STANFORD UNIVERSITY

Complete Study Notes

Large Language Models

Full 9-Lecture Course

L01 Transformer Architecture	**L02** Transformer-Based Models & Tricks	**L03** Transformers & Large Language Models
L04 LLM Training	**L05** LLM Tuning	**L06** LLM Reasoning
L07 Agentic LLMs	**L08** LLM Evaluation	**L09** Recap & Current Trends

50+ Pages of Clear, Simple Explanations · Key Concepts · Examples · Analogies

Table of Contents

LECTURE 1

Transformer Architecture

The Foundation of Modern AI — Attention Is All You Need

1.1 Why Did We Need Transformers?

Before Transformers, the dominant models for language tasks were **Recurrent Neural Networks (RNNs)** and **LSTMs**. These models read text word by word, left to right, carrying a "hidden state" that tried to remember everything seen so far. This created two big problems:

- **Vanishing Gradient Problem:** In long sentences, information from early words would "fade away" by the time the model reached later words, making it hard to learn long-range dependencies.
- **Sequential Processing:** Because each word had to be processed one at a time, training was very slow — you couldn't use modern parallel GPUs efficiently.

In 2017, a Google team published the famous paper **"Attention Is All You Need"** which introduced the Transformer. The key insight: instead of reading text sequentially, **let every word look at every other word simultaneously** using a mechanism called Attention.

1.2 The Big Picture: Encoder-Decoder Architecture

The original Transformer has two main parts — an **Encoder** and a **Decoder**. Think of it like translating a language: the Encoder reads the input sentence and creates a rich representation of its meaning; the Decoder then generates the output sentence one word at a time.

Encoder	Reads and understands the input. Used in models like BERT. Good for classification, understanding tasks.
Decoder	Generates output text. Used in models like GPT. Good for text generation, completion.
Encoder-Decoder	Both combined. Used in models like T5, BART. Good for translation, summarization.

1.3 Self-Attention: The Heart of the Transformer

Self-attention allows each word to "attend" to (pay attention to) every other word in the sentence, including itself. The key intuition: **the meaning of a word depends on its context**. For example, the word "bank" means something different in "river bank" vs "savings bank."

How does it work? Three magic vectors: Q, K, V

For each word, we create three vectors using learned weight matrices:

- **Query (Q):** "What am I looking for?" — represents what this word wants to find.
- **Key (K):** "What do I have?" — represents what each word offers.

- **Value (V):** "What information should I share?" — the actual content.

> ***Simple Analogy:*** *Think of a library. Your Query is your search request. Each book's Key is its catalog entry. The Value is the book's actual content. You match your Query to all the Keys, find the most relevant books, and read their Values.*

Mathematically: $\mathbf{Attention(Q, K, V) = softmax(QK^T / \sqrt{d_k}) \times V}$

The division by √d_k (square root of key dimension) prevents the dot products from getting too large, which would push softmax into regions with very small gradients.

1.4 Multi-Head Attention

Instead of performing attention once, the Transformer performs it **multiple times in parallel** with different learned weight matrices — these are called "heads." Each head can learn to attend to different types of relationships simultaneously.

> **Example:** In the sentence "The animal didn't cross the street because it was too tired," one attention head might learn that "it" refers to "animal," while another head focuses on the word "tired" being connected to "animal" for sentiment purposes.

Typical configurations: 8, 12, or 16 attention heads. Outputs from all heads are concatenated and projected.

1.5 Positional Encoding

Unlike RNNs, Transformers process all words simultaneously, so they have no built-in sense of word order. **Positional Encoding** solves this by adding information about each word's position to its embedding.

The original paper uses sine and cosine functions of different frequencies — essentially a unique "position fingerprint" for each location in the sequence. Modern models often use **Rotary Position Embedding (RoPE)** or **Learned Positional Embeddings**.

1.6 Feed-Forward Network (FFN)

After attention, each position passes through a **Feed-Forward Network** — two linear layers with a ReLU or GELU activation in between. This is applied independently to each position. It acts like a "thinking" step where the model processes and transforms the attended information.

> *Typical FFN size: 4× the model dimension. So if the model has dimension 768, FFN has inner dimension 3072.*

1.7 Layer Normalization and Residual Connections

Two crucial training tricks make Transformers stable:

- **Residual Connections (Skip Connections):** Add the input of a layer directly to its output — Output = LayerNorm(x + Sublayer(x)). This helps gradients flow back during training and allows the model to "skip" layers if needed.
- **Layer Normalization:** Normalizes the activations to have zero mean and unit variance, keeping training stable. Applied before (Pre-LN) or after (Post-LN) each sub-layer.

1.8 Putting It All Together: One Transformer Layer

Step	Operation	What It Does
1	Input Embedding + Positional Encoding	Convert tokens to vectors, add position info
2	Multi-Head Self-Attention	Each word attends to all other words
3	Add & Norm	Residual connection + Layer Normalization
4	Feed-Forward Network	Transform each position independently
5	Add & Norm	Another residual connection + Layer Norm

Key Insight: A full Transformer stacks many of these layers (e.g., 12 layers for BERT-base, 96 layers for GPT-4). Each layer refines the representation, understanding language at progressively higher levels of abstraction.

