fewer parameters

Modern Approach. Post-Chinchilla models follow this guidance:

- LLaMA-65B: 65B parameters, 1.4T tokens
- Mistral-7B: 7B parameters, scaled appropriately
- Emphasis on high-quality training data

3.4.3 Three Scaling Regimes

Understanding when different factors dominate:

Small-Scale Regime. Model capacity is the bottleneck:

- Increasing N yields largest improvement
- Limited by model expressiveness
- Example: Sub-billion parameter models

Medium-Scale Regime. Dataset size becomes limiting:

- Need more diverse training data
- Diminishing returns from just scaling N
- Example: 1B–100B parameter range

Large-Scale Regime. Compute efficiency matters most:

- Both N and D must scale together
- Infrastructure and optimization critical
- Example: 100B+ parameter models

3.5 Practical Applications

3.5.1 Model Development Planning

Given compute budget C, determine optimal:

$$N^* = k_1 \cdot C^{0.50} \tag{7}$$

$$D^* = k_2 \cdot C^{0.50} \tag{8}$$

where k_1, k_2 are task-dependent constants.

3.5.2 Performance Prediction

Estimate final loss before training:

$$L_{\text{pred}} = \frac{\alpha_0}{N^{\alpha_N} \cdot D^{\alpha_D}} \tag{9}$$

This enables cost-benefit analysis for model development.

3.5.3 Bottleneck Identification

If current performance is:

- **Far from predicted:** Optimization or data quality issues
- **Close to predicted but insufficient:** Need more N, D, or C
- **Better than predicted:** Architecture or training innovations

3.6 Data Availability Constraints

Scaling laws assume unlimited high-quality data, but reality imposes limits:

3.6.1 Internet-Scale Data

Estimates suggest:

- 10-100 trillion tokens of web text available
- Quality varies dramatically
- Deduplification reduces effective size
- Multilingual data has different coverage

3.6.2 Synthetic Data

When natural data exhausted:

- **Distillation:** Learn from stronger models
- **Self-play:** Generate and curate own data
- **Targeted generation:** Focus on capability gaps
- **Quality over quantity:** Careful curation and filtering

3.7 Downstream Task Performance

Pretraining scaling predicts downstream capabilities:

3.7.1 Emergent Abilities

Some capabilities appear suddenly at scale:

- Multi-step reasoning
- In-context learning
- Code generation
- Multilingual transfer

Debate: Whether these are truly emergent or merely crossing evaluation thresholds.

3.7.2 Fine-Tuning Efficiency

Larger pretrained models typically:

- Require less fine-tuning data
- Converge faster in fine-tuning
- Transfer better across domains
- Benefit more from parameter-efficient methods (LoRA)

3.8 Inference Scaling

Beyond training, inference costs matter:

3.8.1 Performance vs. Inference Cost

Tradeoffs:

- **Larger models:** Better quality, higher latency/cost
- **Smaller models:** Worse quality, lower latency/cost
- **Quantization:** Reduces inference cost at slight quality loss

3.8.2 Deployment Considerations

For production:

$$\text{Total Cost} = \text{Training Cost} + (\text{Inference Cost} \times \text{Number of Requests}) \quad (10)$$

Since models serve billions of requests, inference costs often dominate. This motivates:

- Smaller, cheaper models when sufficient
- Knowledge distillation from large to small models
- Efficient architectures (Mixture-of-Experts)
- Quantization and pruning

3.9 Limitations and Future Directions

3.9.1 Limitations of Current Scaling Laws

- **Architecture dependence:** Scaling exponents may differ for different architectures
- **Task specificity:** Downstream tasks may not scale identically to pretraining loss
- **Data quality:** Scaling laws assume i.i.d. data; quality matters
- **Multimodality:** Vision-language models may follow different laws

3.9.2 Open Questions

- How do Mixture-of-Experts models scale?
- Do different modalities (vision, audio) follow similar laws?
- How does post-training (RL, fine-tuning) scale?
- Are there fundamental limits to scaling?
- How do architectural innovations affect scaling?

3.10 Summary

Key takeaways:

- Model performance scales predictably with size, data, and compute
- Power laws enable performance prediction across orders of magnitude
- Chinchilla scaling: balance parameters and data equally
- Practical implications for model development and deployment
- Consider both training and inference costs
- Data availability and quality impose real constraints

Understanding scaling laws is essential for:

- **Research:** Prioritize high-leverage improvements
- **Development:** Allocate resources optimally
- **Deployment:** Balance quality and cost
- **Strategy:** Anticipate future capabilities

As the field matures, scaling laws will continue evolving, but their fundamental insight—that AI progress is predictable and scalable—remains transformative.

4 Ray: A Distributed Computing Framework

4.1 Motivation and Background

As compute and memory requirements of LLMs keep increasing, single-server settings can no longer accommodate their development, training, and serving needs. Ray [**raypaper**] addresses this challenge by enabling fast and easy application scaling to distributed environments. It provides scalable libraries to support the complete life-cycle of ML applications, pythonic primitives for intuitive development, and integration with existing tools and infrastructure like Kubernetes, AWS, and Azure.

At its core, Ray utilizes three main abstractions:

- **Tasks (remote functions):** A Python function decorated with `@ray.remote`. Each invocation launches a stateless parallel computation executed on any available CPU or GPU.
- **Actors:** A Python class decorated with `@ray.remote` becomes an actor. Unlike tasks, actors are *stateful*—they live on a worker process and can preserve variables across multiple method calls.
- **Worker Node:** Under the hood, Ray maintains a pool of worker nodes, which consist of multiple physical processes. Each node hosts a *raylet* process, which manages the node's shared resources, and one or more worker processes responsible for task submission and execution.

Ray's cluster hosts the head node, which contains the *Global Control Store*(GCS), a server managing cluster-level metadata and operations like scheduling. For fault tolerance, Ray introduced an ownership model, which can be combined with lineage reconstruction to recover lost objects in the event of a node's failure. These components contribute to Ray's core principles for API simplicity and generality, while enabling system performance and reliability. More information about Ray's architecture can be found in the designated architecture whitepaper [**rayarchitecture**].

