PHASE 4

Sequence Models, NLP & Transformers

From RNNs to attention to LLMs, tokenization, fine-tuning, and RAG

Phase goal

Master the architecture behind modern AI. Build a Transformer from first principles, understand tokenization (including its CJK quirks), and stand up a retrieval-augmented generation system. This is the highest-leverage phase in the curriculum.

Estimated time: 6–8 weeks | **Track:** Comprehensive ML & Deep Learning Curriculum

A self-paced, depth-first path from mathematical foundations to production deep learning.

Contents

1 Why this is the highest-leverage phase

The Transformer is the single most important architecture in modern machine learning. It powers large language models, underpins modern vision (ViT, Phase 3), audio, code, and multimodal systems, and is where the overwhelming majority of current research and commercial value concentrates. If your time is limited, this phase plus Phase 7 (deployment) is the highest-return path. We build up to it through classical sequence models so the design choices feel inevitable rather than arbitrary.

2 Classical sequence models

2.1 RNNs and the vanishing-gradient problem

A recurrent network processes a sequence one step at a time, maintaining a hidden state $\mathbf{h}_t$ that summarizes the past:

$$\mathbf{h}_t = \tanh(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{xh}\mathbf{x}_t + \mathbf{b}).$$

Backpropagation through time multiplies many Jacobians together; if their norms are < 1 the gradient vanishes (the network forgets long-range context), if > 1 it explodes. This is the central limitation of vanilla RNNs.

Intuition

The vanishing-gradient problem means a plain RNN struggles to connect "The *cat* … [50 words] … *was* hungry" — by the time it reaches "was," the signal from "cat" has decayed. Every later innovation (gates, attention) is fundamentally about preserving information across long distances.

2.2 LSTMs and GRUs

LSTMs add a **cell state** and multiplicative **gates** (input, forget, output) that let the network learn what to keep, forget, and expose — creating a gradient highway across time (the same "additive path" insight as ResNet's skip connection). GRUs are a simpler two-gate variant. These dominated NLP from roughly 2014–2017.

2.3 Sequence-to-sequence and the bottleneck

Encoder–decoder (seq2seq) models compress an input sequence into a single fixed vector, then decode the output (machine translation, summarization). The flaw: cramming an entire sentence into one vector is a bottleneck. The fix — letting the decoder *look back* at all encoder states — is **attention**, and it changed everything.

3 Attention & Transformers

3.1 The attention mechanism

Attention lets a model, for each position, retrieve a weighted combination of *all* positions based on relevance. Cast it as a soft dictionary lookup with **queries**, **keys**, and **values**.

Definition

Scaled dot-product attention. Given queries $\mathbf{Q}$, keys $\mathbf{K}$, values $\mathbf{V}$ (rows are tokens, dimension d_k):

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

$\mathbf{QK}^\top$ scores every query against every key; the $\sqrt{d_k}$ scaling keeps the softmax in a sensible range; softmax turns scores into weights that sum to 1; multiplying by $\mathbf{V}$ returns a weighted blend of values.

Intuition

For each word, attention asks "which other words should I pay attention to?" and builds a context-aware representation accordingly. "It" in "the trophy didn't fit in the suitcase because *it* was too big" can attend to "trophy." The scaling by $\sqrt{d_k}$ matters: without it, large dot products push softmax into saturation where gradients vanish.

3.2 Self-attention and multi-head attention

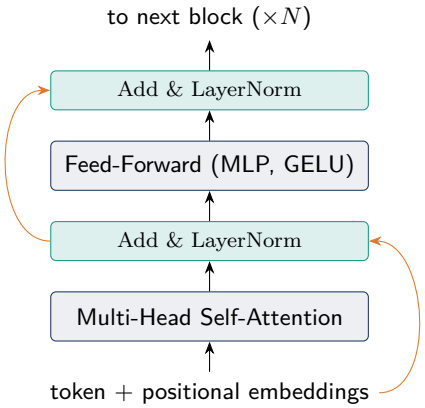
In **self-attention**, $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ all come from the same sequence (each token attends to the others). **Multi-head** attention runs several attention operations in parallel with different learned projections, then concatenates them — so different heads can specialize (one tracks syntax, another coreference, another position).