LECTURE 2

Transformer-Based Models & Tricks

BERT, GPT, T5 and the Engineering Magic Behind Them

2.1 The Three Families of Transformer Models

Based on which parts of the Transformer architecture they use, models fall into three families:

Encoder-Only (BERT-style)	See the FULL sentence at once. Great for understanding tasks like classification, NER, question answering. Bidirectional context.
Decoder-Only (GPT-style)	Only see past tokens — left to right. Great for text generation. Most modern LLMs use this approach.
Encoder-Decoder (T5-style)	Best for sequence-to-sequence tasks: translation, summarization, question answering with generated answers.

2.2 BERT: Bidirectional Encoder Representations from Transformers

BERT (Google, 2018) revolutionized NLP by pre-training a deep bidirectional Transformer encoder on massive unlabeled text. Key innovation: BERT reads text in **both directions** simultaneously.

Pre-training Tasks:

- **Masked Language Modeling (MLM):** Randomly mask 15% of tokens; predict them. Like a fill-in-the-blank test. Forces the model to understand context from both sides.
- **Next Sentence Prediction (NSP):** Given two sentences, predict if the second follows the first. Helps learn sentence-level relationships.

BERT variants: *RoBERTa (more data, no NSP), ALBERT (parameter sharing), DistilBERT (distilled, smaller), DeBERTa (disentangled attention). All follow the same encoder-only paradigm.*

2.3 GPT: Generative Pre-trained Transformer

GPT (OpenAI) uses only the Transformer decoder with **causal (left-to-right) self-attention**. Each token can only attend to previous tokens — it cannot "cheat" by looking at future tokens. This makes it naturally suited for text generation.

Pre-training: Simple next-token prediction — given a sequence of tokens, predict the next one. This is called **Language Modeling**.

GPT-1 → GPT-2 → GPT-3 → GPT-4: Each generation scaled up parameters, data, and compute dramatically. GPT-3 had 175 billion parameters and showed surprising emergent abilities.

2.4 T5: Text-to-Text Transfer Transformer

T5 (Google, 2020) reframes **every NLP task as text-to-text**. Classification becomes "Classify: [text] → positive." Translation becomes "Translate English to French: [text] → [french]." This unified framework allows one model to handle all tasks.

2.5 Important Training Tricks

Tokenization:

Raw text must be converted to numbers. Modern models use **subword tokenization** — words are split into frequent subword pieces. This handles rare words and different languages efficiently.

- **Byte-Pair Encoding (BPE):** Used by GPT. Iteratively merges the most frequent pairs of characters/subwords.
- **WordPiece:** Used by BERT. Similar to BPE but uses likelihood instead of frequency for merges.
- **SentencePiece:** Language-agnostic tokenizer, operates directly on raw text without pre-tokenization.

Embeddings:

Words are mapped to dense vectors in a high-dimensional space (e.g., 768 or 1024 dimensions). Similar words end up close to each other. The famous example: king – man + woman ≈ queen.

Dropout:

During training, randomly "drop" (set to zero) some neurons with probability p (typically 0.1). This prevents overfitting by making the network more robust. At inference, all neurons are used but scaled by (1-p).

2.6 Scaling Laws

Research by OpenAI and DeepMind showed that model performance improves **predictably** as you scale three things: number of parameters (N), amount of training data (D), and compute budget (C). This led to the famous **Chinchilla scaling laws** (Hoffmann et al., 2022):

> **Key Finding:** Previous models (like GPT-3) were undertrained — they used too many parameters relative to their data. The Chinchilla rule: train on roughly **20 tokens per parameter** for compute-optimal training. A 70B model should see ~1.4 trillion tokens.

2.7 Efficient Attention Mechanisms

Standard attention has $O(n^2)$ complexity — it scales quadratically with sequence length. For long documents, this becomes prohibitively expensive. Several solutions have emerged:

- **Sparse Attention:** Only attend to a subset of tokens (nearby tokens + some global tokens).
- **Linear Attention:** Approximates attention in O(n) time using kernel methods.
- **FlashAttention:** Same mathematical result as standard attention, but uses memory-efficient GPU kernels — 2-4× faster in practice.
- **Grouped Query Attention (GQA):** Multiple query heads share key-value heads, reducing memory for the KV cache.

LECTURE 3

Transformers & Large Language Models

From Pre-training to Prompting — How LLMs Actually Work

3.1 What Is a Language Model?

A language model assigns a **probability to sequences of text**. Given words seen so far, it predicts the probability distribution over the next word. Formally, it models P(w_t | w_1, ..., w_{t-1}). Training: maximize the likelihood of the training data (minimize cross-entropy loss).

> ***Analogy:*** *Your phone's autocomplete is a tiny language model. It predicts the next word based on your typing history and context. LLMs do this at massive scale with billions of parameters and trillions of tokens of training data.*

3.2 The Pre-training Revolution

Before LLMs, NLP models were trained from scratch for each specific task. The paradigm shift: **pre-train once on massive data, then adapt**. Pre-training teaches the model general knowledge about language, facts, reasoning, and the world.

Pre-training Data	Terabytes of internet text, books, code, Wikipedia, papers. Often trillions of tokens.
Pre-training Objective	Next-token prediction (GPT-style) or masked token prediction (BERT-style).
Pre-training Cost	Millions of dollars. GPT-4 estimated ~$100M+. Requires thousands of GPUs for weeks.
Result	A "base model" with broad capabilities but no specific instruction-following behavior.