4.2 Historical Context and Ecosystem

Researchers at UC Berkeley's RISELab introduced Ray at 2018 [**raypaper**] and it has since evolved into a widely adopted ecosystem for scaling AI and Python workloads. Today, Ray is maintained by Anyscale and supports a range of libraries for specialized tasks:

- **Ray RLlib**: High-Performance RL
- **Ray Serve**: Scalable & Programmable Model Serving.
- **Ray Tune**: Scalable Hyperparameter Tuning.
- **Ray Train**: Distributed Model Training
- **Ray Data**: Distributed Data Processing

4.3 Core Programming Model and Syntax

The minimal Ray program utilizes standard Python. To enable Ray, we import and initialize it:

```
import ray
ray.init()  # connect to cluster or start locally
```

Remote functions (tasks). Decorate a Python function with `@ray.remote` to execute it as a parallel task:

```
@ray.remote
def square(x):
  return x * x

futures = [square.remote(i) for i in range(4)]
print(ray.get(futures))  # [0, 1, 4, 9]
```

Actors. Use actors to encapsulate state across multiple method calls. An actor is a class decorated with `@ray.remote` and runs persistently on a worker process:

```
@ray.remote
class Counter:
  def __init__(self):
    self.value = 0
  def increment(self):
    self.value += 1
```

```
        return self.value

counter = Counter.remote()
print(ray.get(counter.increment.remote()))  # 1
```

Workers. Behind the scenes, Ray launches a pool of worker processes (Python interpreters). Tasks and actors are scheduled onto these workers by the Ray runtime. While you rarely interact with workers directly, you can control resource allocation (e.g., CPUs, GPUs) when defining tasks or actors:

```
@ray.remote(num_cpus=2)
def heavy_task(x):
  return x ** 2
```

Here, Ray will schedule `heavy_task` only on workers with at least two available CPUs.

4.4 Strengths, Weaknesses, and Integration

Strengths. The Python-first design of Ray lowers the barrier for distributed computing. Its modular libraries (Tune, RLlib, Serve) cover end-to-end machine learning workflows. It integrates naturally with PyTorch, TensorFlow, and JAX.

Weaknesses. Ray introduces runtime overhead for small-scale tasks, and cluster management may require careful tuning for performance. It is best suited for medium to large-scale AI applications in distributed environments rather than fine-grained parallelism in single-server settings.

4.5 Installation and Getting Started

Ray can be installed with:

```
pip install ray
```

Launching Ray locally requires no configuration. To connect to a multi-node cluster, one can start the head node with `ray start --head` and join worker nodes with `ray start --address='...'`.

Ray can also be combined with orchestration systems such as Kubernetes or Slurm for deployment on HPC or cloud environments.

5 vLLM

```
|-------------------------------------------------------------------|
|                  OUTER LOOP: RL iterations / epochs               |
|                                                                   |
|   |---------------|      |--------------------|      |---------|   |
|   |      vLLM     | ---> |  Compute Rewards   | ---> |DeepSpeed|   |
|   | Generate      |      | (human/RM via      |      | Train   |   |
|   | Trajectories  |      |  vLLM if needed)   |      | Policy  |   |
|   |---------------|      |--------------------|      |---------|   |
|          |                         |                      |        |
|          |------------------------------------------------|        |
|                      (Updated Policy fed back)                    |
|-------------------------------------------------------------------|
```

5.1 Motivation

Imagine a reinforcement learning (RL) training loop: we **generate trajectories**, compute rewards, and **update the policy** model.
vLLM and **DeepSpeed** act as *system-level optimizers* for these two core stages.
vLLM accelerates **trajectory generation** and **reward inference**, making large-scale sampling highly efficient through optimized GPU memory management (PagedAttention).
DeepSpeed accelerates **policy training**, enabling distributed, memory-efficient optimization (via ZeRO) for massive models.
Together, they form the backbone of a scalable RL pipeline—vLLM for the inference-heavy side, DeepSpeed for the training-heavy side—both maximizing GPU utilization across the loop.

5.2 Section 1—The KV Cache and Why It Matters

5.2.1 Attention Mechanism Formula

The general form of the **scaled dot-product attention** is:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V,$$

where we have

- Q = Query matrix
- K = Key matrix
- V = Value matrix
- d_k = Dimension of the key vectors (used for scaling)

At each decoding step, the model must compute attention between the *current* token(s) and *all past key–value pairs* to determine what context is most relevant.

5.2.2 2. Computational Cost Without Caching

During **autoregressive generation**, each new token requires recomputing the attention using *all* previous tokens:

$$\text{Attention}_t = \text{softmax}\left(\frac{Q_t K_{1:t}^T}{\sqrt{d_k}}\right) V_{1:t}.$$

That means for every new token t, you rebuild $K_{1:t}$ and $V_{1:t}$. This leads to **quadratic time complexity** $O(T^2)$ for a sequence of length T. This becomes very expensive for long sequences.

```
             Generating the 4th token

  |------------------------|
  | tok 1 | tok 2 | tok 3 |  <-- already processed
  |------------------------|
              ^
              |  attend to all
            [ tok 4 ]
```

5.2.3 3. KV Caching: The Core Idea

KV caching avoids recomputing K and V for the prefix every step.

- On the first pass, compute K and V for the prompt/prefix and **store** them.
- On each new token, compute only the new Q, and **reuse** cached K, V.

This keeps the cost *per new token* closer to $O(T)$ instead of $O(T^2)$.

5.2.4 4. Why KV Cache Is Not Trivial in Serving

Real LLM serving is not a single long sequence:

- Many requests arrive at different times.
- Each request has a different length.
- Some requests finish, some keep generating.

So we must keep many KV caches in GPU memory, all growing at different speeds.

5.2.5 5. Baseline vs With KV Cache

Compute Keys/Values	Recomputed each step	Reused from cache
Complexity	$O(T^2)$	$O(T)$
Speed	Slow for long sequences	Much faster

5.2.6 6. Throughput via Batching, and Its Two Problems

To use the GPU efficiently, we **batch** requests. But in practice:

1. **Asynchronous arrivals:** requests don't start together.
2. **Variable lengths:** prompts/outputs differ; some finish early.