Key idea

Self-attention's key advantages over RNNs: (1) it connects any two positions in *one* step (no long-range decay), and (2) it is fully *parallel* across positions (no sequential dependency), which is what made training on internet-scale data feasible. The cost is $O(n^2)$ in sequence length n — the motivation behind FlashAttention and long-context research.

3.3 The Transformer, end to end

The original "Attention Is All You Need" (2017) architecture stacks identical blocks. Each block has two sublayers — multi-head self-attention and a position-wise feed-forward network — each wrapped with a **residual connection** and **layer normalization**.



Positional encoding. Because attention is permutation-invariant (it has no inherent notion of order), we inject position information — via fixed sinusoids (original) or learned/rotary embeddings (RoPE, now standard in LLMs). **LayerNorm placement** matters: modern LLMs use *pre-norm* (normalize before each sublayer) for more stable deep training.

```
import torch, torch.nn.functional as F
def scaled_dot_product_attention(Q, K, V, mask=None):
    d_k = Q.size(-1)
    scores = (Q @ K.transpose(-2, -1)) / d_k**0.5      # (.., n, n)
```

```
    if mask is not None:                                    # causal/padding mask
        scores = scores.masked_fill(mask == 0, float("-inf"))
    weights = F.softmax(scores, dim=-1)
    return weights @ V                                      # (.., n, d_v)
```

4 Modern NLP & LLMs

4.1 Tokenization

Models operate on integer token IDs, not raw text. Subword tokenizers strike a balance between character-level (long sequences) and word-level (huge vocab, out-of-vocabulary words):

- **BPE** (Byte-Pair Encoding): iteratively merge the most frequent symbol pair. Used by GPT.
- **WordPiece**: similar, merges by likelihood. Used by BERT.
- **SentencePiece / Unigram**: language-agnostic, operates on raw bytes/characters — no pre-tokenization by spaces.

CJK tokenization has real quirks

English is conveniently delimited by spaces; Chinese is not. The sentence 我喜欢机器学习 ("I like machine learning") has no spaces, so a tokenizer cannot rely on whitespace to find word boundaries. Practical consequences for your bilingual document work:

- Most modern LLM tokenizers are **byte-level BPE**, so they handle Chinese, but often split a single character into multiple byte-tokens — inflating token counts (and API cost) for CJK text relative to English.
- A word like 机器学习 ("machine learning") may become several tokens; segmentation is implicit and learned, not linguistic.
- Always inspect how *your* tokenizer treats your corpus: token counts, whether rare characters round-trip, and whether mixed Chinese/English/code text fragments cleanly.

Inspecting tokenization

Tokenize the Chinese phrase 机器学习很有趣 ("machine learning is fun") and its English equivalent, then compare how each is split:

```
from transformers import AutoTokenizer
tok = AutoTokenizer.from_pretrained("bert-base-multilingual-cased")
zh = "..."   # the Chinese phrase above
print(tok.tokenize(zh))                    # CJK: roughly per-character pieces
print(tok.tokenize("machine learning"))  # English: subword pieces
# Compare token COUNTS across languages for the same meaning -> cost/latency.
```

The Chinese string typically yields about one token per character (six tokens here), while a byte-level BPE may emit several byte-tokens *per* character. Run this on your own Chinese↔English documents and compare token counts; it directly affects context-window budgeting and API spend.

4.2 Pretraining objectives

- **Masked LM (BERT-style, encoder):** mask random tokens and predict them from both sides — bidirectional context, great for *understanding* tasks (classification, NER, embeddings).

- **Autoregressive LM (GPT-style, decoder):** predict the next token from the left context only (causal mask) — great for *generation*. This is the basis of modern chat LLMs.

4.3 Adapting models: fine-tune vs. prompt vs. PEFT

Approach	When to use
Prompting / in-context	zero/few-shot; no training; fastest to try; great baseline
Full fine-tuning	lots of task data, need maximum quality, have the compute
PEFT (LoRA/QLoRA)	adapt a large model cheaply by training small low-rank adapters