3.3 Emergent Abilities

One of the most surprising discoveries: as models scale up, they suddenly develop capabilities that smaller models completely lack. These are called **emergent abilities** — they appear abruptly at certain scale thresholds.

Examples of emergent abilities:

- **In-context learning:** Learning from examples provided in the prompt without any weight updates.
- **Chain-of-thought reasoning:** Ability to solve multi-step math problems when prompted to "think step by step."
- **Code generation:** Writing functional code in multiple programming languages.
- **Multi-step instruction following:** Understanding and executing complex multi-part instructions.

These abilities appear around 10-100 billion parameters, though the exact thresholds are debated.

3.4 Prompting: Talking to LLMs

A **prompt** is the input text given to an LLM. The way you write a prompt dramatically affects the output. Prompt engineering is the art and science of crafting effective prompts.

Zero-Shot Prompting:

Ask the model to perform a task with no examples. Just describe what you want:

```
"Classify the sentiment of this review as positive or negative: 'The food was amazing!'"
```

Few-Shot Prompting (In-Context Learning):

Provide a few examples of input-output pairs before your actual query:

```
"Review: Great food! → Sentiment: Positive Review: Terrible service! → Sentiment:
Negative Review: Average experience. → Sentiment: "
```

Chain-of-Thought (CoT) Prompting:

Ask the model to reason step by step. This dramatically improves performance on math, logic, and multi-step problems. Magic phrase: **"Let's think step by step"** or "Think carefully before answering."

> *Example improvement: On GSM8K (grade school math), CoT prompting can improve accuracy from ~20% to ~60%+ on large models.*

3.5 Context Window and KV Cache

The **context window** is the maximum number of tokens an LLM can process at once. Early GPT-3: 4,096 tokens. Modern models: 128K to 1M+ tokens (Claude, Gemini).

The **Key-Value (KV) Cache** stores computed key and value vectors from previous tokens. During generation, new tokens only need to compute attention with cached K, V vectors, not recompute everything from scratch. This dramatically speeds up inference.

3.6 Temperature and Sampling Strategies

When generating text, the model produces a probability distribution over the vocabulary. How you **sample** from this distribution determines the output style:

Greedy Decoding	Always pick the highest probability token. Fast but repetitive, often boring.
Temperature (T)	T<1 = more focused/deterministic. T>1 = more random/creative. T=1 = unchanged.
Top-K Sampling	Sample only from the top K most likely tokens. Typical K=50.
Top-P (Nucleus)	Sample from smallest set of tokens whose cumulative probability ≥ P. Typical P=0.9 or 0.95.
Beam Search	Keep top B candidate sequences at each step. Better quality but slower, less diverse.

LECTURE 4

LLM Training

Data, Compute, Optimization — How We Train Billion-Parameter Models

4.1 The Training Pipeline Overview

Training an LLM is a massive engineering and scientific challenge. It involves: collecting and cleaning enormous datasets, distributing training across thousands of GPUs, choosing the right optimization algorithms, and monitoring for training instabilities.

4.2 Data: The Foundation

Modern LLMs are trained on **trillions of tokens** from diverse sources. Data quality matters as much as quantity — "garbage in, garbage out."

Common Data Sources:

- **Common Crawl:** Web scrapes of the entire internet — massive but noisy. Requires heavy cleaning.
- **Books:** Project Gutenberg, Books3 — high quality, long-form reasoning.
- **Wikipedia:** Factual, structured, well-written — heavily used despite being small.
- **GitHub/Code:** Programming code — not just for coding, also improves reasoning.
- **ArXiv, PubMed:** Scientific papers — adds technical and scientific knowledge.

Data Cleaning Steps:

Language filtering (remove non-English if needed), deduplication (exact and near-duplicate removal), quality filtering (perplexity-based, heuristic rules), content filtering (remove harmful/personal content).

> **Key Insight:** Research shows that high-quality curated data can outperform much larger amounts of raw internet text. The Phi-1 and Phi-2 models from Microsoft showed that training a small model on "textbook-quality" data achieved performance comparable to much larger models.

4.3 Optimization: How the Model Learns

Training uses **gradient descent** — repeatedly computing how wrong the model is (loss), how to change each parameter to reduce that error (gradient), and taking a small step in that direction.

Adam Optimizer:

The standard for LLM training. Adam combines momentum (remembering past gradients) with adaptive learning rates (different rates for different parameters). Key hyperparameters: learning rate (typically 1e-4 to 3e-4), $\beta1=0.9$, $\beta2=0.999$, $\varepsilon=1e-8$.

Learning Rate Scheduling:

- **Warmup:** Start with a very small learning rate and increase gradually for the first few thousand steps. Prevents instabilities at the start.

- **Cosine Decay:** After warmup, gradually decrease the learning rate following a cosine curve, reaching near-zero at the end of training.

Gradient Clipping:

Cap the gradient norm at a threshold (e.g., 1.0). Prevents "exploding gradients" where a single bad batch causes catastrophic parameter updates.