This leads to idle threads and poor packing unless we **rebatch every decoding iteration** (a.k.a. *iteration-level scheduling / cellular batching*).

5.2.7 7. KV Cache Still Has Issues

Even with better batching, many concurrent, variable-length requests cause **GPU memory fragmentation**—turning into a GPU memory management problem that Section 2 addresses.

5.3 Section 2—PagedAttention: Solving the KV Cache Memory Problem

5.3.1 1. Motivation: KV Cache Fixes Computation, but Creates a Memory Challenge

KV caching saves compute but each request's cache **grows autoregressively** as new tokens are generated. With many concurrent requests, caches grow and finish at different times, stressing GPU memory.

5.3.2 2. The Naïve Way: Consecutive (Contiguous) Allocation

```
|----------------------------------------------|
| [Req A (50 tok)] [Req B (30 tok)] [Req C (80)]|
|----------------------------------------------|
```

If B finishes early, we get a hole:

```
|-------------------------------------------------------------|
| [Req A (50 tok)] [cleared (30 tok)] [Req C (80)] [Req D (60)]|
|-------------------------------------------------------------|
```

Now A or C may want to grow, but the memory right next to them is not free.
This leads to:

- fragmentation,
- expensive copies/reallocations,
- or rejecting new/longer requests.

5.3.3 3. The "Cut It Up" Idea—Why It's Hard

Splitting a request's KV into smaller chunks would pack memory better, *but* then attention has to reconstruct the correct order—hard to track without structure.

```
|----------------------------------------------|
| [Req A (50 tok)] [Req B (30 tok)] [Req C (80)]|
|----------------------------------------------|

|-------------------------------------------------------------------------|
| [Req A (50 tok)] [Req D part 1 (30 tok)] [Req C (80)] [Req D part 2 (30)]|
|-------------------------------------------------------------------------|
```

5.3.4 4. The Breakthrough: PagedAttention

PagedAttention slices GPU memory into **uniform pages** (small fixed-size blocks, e.g. a fixed number of tokens per page). Each request's KV cache is a *list of pages*, not one big block.

```
GPU pages:  P1 | P2 | P3 | P4 | P5 | P6 | ...
Req A  → P1,P2
Req B  → P3
Req C  → P4,P5
A grows → take P6 (no moves, just add a page)
```

5.3.5 5. The Page Table (Indirection)

A lightweight **page table** maps *logical token indices → (page, offset)*:

```
token 37 → page_table[37//page_size] + (37 % page_size)
```

Attention kernels read K/V via this mapping, so KV can be physically scattered yet **logically contiguous**.

5.3.6 6. Benefits

- Avoids large reallocations when a request grows.
- Reuses freed pages from finished requests.
- Greatly reduces fragmentation.
- Enables many concurrent, variable-length sequences.

5.3.7 7. Interaction with Scheduling

Because vLLM also does per-iteration / cellular scheduling, PagedAttention makes it feasible to mix:

- long-running chats,
- short bursts,
- reward-model-style inference

on the same GPU without running into "I can't fit that request" issues.

5.4 Section 3—Inside vLLM: How Paging and Scheduling Work Together

5.4.1 1. Goal

Keep the GPU highly utilized despite **asynchronous arrivals** and **variable-length** requests.

5.4.2 2. Runtime Pipeline

```
User/API → Request Queue → Scheduler (per-iteration) → GPU Workers → PagedAttention →
```

- **Request Queue:** holds incoming work
- **Scheduler:** mixes new and ongoing requests instead of waiting for a full batch; also removes finished sequences (those that reached EOS or max length)
- **GPU Workers:** run attention kernels; read K/V via page table; write new K/V into free pages
- **PagedAttention:** provides flexible KV memory without copies
- **Streaming:** return tokens as they are ready while continuing decoding

5.4.3 3. Why This Works

- PagedAttention allows **dynamic batching** without moving memory.
- The scheduler keeps batches full each iteration.
- Memory stays compact and reusable; fewer OOMs and higher throughput.

5.4.4 4. Concept Diagram

```
|-------------------------------------------------------------------------------|
| [Incoming Requests] -> Queue -> Scheduler -> GPU Worker -> PagedAttention ->  |
|                             ^                                    |             |
|                        Stream tokens back to clients continuously             |
|-------------------------------------------------------------------------------|
```

5.5 Section 4—Minimal "How to Run" Demos

5.5.1 4.1 Install

```
# Fresh environment recommended
# If you're in Jupyter, the line below installs into the notebook kernel.
# If you're in a real project, prefer requirements.txt or pyproject.toml
%pip install -U vllm transformers accelerate openai
```

5.5.2 4.2 Pick a model

Qwen3-4B-Instruct-2507

```
from vllm import LLM, SamplingParams
import torch

# Check GPU
if torch.cuda.is_available():
    print(f"GPU: {torch.cuda.get_device_name(0)}")
else:
    print("No GPU found, vLLM will be limited")

# Create the model
llm = LLM(
    model="Qwen/Qwen2-1.5B-Instruct",
    trust_remote_code=True,
)

# Sampling params
params = SamplingParams(
    temperature=0.7,
    top_p=0.9,
    max_tokens=200,
    skip_special_tokens=False
)

prompts = [
    "Tell me a short story about a robot.",
    "List three benefits of paged KV cache for LLM serving."
]

outputs = llm.generate(prompts, params)

for i, output in enumerate(outputs):
    print(f"\n{'='*50}")
    response = output.outputs[0].text
    print(f"Response {i+1}: {response}")
```

6 DeepSpeed Tutorial

Focus: **why DeepSpeed**, the core **ZeRO** idea.

6.1 Section 1 — Why DeepSpeed: The GPU Memory Problem in Training

6.1.1 1) Motivation

Training large models hits a **memory wall**. Even with big GPUs... the forward + backward pass stores many tensors simultaneously.

6.1.2 2) What consumes memory during training

Component	What it is	Notes
Model Parameters	The trainable weights	$\approx 1\times$ model size
Activations	Intermediates saved for backward	scales with batch×seq×hidden
Gradients	Produced by backprop	$\approx 1\times$ model size
Optimizer States	For Adam: momentum (m), variance (v)	$\approx 2\times$ model size
Temporary Buffers	Workspace for matmuls/communication	dynamic overhead

With Adam, per-GPU memory can be **4–6×** the model size.

6.1.3 3) Why naïve data parallel wastes memory

Every GPU **replicates**:

- model parameters
- gradients
- optimizer states

So if 4 GPUs train a model with Adam:

- each one keeps params
- each one keeps grads
- each one keeps Adam's `m`, `v` states

That means total memory across the cluster is 4× bigger than needed, but **each individual GPU** still has to fit:

- parameters
- gradients
- optimizer (2× params)

So 4 GPUs do **not** make a single GPU need less memory — they just let you do bigger *data* throughput.

6.1.4 4) Why model parallel is NOT a good idea

Model parallelism = splitting the computation across multiple GPUs instead of a full copy. Each GPU **computes a subset of layers**.

- **Manual partitioning** – Layers or tensors must be explicitly assigned to GPUs.
- **High communication cost** – GPUs must constantly exchange activations and gradients.
- **Sequential dependencies** – Layers depend on outputs from previous GPUs, creating idle "pipeline bubbles."
- **Backward complexity** – Gradients must flow across devices, increasing synchronization overhead.

In short: This reduces both memory *and* compute per GPU, but adds cross-GPU communication for every layer.

6.1.5 4) DeepSpeed's core idea

ZeRO (Zero Redundancy Optimizer) partitions these states **across** devices instead of replicating them, cutting memory per GPU roughly by the number of devices.

6.2 Section 2 — ZeRO: The Heart of DeepSpeed

6.2.1 Big idea

Don't replicate all training states on each GPU; **shard** them. Communication reconstructs what's needed on the fly.

6.2.2 ZeRO's three stages

Stage 1 — Shard optimizer states In optimizers like **Adam** or **AdamW**, at step t each trainable parameter θ_i has **optimizer states** that keep track of its historical updates — for example:

$$\begin{aligned} m_i^t &\leftarrow \beta_1 m_i^{t-1} + (1-\beta_1) g_i^t, \\ v_i^t &\leftarrow \beta_2 v_i^{t-1} + (1-\beta_2)(g_i^t)^2, \\ \theta_i^t &\leftarrow \theta_i^{t-1} - \alpha \frac{m_i^t/(1-\beta_1^t)}{\sqrt{v_i^t/(1-\beta_2^t)} + \varepsilon}, \end{aligned}$$

where

1. m_i = first moment (momentum term);
2. v_i = second moment (variance term);
3. g_i = gradient of that parameter;
4. θ_i = parameter value itself.

Each parameter θ_i updates only using its own m_i, v_i, and g_i. In vector form, Adam's update is applied elementwise:

$$\theta \leftarrow \theta - \alpha \frac{m}{\sqrt{v} + \varepsilon}.$$

So if we split the vector of parameters into chunks, each chunk can be updated entirely on its own — as long as it has access to its local m, v, and g.

Because the optimizer states are independent: - We can shard the m and v tensors across GPUs. - GPU 0 stores m, v for parameters [0˘25%], GPU 1 for [25˘50%], etc. - Each GPU updates its subset of parameters using its own local optimizer states.

Naïve DP:

- GPU0: params + grads + optimizer
- GPU1: params + grads + optimizer
- GPU2: params + grads + optimizer
- GPU3: params + grads + optimizer

So 4 copies of everything.

ZeRO-1: *partition* optimizer states across GPUs. Everyone still has:

- parameters
- gradients

but **only one GPU** keeps optimizer states for a subset of params.

```
GPU0: gets optimizer states[0-25%]
GPU1: gets optimizer states[25-50%]
GPU2: gets optimizer states[50-75%]
GPU3: gets optimizer states[75-100%]
```

No communication is needed during the optimizer step except possibly for syncing the updated parameters after.

Training Pipeline for ZeRO-1

```
Forward (parameters duplicated) → Backward (parameters, gradients duplicated) →
Local Update (ONLY its shard of optimizer states) → Broadcasting (updated parameters
```

Summary

- Save: optimizer memory
- Comm: low
- Still replicate: params + grads

Stage 2 — Shard gradients too In DP, after backward, each GPU has **full gradients**. That's redundant.

Idea: do a `reduce-scatter` instead of `all-reduce`.

- `all-reduce`: everyone ends with the full reduced gradient.
- `reduce-scatter`: the reduction happens, but the result is **sharded** across GPUs.

So:

- gradients are **partitioned**

- optimizer states are **partitioned**
- parameters are still **replicated**

Memory effect: you remove another $\approx 1\times$ model size per GPU.

Training Pipeline for ZeRO-2

```
Forward (params replicated) →
Backward (partial grads produced) →
Reduce-scatter (partition grads across GPUs) →
Local Update (ONLY its shard of optimizer states) →
Broadcasting (updated parameters to others)
```

Summary

- Gradients are partitioned using **reduce-scatter**.
- **Save:** optimizer + gradient memory.
- **Comm:** low.

Stage 3 — Shard parameters (everything) **Idea:** share model parameters

```
GPU0: gets \theta[0-25%]
GPU1: gets \theta[25-50%]
GPU2: gets \theta[50-75%]
GPU3: gets \theta[75-100%]
```

So now:

- parameters are sharded
- gradients are sharded
- optimizer states are sharded

That's why ZeRO-3 gives the biggest memory win.

ZeRO Stage	What's sharded	Memory reduction
1	Optimizer states	$\approx 2\times$
2	+ Gradients	$\approx 3\times$
3	+ Parameters	≈ 4–$8\times$

6.2.3 Illustration (conceptual)

```
Naive DP (replicated):   [full][full][full]
ZeRO-1 (opt sharded):    [P,G, O][P,G, O][P,G, O]
ZeRO-2 (opt+grad shard): [P, G,O][P, G,O][P, G,O]
ZeRO-3 (all sharded):    [P,G,O][P,G,O][P,G,O]
P=params, G=grads, O=optimizer shards
```

7 Low-Rank Adaptation (LoRA)

7.1 Motivation: Why LoRA?

The Problem: Modern large language models (LLMs) have trillions of parameters pretrained on massive datasets. Post-training typically uses much smaller datasets focused on specific domains. It seems wasteful to update trillions of parameters for gigabit/megabit training data.
The Solution: Parameter-Efficient Fine-Tuning (PEFT) adjusts large networks by updating only a small subset of parameters [**houlsby2019parameter**].
Key Insight: Post-training often requires much less capacity than pretraining, so we can represent updates efficiently with low-rank matrices.