4.4 Distributed Training: Using Thousands of GPUs

A single GPU cannot hold billions of parameters or process enough data. Distributed training spreads the work across many GPUs using three key strategies:

Data Parallelism	Each GPU gets a copy of the full model but a different batch of data. Gradients are averaged across GPUs (AllReduce). Easiest to implement.
Model Parallelism (Tensor)	Split individual layers across GPUs — e.g., each GPU handles a subset of attention heads. Used when a single layer is too large.
Pipeline Parallelism	Split the layers (depth-wise) across GPUs. GPU 1 handles layers 1-8, GPU 2 handles layers 9-16, etc. Requires careful scheduling to avoid bubbles.
3D Parallelism	Combine all three. Used for training the largest models (GPT-4, Llama 3).

4.5 Mixed Precision Training

To save memory and speed up training, modern LLM training uses **mixed precision**: weights are stored in FP32 (full precision) but computations happen in FP16 or BF16 (half precision). BF16 (Brain Float) is preferred over FP16 because it has the same exponent range as FP32, making it more numerically stable.

4.6 Gradient Checkpointing

During backpropagation, we need to store all intermediate activations — this consumes enormous memory. Gradient checkpointing saves memory by recomputing activations during the backward pass instead of storing them. Trade-off: ~33% more compute, but much less memory — often enabling training of much larger models.

4.7 Training Instabilities and Solutions

Common problems during large-scale training:

- **Loss Spikes:** Sudden increases in loss, often from a bad batch of data or numerical overflow. Solution: gradient clipping, data quality filtering, loss spike detection and rollback.
- **Training Divergence:** Loss goes to NaN or infinity. Often fixed by reducing learning rate or switching to BF16.
- **Dead Attention Heads:** Some attention heads become inactive. Pre-LN (layer norm before attention) helps avoid this.

LECTURE 5

LLM Tuning

Fine-Tuning, RLHF, and Parameter-Efficient Methods

5.1 Why Fine-Tuning?

A pre-trained base model knows a lot about language but doesn't know how to be helpful, harmless, or honest. It might complete a question rather than answer it, or refuse harmless requests. Fine-tuning adapts the base model to **follow instructions and align with human preferences**.

> The journey from base model to helpful assistant typically involves: (1) Supervised Fine-Tuning (SFT) and (2) Reinforcement Learning from Human Feedback (RLHF).

5.2 Supervised Fine-Tuning (SFT)

The simplest approach: collect a dataset of **(instruction, ideal response)** pairs, then fine-tune the model on these pairs using standard next-token prediction. The model learns to generate helpful, well-formatted responses.

Key dataset examples: OpenAI's InstructGPT data, Alpaca (52K GPT-4 generated examples), FLAN (mixing many tasks). Quality of demonstrations matters enormously — a small high-quality dataset often beats a large noisy one.

5.3 RLHF: Reinforcement Learning from Human Feedback

SFT has a limitation: it's hard for humans to write perfect responses, but much easier to compare two responses and say which is better. RLHF exploits this: train a model to generate responses that humans prefer.

Step 1 — Reward Model (RM) Training:

Collect pairs of model responses to the same prompt. Have humans rank them. Train a separate **Reward Model** (also a Transformer) to predict the human preference score. The RM outputs a scalar score: higher = better response.

Step 2 — RL Fine-Tuning with PPO:

Use **Proximal Policy Optimization (PPO)** — an RL algorithm — to fine-tune the LLM to maximize the reward model's score. A KL-divergence penalty keeps the model from straying too far from the original SFT model (prevents reward hacking).

> ***Reward Hacking:*** *Without the KL penalty, models learn to exploit the reward model — generating technically high-scoring but actually bad outputs. Example: generating very long, repetitive text that happens to score well on a flawed reward model.*

5.4 Direct Preference Optimization (DPO)

RLHF with PPO is complex and unstable. DPO (2023) is an elegant simplification: it directly trains the LLM on preference pairs **without a separate reward model** or RL loop. DPO reformulates the RL objective into a classification loss that can be optimized directly.

> DPO has become the dominant alternative to RLHF in practice — simpler, more stable, and achieves competitive results. Most open-source models (Llama, Mistral instruct versions) use DPO.

5.5 Constitutional AI (CAI)

Anthropic's approach: instead of human labelers rating responses, use **AI feedback guided by a "constitution"** — a set of principles. The model critiques its own outputs according to the principles, then revises them. A reward model trained on AI feedback replaces human raters. This is called RLAIF (RL from AI Feedback).

5.6 Parameter-Efficient Fine-Tuning (PEFT)

Full fine-tuning means updating all billions of parameters. This is expensive and risks catastrophic forgetting (losing pre-trained knowledge). PEFT methods update only a small subset of parameters, achieving similar results at much lower cost.

LoRA: Low-Rank Adaptation

The most popular PEFT method. Key insight: weight updates during fine-tuning tend to be **low-rank** — they live in a lower-dimensional subspace.

```
LoRA freezes the original weights W and adds two small matrices A and B: W_new = W + BA
(where A ∈ R^{d×r}, B ∈ R^{r×k}, and r << min(d,k)).
```

> *Only A and B are trained. With rank r=8 or r=16, LoRA updates <1% of parameters while achieving near-full fine-tuning performance. QLoRA applies LoRA to 4-bit quantized models, enabling fine-tuning of 70B models on a single GPU.*

Prefix Tuning / Prompt Tuning:

Add trainable "virtual tokens" (prefixes) to the input or activations. Only these prefixes are updated. Even simpler: **Prompt Tuning** adds trainable embeddings only at the input layer. Works well for large models (>10B parameters).

Adapter Layers:

Insert small bottleneck layers (down-project → nonlinearity → up-project) inside each Transformer layer. Only these adapters are trained. Popular for multi-task scenarios.

5.7 Catastrophic Forgetting and Continual Learning

When fine-tuning on new tasks, models often "forget" previously learned knowledge — this is **catastrophic forgetting**. Mitigation strategies include: mixing fine-tuning data with pre-training data, elastic weight consolidation (EWC), and using PEFT methods (frozen base model is protected).

LECTURE 6

LLM Reasoning

Chain-of-Thought, Tool Use, and Teaching Models to Think

6.1 What Is Reasoning for LLMs?

Reasoning refers to the ability to solve problems that require **multiple steps, logical inference, mathematical operations, or planning**. Early LLMs struggled with even basic arithmetic. The key insight that changed everything: **if you ask the model to show its work, it performs much better**.

6.2 Chain-of-Thought (CoT) Prompting — Deep Dive

Introduced by Wei et al. (2022), CoT prompting asks the model to generate intermediate reasoning steps before giving a final answer. This simple change leads to dramatic improvements on math, commonsense, and symbolic reasoning benchmarks.

Standard vs. CoT Example:

```
Standard: "Roger has 5 balls. He buys 2 more cans of balls with 3 balls each. How many balls?" → Model might say "11" directly. CoT: "Roger starts with 5 balls. He buys 2 cans × 3 balls = 6 balls. 5 + 6 = 11. The answer is 11." → Correct and explainable.
```

Key variants:

- **Zero-Shot CoT:** Just add "Let's think step by step" — no examples needed. Surprisingly effective on large models.
- **Few-Shot CoT:** Provide several worked examples with reasoning chains. More reliable but requires prompt engineering.
- **Self-Consistency:** Sample multiple reasoning paths with high temperature, then take majority vote on the final answer. Big improvements on math.
- **Least-to-Most Prompting:** Break hard problems into simpler sub-problems and solve them in order.

6.3 Tool-Augmented LLMs

LLMs have a fundamental limitation: they cannot update their knowledge after training, and they make arithmetic errors. Solution: give the LLM access to **external tools**.

Calculator / Code Executor	Model writes code (Python) to compute answers. Avoids arithmetic mistakes.
Search Engine	Retrieve current information from the web. Addresses knowledge cutoff problem.
Database	Query structured data (SQL) for precise factual retrieval.
APIs	Weather, maps, stock prices, calendars — real-world data access.
Memory / Vector Store	Retrieve relevant past context or documents (RAG).

ReAct Framework: Interleave Reasoning (think about what to do) and Acting (use tools). The model generates "Thought: ...", "Action: ...", "Observation: ..." cycles until it has an answer.

6.4 Retrieval-Augmented Generation (RAG)

A powerful technique to give LLMs access to a custom knowledge base without retraining:

1. **Index:** Convert documents to vector embeddings using an embedding model. Store in a vector database.
2. **Retrieve:** For each query, find the most similar document chunks using cosine similarity or FAISS.
3. **Generate:** Provide retrieved chunks as context to the LLM along with the original question.

Benefits: *Up-to-date knowledge, source attribution, reduced hallucination, no need to retrain for new data.*
Limitations: *Retrieval quality bottleneck, context window limits, may not synthesize across documents well.*

6.5 Process Reward Models (PRMs)

Standard reward models evaluate the final answer (Outcome Reward Model / ORM). Process Reward Models evaluate **each reasoning step** for correctness. This provides denser feedback and can guide the model to find correct reasoning paths even when the final answer alone is ambiguous.

PRMs combined with tree search (beam search over reasoning steps) show significant improvements on competition-level math (MATH, AMC). This is the basis of OpenAI's "o1" model reasoning approach.

6.6 Test-Time Compute Scaling

Traditional scaling: train bigger models. Newer insight: you can also improve performance by using **more computation at inference time** (test-time compute). Methods:

- Generate multiple solutions and verify/vote among them.
- Search through a tree of reasoning steps using a verifier.
- Use iterative refinement — generate, critique, and improve.

OpenAI's o1 model trains the model to use "thinking tokens" — extended internal reasoning before answering. More thinking tokens = better performance, at the cost of latency and compute.

6.7 Common Reasoning Failures

Understanding failure modes helps in designing better systems:

- **Hallucination:** Confidently stating false information. LLMs generate plausible-sounding text that may be factually wrong.
- **Sycophancy:** Agreeing with the user even when wrong. Models learn to please evaluators rather than be truthful.
- **Reversal Curse:** Models know "A is B" but fail at "B is A" — poor bidirectional factual recall.
- **Compositional Failures:** Can answer A and B separately but fail at "A and B combined."

LECTURE 7

Agentic LLMs

Autonomous AI — Planning, Memory, and Multi-Agent Systems

7.1 What Is an LLM Agent?

An **LLM agent** is an LLM that can take actions in an environment, observe results, and continue acting toward a goal. Unlike a simple chatbot that responds once, an agent operates in a loop: **Perceive** → **Think** → **Act** → **Observe** → **Repeat**.

Key components of an agent system:

- **LLM Brain:** The core model that does reasoning and decision-making.
- **Memory:** Short-term (context window) and long-term (vector store, database).
- **Tools:** Web search, code execution, file access, APIs.
- **Planning:** Breaking complex goals into subtasks.
- **Action Space:** What the agent can do — browse web, write code, send emails, etc.