7.2 LoRA Mathematical Foundation

Core Equation: The fundamental LoRA update is expressed as:

$$W' = W + \gamma BA \tag{11}$$

Where:

- $W \in \mathbb{R}^{N \times N}$: Original frozen weight matrix
- $A \in \mathbb{R}^{r \times N}$: Low-rank adapter matrix (input projection)
- $B \in \mathbb{R}^{N \times r}$: Low-rank adapter matrix (output projection)
- $\gamma = \alpha / r$: Scaling factor (maintains learning rate stability across ranks)
- $r \ll N$: Rank of the adaptation (typically 8–64)

Intuition: Instead of updating the full N^2 parameters in W, we learn a low-rank approximation of the update using only $2Nr$ parameters. This is the key insight behind LoRA [**hu2021lora**].

7.3 LoRA Advantages

7.3.1 Multi-tenant Serving

LoRA enables efficient multi-tenant serving by keeping original weights unchanged while storing multiple adapters in memory. This allows sampling from different model versions simultaneously in batched inference. Modern inference engines such as vLLM [**kwon2023efficient**] and SGLang [**zheng2023efficiently**] support this natively.

7.3.2 Memory Efficiency

- **Training**: No optimizer state for base weights (often stored in higher precision)
- **Storage**: Adapters are much smaller than full model checkpoints
- **Transfer**: Fast loading/transfer of small adapter files

7.3.3 Compute Efficiency

- **FLOPs per matrix**: Approximately $2N^2 + 6Nr$ vs $3N^2$ for full fine-tuning
- **Ratio**: Approximately 2/3 the compute when $r \ll N$
- **Scaling**: More efficient as model size increases

7.4 Can LoRA Match Full Fine-Tuning Performance?

Answer: Yes, under specific conditions.

7.4.1 The Key Question

Can LoRA match the performance of full fine-tuning, and if so, under which conditions? This question has been comprehensively investigated by Schulman et al. [**schulman2025lora**].

7.4.2 Conditions for Equal Performance

When LoRA Matches Full Fine-Tuning:

- **Dataset size**: Small-to-medium post-training datasets (typical instruction-tuning/reasoning)
- **Rank**: Sufficient rank to capture essential information
- **Layer coverage**: Apply LoRA to all weight matrices (especially MLP and MoE layers)

- **Hyperparameters**: Proper learning rate scaling with $\gamma = \alpha/r$

When LoRA Underperforms:

- **Capacity exceeded**: When dataset size exceeds LoRA parameter capacity
- **Attention-only**: Applying LoRA only to attention layers (even with matched parameter count)
- **Very large datasets**: Settings resembling pretraining with massive data

7.5 Experimental Setup and Results

7.5.1 Methodology

The experimental methodology follows the approach of Schulman et al. [**schulman2025lora**]:

- **Models**: Llama 3 series, Qwen3 (including MoE)
- **Datasets**: Tulu3 (instruction-following), OpenThoughts3 (reasoning)
- **Rank sweep**: 1 to 512 across 3 orders of magnitude
- **Learning rate**: Swept for each condition to eliminate LR confounds
- **Metrics**: Log loss (not sampling-based) for clean scaling laws

7.5.2 Key Findings

Learning Curves by Rank: High-rank LoRA ($r = 64$–512) overlaps with full fine-tuning, while low-rank LoRA ($r = 1$–8) underperforms when capacity is exceeded. This pattern is consistent across different model sizes and datasets.
Batch Size Sensitivity: LoRA is less tolerant of large batch sizes than full fine-tuning. This penalty is not mitigated by increasing rank and is a property of the product-of-matrices parametrization.
Layer Coverage Impact: Attention-only LoRA underperforms even with matched parameters. Full coverage (MLP + attention) performs significantly better. MoE layers are particularly important for LoRA effectiveness.
Learning Rate Impact: The optimal learning rate is approximately 10 times higher for LoRA than for full fine-tuning [**schulman2025lora**].

7.5.3 Compute Efficiency Analysis

- **Theoretical**: LoRA uses approximately 2/3 the FLOPs of full fine-tuning per weight matrix
- **Practical**: Often faster overall due to reduced memory bandwidth
- **Scaling**: Advantage increases with model size

7.6 Experimental Results on PBMC3k Dataset

Our experiments on the PBMC3k single-cell RNA sequencing dataset [**zheng2017massively**], which contains approximately 3,000 peripheral blood mononuclear cells with gene expression measurements for approximately 20,000 genes, confirm the findings of Schulman et al. [**schulman2025lora**]:

LoRA Full Coverage Matches Full Fine-Tuning:

- LoRA Full achieved accuracy within 1% of full fine-tuning
- Used only approximately 10% of trainable parameters
- Confirms the main theoretical finding

Attention-Only LoRA Underperforms:

- LoRA Attention-Only achieved lower accuracy than full coverage
- Even with similar parameter counts, layer coverage matters
- Critical insight: Apply LoRA to **all layers**, not just attention

Computational Efficiency:

- **Parameters**: LoRA trains 10$\times$ fewer parameters than full fine-tuning
- **Speed**: LoRA is approximately 1.5–2$\times$ faster in training time
- **Memory**: Significantly lower memory footprint (no optimizer state for frozen weights)

Hyperparameter Sensitivity:

- **Rank**: Higher ranks ($r = 16, 32$) approach full fine-tuning performance; very low ranks ($r = 2$) show capacity limits

- **Learning Rate**: LoRA is sensitive to learning rate and typically requires 5–10$\times$ higher learning rates
- **Batch Size**: Standard batch sizes work well for LoRA

7.7 Discussion and Key Takeaways

7.7.1 The "Low-Regret Regime"

Schulman et al. [**schulman2025lora**] identify a regime where LoRA performs similarly to full fine-tuning. This regime:

- Covers most post-training scenarios
- Enables efficient fine-tuning in many applications
- Makes powerful adaptation accessible with fewer resources