7.2 Agent Architectures

ReAct (Reason + Act):

The model alternates between reasoning steps (Thought: ...) and action steps (Action: ...). Observations from actions are fed back into the context. Simple, effective, widely used.

Plan-and-Execute:

Separate planning and execution phases. A planner LLM generates a high-level plan (list of subtasks). An executor LLM carries out each subtask. Better for complex, multi-step tasks where upfront planning helps.

Reflexion:

After failing a task, the agent reflects on what went wrong and stores the reflection in memory. Future attempts use this reflection to avoid the same mistakes. A form of learning without weight updates.

Tree of Thoughts (ToT):

Instead of a single reasoning chain, explore multiple reasoning branches simultaneously. Use a value function to evaluate which branches are most promising and prune poor ones. Especially effective for tasks where you need to explore many possibilities.

7.3 Memory Systems for Agents

Sensory Memory	Raw perceptions — images, text, audio currently being processed.
Working Memory (Context)	The context window — what the model can currently "see" and reason about. Limited to thousands of tokens.

Episodic Memory	Past experiences and interactions, stored in a vector database and retrieved as needed.
Semantic Memory	General facts and knowledge — contained in the model's weights from pre-training.
Procedural Memory	Skills and how to do things — also in weights; also learned from demonstrations.

7.4 Multi-Agent Systems

Instead of one agent trying to do everything, multiple specialized agents collaborate. This allows parallelization, specialization, and checking each other's work.

Common Patterns:

- **Supervisor-Worker:** One orchestrator agent breaks tasks and delegates to specialized workers (coder, researcher, critic, etc.).
- **Debate / Critique:** Multiple agents generate solutions; others critique them; consensus or best answer selected.
- **Assembly Line:** Sequential pipeline where each agent processes and passes results to the next.
- **Peer-to-Peer:** Agents communicate directly as needed without central orchestration.

> ***Example — Software Development:*** *AutoGPT-style systems have a PM agent writing specs, a coder agent writing code, a tester agent running tests, and a debugger agent fixing errors.*

7.5 Tool Use and Function Calling

Modern LLMs support structured **function calling**: the model outputs a structured JSON object specifying which tool to call and with what arguments, rather than free-form text. This makes it easy to integrate LLMs with external systems reliably.

```
Example function call output: {"function": "search_web", "arguments": {"query": "latest AI research 2025"}}
```

The application executes the function, gets the result, and feeds it back to the LLM. This pattern is the foundation of modern AI assistants, coding assistants, and productivity tools.

7.6 Challenges in Agentic AI

Building reliable agents is hard. Key challenges:

- **Compounding Errors:** Mistakes in early steps cascade through long tasks. Error rate per step compounds multiplicatively.
- **Infinite Loops:** Agents can get stuck in repetitive cycles. Need loop detection and forced termination.
- **Context Overload:** Long agent trajectories fill the context window with observations, leaving less room for reasoning.
- **Trust and Safety:** Agents with tool access can do irreversible damage — delete files, send emails, make purchases.
- **Evaluation:** Hard to evaluate open-ended agent performance automatically.

Safety Principle: Agents should operate with **minimal necessary permissions**, confirm before irreversible actions, and maintain human oversight for high-stakes decisions.

LECTURE 8

LLM Evaluation

How Do We Know If an LLM Is Good?

8.1 Why Evaluation Is Hard

Evaluating LLMs is fundamentally different from traditional ML evaluation. There's no single "accuracy" metric. Outputs are open-ended natural language. What makes a response "good"? Correctness? Helpfulness? Safety? Style? Different tasks need different evaluation approaches.

8.2 Automated Benchmarks

Automated benchmarks allow fast, reproducible evaluation. Most use multiple-choice or short-answer formats where answers can be checked programmatically.

Key Academic Benchmarks:

MMLU	Massive Multitask Language Understanding — 57 subjects from elementary to expert level. Multiple choice. Tests broad knowledge.
HumanEval / MBPP	Code generation — write Python functions from docstrings. Tests programming ability.
GSM8K	Grade school math word problems. Tests multi-step mathematical reasoning.
MATH	Competition math problems. Much harder than GSM8K — tests advanced mathematical reasoning.
ARC Challenge	Grade-school science questions. Tests scientific reasoning.
HellaSwag	Commonsense completion — which sentence comes next? Tests world knowledge.
TruthfulQA	Questions where humans often have misconceptions. Tests honesty and knowledge vs. sycophancy.
BIG-Bench	Beyond the Imitation Game — 200+ diverse tasks designed to challenge and identify model capabilities.

8.3 The Benchmark Contamination Problem

A major concern: if benchmark test data appears in the training set, the model memorizes answers rather than demonstrating genuine capability. This is called **data contamination**. As major benchmarks become saturated (near-human performance), new, harder benchmarks are constantly needed.

> **Example:** MMLU went from <50% (human-level ~89%) to models scoring 90%+ in just a few years. The benchmark is now saturated. New benchmarks like GPQA (PhD-level), FrontierMath, and ARC-AGI are emerging to test frontier capabilities.

8.4 Human Evaluation

For open-ended generation (chatbots, creative writing), automated metrics fail. Human evaluators are the gold standard but expensive and slow.

Evaluation Dimensions:

- **Helpfulness:** Does the response actually help with the task?
- **Accuracy:** Is the information factually correct?
- **Harmlessness:** Is the response safe and non-offensive?
- **Coherence:** Is the response well-structured and logical?
- **Fluency:** Is the language natural and grammatically correct?

Common Protocol: Pairwise comparisons — show evaluators two model responses, ask which is better. More reliable than absolute ratings.

8.5 LLM-as-Judge

Use a capable LLM (like GPT-4 or Claude) to evaluate other LLMs' outputs. Much cheaper than human evaluation, can scale to large datasets, and shows high correlation with human judgments when done carefully.

Popular Benchmarks Using LLM-as-Judge:

- **MT-Bench:** 80 multi-turn conversation questions, judged by GPT-4.
- **AlpacaEval:** Compares model outputs against reference answers; uses GPT-4 as judge. Reports "win rate."
- **Arena (LMSYS Chatbot Arena):** Real users chat with two anonymous models, vote for the better one. Generates an ELO leaderboard.

Limitations of LLM-as-Judge: *Biases toward verbose responses, biases toward its own outputs (self-preference bias), biases toward stylistically similar text. Position bias — prefers the response shown first.*

8.6 Safety and Alignment Evaluation

Evaluating whether a model is safe and well-aligned requires specialized approaches:

- **Red-Teaming:** Humans (or automated systems) try to elicit harmful, dangerous, or unethical outputs. Jailbreaks, prompt injections, adversarial prompts.
- **Automated Red-Teaming:** Use another LLM to generate adversarial prompts at scale.
- **Refusal Rate:** What percentage of clearly harmful requests does the model refuse?
- **False Refusal Rate:** What percentage of harmless requests does it incorrectly refuse? (Over-refusal is also a problem.)
- **Stereotype / Bias Tests:** Does the model treat different groups differently? Uses templates like WinoBias.

8.7 Metrics for Specific Tasks

BLEU / ROUGE	N-gram overlap metrics for translation and summarization. Fast but poorly correlate with human judgments on modern models.
BERTScore	Semantic similarity using BERT embeddings. Better than n-gram overlap for semantic correctness.

Pass@k	For code: what fraction of problems does the model solve if given k tries? Standard for code benchmarks.
Exact Match (EM)	For QA: does the model output exactly match the reference answer?
F1 Score	Token overlap between prediction and reference, handling partial matches.
Perplexity (PPL)	How surprised is the model by test text? Lower = better language modeling. Not useful for instruction following.

LECTURE 9

Recap & Current Trends

The State of LLMs and Where the Field Is Heading

9.1 The LLM Timeline: A Quick Recap

The field has moved incredibly fast. Key milestones:

2017	"Attention Is All You Need" paper — Transformer invented at Google.
2018	BERT (Google) and GPT-1 (OpenAI) — pre-training paradigm established.
2019	GPT-2 (1.5B) — OpenAI initially withheld due to misuse concerns. RoBERTa, XLNet.
2020	GPT-3 (175B) — in-context learning discovered. T5, Scaling Laws paper.
2021	Codex (code generation), FLAN (instruction tuning), DALL-E for images.
2022	InstructGPT / ChatGPT — RLHF. Chinchilla scaling laws. Stable Diffusion.
2023	GPT-4, Llama, Mistral, Claude — open models emerge. RAG popularized. DPO.
2024	Llama 3, Gemini, Claude 3, Mixtral — MoE. Reasoning models (o1). Multimodal.
2025	Long context (1M+ tokens), vision-language, agent systems, test-time compute scaling.

9.2 Multimodal LLMs

Modern frontier models are not just text — they understand and generate **images, audio, video, and code**. The key insight: different modalities can be projected into the same vector space and processed by the same Transformer architecture.

- **Vision-Language Models:** LLaVA, GPT-4V, Claude Sonnet — understand images and answer questions about them.
- **Text-to-Image:** DALL-E, Stable Diffusion, Midjourney — generate images from text descriptions.
- **Speech:** Whisper (ASR), voice LLMs — understand and generate spoken language.
- **Video:** Sora, Gemini — understand and generate video content.

Technical approach: encode each modality into tokens using modality-specific encoders (e.g., vision encoder for images), then concatenate with text tokens and process with the LLM.

9.3 Mixture of Experts (MoE)

A key efficiency innovation: instead of all parameters being active for every token, **route each token to a subset of "expert" feed-forward networks**. Only 1-8 experts out of many (e.g., 64) are activated per token.

This allows massive model capacity with manageable compute per token.

> A 70B parameter dense model and a 70B parameter MoE model can have very different effective compute costs. Mixtral 8x7B has 47B total parameters but only activates 13B per token — matches a 13B dense model's compute cost while having 47B of capacity.

Key challenge: load balancing — ensuring all experts get used, not just a few "popular" ones. Auxiliary loss functions encourage balanced routing.

9.4 Long Context and Memory

Context windows have grown dramatically: GPT-3 (4K) → Claude (200K) → Gemini 1.5 (1M+). However, longer context introduces challenges:

- **"Lost in the Middle" Problem:** Models struggle to retrieve information from the middle of long contexts. Performance is best at the beginning and end.
- **Computational Cost:** Attention is $O(n^2)$ — 10× more context = 100× more attention compute.
- **Solutions:** FlashAttention, sparse attention, sliding window attention, retrieval augmentation for very long documents.