7.7.2 Practical Implications

- **When to use LoRA**: Most post-training scenarios, especially with limited compute
- **When to use full fine-tuning**: Very large datasets, when maximum performance is critical
- **Best practices**: Apply to all layers, use sufficient rank, proper learning rate scaling

7.8 Best Practices

Based on theoretical understanding and empirical validation:

Implementation Guidelines:

- Apply LoRA to all transformer layers (attention + MLP)
- Start with $r \in [16, 32]$ for medium-sized models
- Set scaling factor $\alpha = 2r$ as default
- Use learning rates 5–10$\times$ higher than for full fine-tuning
- Monitor validation performance to detect capacity limitations

When to Use LoRA:

- Post-training with limited computational resources
- Multi-task serving environments
- Rapid experimentation and iteration
- Storage and transfer constraints

When to Use Full Fine-Tuning:

- Very large datasets approaching pretraining scale
- Maximum performance critical regardless of cost
- Settings where low-rank assumption may not hold

7.9 Conclusion

Low-Rank Adaptation (LoRA) provides an effective and efficient approach to fine-tuning large pretrained models. Our analysis and experiments confirm that:

1. LoRA with full layer coverage matches full fine-tuning performance in the "low-regret regime"
2. Layer coverage is critical—applying LoRA only to attention layers underperforms
3. LoRA achieves 90–99% parameter reduction while maintaining performance
4. Higher learning rates (5–10×) are required compared to full fine-tuning
5. The approach generalizes across domains from NLP to computational biology

LoRA has become a foundational technique in modern deep learning, enabling efficient adaptation of large models while maintaining performance comparable to full fine-tuning.

8 Supervised Finetuning

Write your content here...

9 LLM Inference and Deployment

9.1 Motivation and Background

Inference is the process of using a trained model to generate outputs. While training often receives more attention in the research community, inference is where most of the value accrues in production systems. A single model may be trained once but serve billions of inference requests. Understanding efficient inference, deployment strategies, and optimization techniques is crucial for building practical AI systems.

Modern LLM inference involves several key challenges:

- **Cost management:** API calls can become expensive at scale, requiring careful model selection and optimization.
- **Latency requirements:** Real-time applications demand low-latency responses.
- **Throughput optimization:** Serving many concurrent users efficiently.
- **Capability extension:** Basic LLMs need tools and external knowledge to be truly useful.
- **Context management:** Handling long conversations and large documents.

9.2 API-Based Inference

The simplest way to run inference is through API services. Instead of managing infrastructure, researchers can access powerful models through standardized interfaces.

9.2.1 OpenRouter: Universal API Gateway

OpenRouter [**openrouter**] unifies multiple model providers into one API service, enabling seamless switching between different models and providers. This abstraction is valuable because:

- **Provider diversity:** Access to GPT-5, Claude Sonnet 4, Gemini, DeepSeek, and many others through one interface.
- **Automatic failover:** If one provider is down or rate-limited, requests can route to alternatives.

- **Cost optimization:** Easy comparison and switching between providers based on price/performance.
- **Model routing:** Automatic selection of best provider for a given model.

9.2.2 Model Selection Criteria

Choosing the right model involves multiple tradeoffs:

Cost Structure. Models are typically priced per million tokens, with separate rates for input and output:

- **DeepSeek V3.1 (free):** \$0/1M tokens (both input and output)
- **Gemini 2.5 Flash:** \$0.30/1M input, \$2.50/1M output
- **GPT-5:** \$5.00/1M input, \$15.00/1M output
- **Claude Sonnet 4:** \$3.00/1M input, \$15.00/1M output

Context Length. Different models support different maximum context windows:

- **Short context (32K):** Sufficient for most conversations
- **Medium context (128K):** Handles long documents
- **Long context (1M+):** Entire codebases or book-length documents

Modalities. Models support different input/output types:

- **Text-to-text:** Traditional language models
- **Text+image-to-text:** Vision-language models (e.g., GPT-5, Claude, Gemini)
- **Text+image-to-text+image:** Multimodal generation (e.g., Gemini 2.5)

Reasoning Capabilities. Some models support explicit reasoning modes:

- **Standard generation:** Direct token-by-token generation
- **Chain-of-thought:** Step-by-step reasoning in output
- **Reasoning effort control:** Models like GPT-5 allow explicit control over reasoning depth

9.3 Tool Calling and Function Integration

Tool calling extends LLM capabilities by allowing models to invoke external functions during generation. This paradigm shift transforms LLMs from pure text generators into orchestrators of complex workflows.

9.3.1 Why Tool Calling Matters

LLMs have inherent limitations:

- **No real-time information:** Training data has a cutoff date
- **No external knowledge:** Cannot access private databases or documents
- **Weak at computation:** Arithmetic and symbolic reasoning are unreliable
- **No actions:** Cannot directly interact with external systems

Tool calling addresses these limitations by giving models the ability to:

- Search the web or databases
- Execute code (Python, SQL, etc.)
- Access APIs (weather, calendar, email)
- Retrieve from knowledge bases
- Perform deterministic computations

9.3.2 Tool Calling Architecture

The typical tool calling flow follows this pattern:

1. **Tool Definition:** Functions are described in a structured schema (JSON):

```
{
  "name": "get_weather",
  "description": "Get current weather for a location",
  "parameters": {
    "type": "object",
    "properties": {
```

```
              "location": {"type": "string"},
              "units": {"type": "string", "enum": ["celsius", "fahrenheit"]}
            }
          }
        }

```

2. **Model Generation:** The LLM decides whether to call a tool and with what arguments.
3. **Tool Execution:** The application executes the function and returns results.
4. **Result Integration:** The model incorporates tool outputs into its response.

9.3.3 Sequential Tool Calling

Models can make multiple tool calls in sequence to accomplish complex tasks:

```
User: "What’s the weather in the capital of France?"
Model: [Calls search("capital of France")]
Tool: "Paris"
Model: [Calls get_weather("Paris")]
Tool: "15°C, partly cloudy"
Model: "The weather in Paris is 15°C and partly cloudy."
```

This capability enables agentic behavior where models can break down complex tasks into multiple steps.

9.4 Model Context Protocol (MCP)

Model Context Protocol [**mcp**] standardizes how LLMs interact with external tools and data sources. Unlike ad-hoc tool implementations, MCP provides:

- **Standardized interface:** Consistent API across different tools
- **Server-side execution:** Tools run on dedicated servers
- **Authentication:** Built-in support for secure access
- **Discovery:** Automatic tool schema detection

9.4.1 MCP Servers

Popular MCP servers include:

- **Notion:** Access to workspace documents and databases
- **Google Calendar:** Event management and scheduling
- **GitHub:** Repository access and code search
- **Slack:** Team communication integration
- **Custom servers:** Organizations can build domain-specific MCP servers

9.5 Prompt Engineering and Optimization

The inputs provided to LLMs dramatically affect output quality. Prompt engineering is the practice of carefully designing these inputs to achieve desired behaviors.

9.5.1 OpenAI Model Spec

OpenAI's Model Spec [**openai_spec**] defines a chain of command for instructions:

1. **Root:** Model Spec root sections (unchangeable)
2. **System:** Model Spec system sections and system messages (OpenAI-controlled)
3. **Developer:** Developer messages and instructions (application-level)
4. **User:** User messages (end-user input)
5. **Guideline:** Model Spec guideline sections (suggestions)

Developer-level instructions (often called "system prompts") allow applications to guide model behavior while respecting safety constraints.

9.5.2 Prompt Optimization with GEPA

Gradient-free Prompt Evolution Algorithm (GEPA) [**gepa**] automates prompt optimization:

- **Iterative improvement:** Evolves prompts through multiple generations
- **Metric-driven:** Optimizes for user-defined success criteria

- **Reflection:** Uses model self-evaluation to guide improvements
- **No gradients:** Works with any LLM API

The typical GEPA workflow:

1. Define evaluation metric (e.g., accuracy on test set)
2. Start with baseline prompt
3. Model reflects on failures and suggests improvements
4. Test improved prompts
5. Iterate until convergence

In experiments, GEPA can improve prompt performance by 10–40% on challenging tasks like mathematical reasoning.

9.6 Context Window Management

Long context windows enable powerful applications but introduce challenges:

9.6.1 Context Length vs. Quality

Models exhibit performance degradation with very long contexts:

- **Lost in the middle:** Information in the middle of long contexts is harder to retrieve
- **Attention dilution:** With limited attention capacity, relevant information gets less focus
- **Context rot:** Performance degrades as context approaches maximum length

9.6.2 Mitigation Strategies

Several techniques address context limitations:

- **Conversation summarization:** Compress long histories into concise summaries
- **Retrieval augmentation:** Fetch only relevant context on-demand

- **Hierarchical processing:** Process documents in chunks with aggregation
- **Attention steering:** Explicitly guide model focus to important sections

9.7 Practical Considerations

9.7.1 Cost Management

Inference costs can scale rapidly. Key strategies:

- **Model routing:** Use cheaper models for simple queries, expensive ones for hard problems
- **Caching:** Store and reuse responses for common queries
- **Batching:** Group multiple requests for efficiency
- **Output length limits:** Constrain max_tokens to control costs

9.7.2 Latency Optimization

For real-time applications:

- **Streaming:** Display tokens as they're generated
- **Speculative decoding:** Generate multiple candidates in parallel
- **Model selection:** Use smaller, faster models when appropriate
- **Regional providers:** Choose geographically close API endpoints

9.7.3 Reliability

Production systems need robust error handling:

- **Rate limit handling:** Implement exponential backoff
- **Failover:** Route to alternative providers when needed
- **Validation:** Check output format and content
- **Monitoring:** Track costs, latency, and error rates

9.8 Summary

Effective LLM inference requires understanding:

- Model selection tradeoffs (cost, performance, capabilities)
- Tool calling for extending capabilities
- Prompt engineering for optimal outputs
- Context management for long conversations
- Practical considerations for production deployment

The inference ecosystem continues to evolve rapidly, with new models, techniques, and best practices emerging regularly. The skills covered in this section provide a foundation for building robust, cost-effective LLM applications.

10 LLM Evaluation and Alignment

10.1 Motivation: Why Evaluation Matters

As language models grow in capability, rigorous evaluation becomes essential for:

- **Progress measurement:** Tracking improvements across model versions
- **Model selection:** Choosing appropriate models for specific tasks
- **Research validation:** Demonstrating the effectiveness of new techniques
- **Safety assessment:** Identifying potential harms and failure modes
- **Capability mapping:** Understanding what models can and cannot do

However, LLM evaluation presents unique challenges:

- **Open-ended generation:** Many valid responses to the same prompt
- **Capability breadth:** Models span reasoning, coding, creativity, and more
- **Benchmark contamination:** Models may have seen evaluation data during training
- **Gaming metrics:** Optimizing for benchmarks without genuine improvement
- **Alignment vs. capability:** Tradeoffs between helpfulness, honesty, and harmlessness

10.2 Evaluation Methodologies

10.2.1 Benchmark Design Principles

Effective benchmarks share common characteristics:

- **Diverse tasks:** Cover multiple domains and capabilities
- **Difficulty range:** Include both easy and hard examples
- **Clear metrics:** Objective, reproducible scoring
- **Low contamination:** Minimize training data overlap
- **Regular updates:** Refresh to avoid overfitting

10.2.2 Common Evaluation Paradigms

Multiple Choice. Present options and measure selection accuracy:

- **Advantages:** Objective scoring, easy to scale
- **Disadvantages:** May not reflect real usage, susceptible to guessing
- **Examples:** MMLU, HellaSwag, ARC

Exact Match. Compare generated text to reference answers:

- **Advantages:** Clear success criterion
- **Disadvantages:** Inflexible, penalizes valid alternatives
- **Examples:** GSM8K (math), HumanEval (code)

Human Evaluation. Expert judgment of outputs:

- **Advantages:** Captures nuanced quality
- **Disadvantages:** Expensive, slow, subjective
- **Examples:** Chatbot Arena, LMSYS leaderboard

LLM-as-Judge. Use strong models to evaluate weaker ones:

- **Advantages:** Scalable, captures complex criteria
- **Disadvantages:** Inherits judge model biases
- **Examples:** AlpacaEval, MT-Bench

10.3 The LM Evaluation Harness

The `lm-evaluation-harness` [**lmeval**] from EleutherAI provides a unified framework for evaluating language models across dozens of benchmarks.