9.5 Open Source vs. Closed Models

A major trend: open-source models rapidly closing the gap with closed frontier models.

Closed Models	GPT-4, Claude, Gemini — proprietary weights, API access only. Maximum capability, safety testing.
Open Models	Llama 3, Mistral, Falcon, Qwen — weights publicly available. Can be fine-tuned, run locally.
Current Gap	As of 2025, best open models (Llama 3 70B) are close to but still behind frontier closed models on most benchmarks.
Trend	Gap narrowing rapidly. Open models enable democratized research and privacy-preserving local deployment.

9.6 Reasoning Models and o1/o3-Style Training

A new paradigm: instead of training models to give quick answers, train them to **think extensively before answering**. OpenAI's o1 model showed that a model trained to use long chains of internal "thinking" before answering dramatically outperforms standard models on hard reasoning tasks.

Key ideas:

- Process supervision — reward correct intermediate steps, not just final answers.
- MCTS / tree search during training data generation.
- Test-time compute scaling — more thinking = better answers.
- Self-play and verifiable rewards for math/code.

> This represents a shift from "scaling training compute" to "scaling inference compute" as the path to better AI.

9.7 AI Safety and Alignment — Open Problems

As models become more capable, alignment becomes more critical. Key open problems:

- **Scalable Oversight:** How do humans verify AI outputs when AI exceeds human expertise? Debate, recursive reward modeling, AI-assisted evaluation.
- **Interpretability:** What is the model actually doing internally? Mechanistic interpretability studies circuits and features inside Transformers (Anthropic, DeepMind research).
- **Robustness:** Models can be jailbroken with clever prompts. Need more robust safety training.
- **Value Alignment:** Whose values should the model embody? How to handle cultural differences and diverse preferences?
- **Deceptive Alignment:** Could a capable model appear aligned during training but behave differently in deployment?

9.8 The Near-Term Frontier

Key directions in active research and development:

- **Continual Learning:** Models that update their knowledge over time without full retraining.
- **World Models:** LLMs that understand physical causality, not just language patterns.
- **Personalization:** Models that adapt to individual users over time.
- **Efficient Inference:** Quantization (4-bit, 2-bit), speculative decoding, distillation — making models faster and cheaper.
- **AI Agents at Scale:** Autonomous agents completing real-world tasks with minimal human supervision.
- **Scientific Discovery:** Using LLMs and AI agents to accelerate research in biology, chemistry, materials science.

LECTURE ★

Quick Reference Cheatsheet

Key Terms and Concepts at a Glance

Essential Glossary

Attention	Mechanism allowing each token to weigh the importance of all other tokens. Core of the Transformer.
Auto-regressive	Generating one token at a time, each conditioned on all previous tokens.
Base Model	A model trained only on next-token prediction. No instruction following. Starting point for fine-tuning.
BPE	Byte-Pair Encoding — tokenization by iteratively merging frequent character pairs into subword units.
CoT	Chain-of-Thought — prompting strategy to elicit step-by-step reasoning.
DPO	Direct Preference Optimization — simpler alternative to RLHF; trains directly on preference pairs.
Embedding	Dense vector representation of a token in high-dimensional space.
Few-Shot	Providing a few input-output examples in the prompt to guide the model.
Fine-Tuning	Continuing training of a pre-trained model on task-specific data.
Hallucination	Confident generation of factually incorrect information.
RLHF	Reinforcement Learning from Human Feedback — training LLMs to match human preferences.
KV Cache	Cached key-value vectors from previous tokens, speeding up autoregressive generation.
LoRA	Low-Rank Adaptation — PEFT method adding small trainable matrices to frozen weights.
MoE	Mixture of Experts — sparse model routing tokens to a subset of expert networks.
PPO	Proximal Policy Optimization — RL algorithm used in RLHF for stable policy updates.
Perplexity	Measure of how well a language model predicts a text sample. Lower = better.
Prompt	Input text provided to an LLM to guide its output.
RAG	Retrieval-Augmented Generation — enhancing LLM with retrieved external knowledge.
SFT	Supervised Fine-Tuning — training on curated (instruction, response) pairs.
Temperature	Sampling parameter controlling output randomness. Higher = more creative/random.

Tokenization	Process of converting raw text into numerical tokens for model input.
Zero-Shot	Asking the model to perform a task with no examples in the prompt.

The LLM Pipeline at a Glance

Stage	What Happens	Key Techniques
1. Data	Collect & clean massive text datasets	Web crawl, dedup, quality filter
2. Tokenize	Convert text to integer tokens	BPE, WordPiece, SentencePiece
3. Pre-train	Train on next-token prediction	Transformer, AdamW, mixed precision
4. SFT	Fine-tune on instruction-response pairs	Supervised learning, curated datasets
5. Alignment	Train to match human preferences	RLHF, DPO, Constitutional AI
6. Evaluate	Test capabilities and safety	Benchmarks, human eval, LLM-as-judge
7. Deploy	Serve to users efficiently	Quantization, KV cache, batching
8. Monitor	Track performance and safety post-launch	Red-teaming, user feedback, logging

These notes cover Stanford's 9-lecture LLM course. For deeper understanding, refer to the original lecture slides, the papers cited within (especially "Attention Is All You Need," "Language Models are Few-Shot Learners," and "Training Language Models to Follow Instructions"), and hands-on practice with open-source models like Llama via Hugging Face.