10.3.1 Key Features

- **Standardized interface:** Consistent API across 200+ tasks
- **Reproducibility:** Fixed random seeds and evaluation protocols
- **Efficiency:** Batched inference and caching
- **Extensibility:** Easy to add custom tasks
- **Model support:** HuggingFace, vLLM, OpenAI API, and more

10.3.2 Usage Pattern

A typical evaluation workflow:

```
lm_eval --model hf \
  --model_args pretrained=meta-llama/Llama-3.1-8B \
  --tasks mmlu,gsm8k,hellaswag \
  --batch_size 8 \
  --output_path results/
```

This produces a detailed report with per-task metrics, aggregate scores, and statistical confidence intervals.

10.3.3 Custom Task Definition

Users can define new evaluation tasks in YAML:

```
task: my_custom_task
dataset_name: my_org/my_dataset
output_type: multiple_choice
metric: acc
num_fewshot: 5
```

This flexibility enables evaluation on proprietary or domain-specific benchmarks.

10.4 Alignment: Beyond Capability

Alignment ensures models are helpful, honest, and harmless—not just capable.

10.4.1 Reinforcement Learning from Human Feedback (RLHF)

RLHF [**rlhf**] trains models to generate outputs humans prefer:

Phase 1: Supervised Fine-Tuning (SFT). Train on high-quality demonstration data:

- Collect expert demonstrations
- Fine-tune base model to imitate
- Produces helpful but not necessarily preferred outputs

Phase 2: Reward Model Training. Learn a preference model from comparisons:

- Present pairs of model outputs
- Humans indicate which is better
- Train classifier to predict preferences
- Reward model scores: $r_\theta(x, y)$

Phase 3: RL Policy Training. Optimize policy to maximize reward:

$$\max_\pi \mathbb{E}_{x \sim \mathcal{D}, y \sim \pi(\cdot|x)}[r_\theta(x, y) - \beta \mathrm{KL}(\pi \| \pi_{\mathrm{ref}})] \tag{12}$$

The KL penalty prevents the policy from deviating too far from the reference model, avoiding reward hacking.

10.4.2 Direct Preference Optimization (DPO)

DPO [**dpo**] simplifies RLHF by directly optimizing on preference data:

- **No reward model:** Directly use preference pairs
- **Simpler training:** Standard supervised learning
- **Better stability:** Avoids RL optimization challenges

The DPO loss:

$$\mathcal{L}_{\text{DPO}} = -\mathbb{E}_{(x,y_w,y_l)\sim\mathcal{D}}\left[\log\sigma\left(\beta\log\frac{\pi_\theta(y_w|x)}{\pi_{\text{ref}}(y_w|x)} - \beta\log\frac{\pi_\theta(y_l|x)}{\pi_{\text{ref}}(y_l|x)}\right)\right] \tag{13}$$

where y_w is the preferred completion and y_l is the dispreferred one.

10.4.3 Constitutional AI

Constitutional AI [**constitutional**] encodes explicit principles:

- **Self-critique:** Model critiques its own outputs
- **Self-revision:** Generates improved versions
- **Preference learning:** Learns from self-comparisons
- **Transparency:** Explicit ethical guidelines

This approach reduces reliance on human labels while maintaining alignment.

10.5 Common Evaluation Pitfalls

10.5.1 Benchmark Contamination

Models may have seen evaluation data during pretraining:

- **Detection:** Check for memorization with perturbed examples
- **Mitigation:** Use private test sets, regular benchmark rotation
- **Reporting:** Disclose known overlaps transparently

10.5.2 Gaming the Metric

Optimizing for benchmarks can degrade real-world performance:

- **Goodhart's Law:** "When a measure becomes a target, it ceases to be a good measure"
- **Solution:** Evaluate on diverse, held-out tasks
- **Holistic assessment:** Combine quantitative and qualitative evaluation

10.5.3 Multi-Issue Problems

A single output may fail in multiple ways:

- Incorrect reasoning *and* poor formatting
- Helpful content *but* potential harm
- Correct answer *but* inefficient approach

Effective evaluation must disentangle these dimensions.

10.5.4 The Alignment Tax

Alignment techniques may reduce capability:

- **Observation:** Aligned models sometimes score lower on capability benchmarks
- **Explanation:** Safety constraints limit certain behaviors
- **Debate:** Whether this represents genuine capability loss or appropriate restraint

Research continues on alignment methods that preserve or enhance capabilities.

10.6 Emerging Evaluation Paradigms

10.6.1 Agentic Evaluation

As models become more agentic, evaluation must adapt:

- **Multi-step tasks:** Measure planning and execution
- **Tool use:** Evaluate when and how models call tools
- **Long-horizon tasks:** Track performance over extended interactions
- **Failure recovery:** Assess adaptation when plans go wrong

10.6.2 Real-World Deployment Metrics

Beyond benchmarks, track production performance:

- **User satisfaction:** Thumbs up/down, engagement time
- **Task success rate:** Did the user accomplish their goal?

- **Safety incidents:** Harmful outputs, jailbreaks
- **Cost efficiency:** Quality per dollar spent

10.7 Best Practices

For Researchers:

- Report results on diverse benchmarks
- Include statistical significance tests
- Disclose evaluation protocols completely
- Consider both capability and alignment
- Test for benchmark contamination

For Practitioners:

- Evaluate on your specific use cases
- Combine automated and human evaluation
- Monitor performance over time
- A/B test model changes
- Track both quality and cost metrics

10.8 Summary

Effective LLM evaluation requires:

- Understanding evaluation paradigms and their tradeoffs
- Using standardized tools like `lm-eval-harness`
- Considering alignment alongside capability
- Avoiding common pitfalls (contamination, gaming)
- Adapting to emerging evaluation needs (agentic systems)

As models become more capable, evaluation methodologies must evolve to capture new dimensions of performance, safety, and alignment.

11 Limitations, Future Works, and Conclusions

Acknowledgments