LoRA / QLoRA

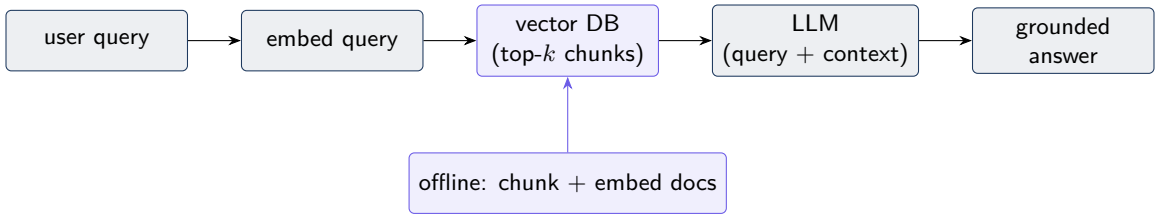
LoRA freezes the pretrained weights and learns a low-rank update $\Delta\mathbf{W} = \mathbf{BA}$ (with $\mathbf{A}, \mathbf{B}$ tiny) for each weight matrix — training $<1\%$ of the parameters while nearly matching full fine-tuning. **QLoRA** adds 4-bit quantization of the frozen base, letting you fine-tune a multi-billion-parameter model on a single consumer GPU. This is the dominant practical way to specialize LLMs today, and it is a direct application of the low-rank idea you met via SVD in Phase 0.

4.4 Embeddings & vector search

An embedding maps text to a dense vector where semantic similarity is geometric proximity (cosine similarity — the Phase 0 dot product again). Embeddings power semantic search, clustering, deduplication, recommendation, and retrieval. Store them in a vector index (FAISS, Chroma, pgvector) and query by nearest neighbors — approximate (ANN) algorithms like HNSW make this fast at scale.

5 Retrieval-Augmented Generation (RAG)

LLMs hallucinate and have a knowledge cutoff. RAG grounds them in your data: retrieve relevant chunks, then have the model answer *using* that context.



Intuition

RAG = open-book exam for an LLM. The quality ceiling is set by *retrieval*, not the model: if the right chunk is not retrieved, no amount of prompting saves you. Most "my RAG is bad" problems are retrieval problems — chunking strategy, embedding quality, and ranking — so instrument and evaluate retrieval separately from generation.

Common pitfall

Common RAG failure modes: chunks too large (dilute relevance) or too small (lose context); no metadata filtering; embedding the question and documents with mismatched models; and no reranking. Add a cross-encoder reranker, tune chunk size/overlap, and consider hybrid (keyword + vector) search. Evaluate with retrieval metrics (recall@k) and answer-faithfulness checks.

5.1 Evaluating generative models

Generation is hard to score. Classic n-gram metrics (BLEU, ROUGE) are weak proxies. Modern practice combines: task-specific metrics, *LLM-as-judge* (with care for bias), human evaluation on a curated set, and for RAG, **faithfulness/groundedness** and **answer relevance** (e.g. the RAGAS framework). Always keep a small, hand-built "golden" eval set.

6 The Hugging Face ecosystem

- **transformers**: thousands of pretrained models behind one API (`AutoModel`, `AutoTokenizer`, `Trainer`, `pipeline`).
- **datasets**: streaming, memory-mapped datasets; easy preprocessing.
- **tokenizers**: fast Rust-backed tokenizers; train your own.
- **PEFT** & **bitsandbytes**: LoRA/QLoRA and quantization.
- Deployment quantization: **GGUF** (llama.cpp, CPU/edge), **bitsandbytes/GPTQ/AWQ** (GPU).

```
from transformers import pipeline
clf = pipeline("sentiment-analysis")          # downloads a pretrained model
print(clf("This curriculum is fantastic!"))    # [{'label': 'POSITIVE', ...}]

# Fine-tuning skeleton with the Trainer API:
from transformers import AutoModelForSequenceClassification, TrainingArguments, Trainer
model = AutoModelForSequenceClassification.from_pretrained("bert-base-uncased", num_labels=2)
args = TrainingArguments(output_dir="out", eval_strategy="epoch",
                         per_device_train_batch_size=16, num_train_epochs=3)
trainer = Trainer(model=model, args=args, train_dataset=ds_train, eval_dataset=ds_val)
trainer.train()
```

7 Checkpoint project

Build a RAG pipeline over your own docs

Goal: a question-answering system grounded in a document set you care about (e.g. your own technical/API docs) — a natural fit for your backend skill set.

1. **Ingest:** load documents, chunk them (experiment with size/overlap), attach metadata.
2. **Embed & index:** embed chunks with a sentence-embedding model; store in FAISS, Chroma, or **pgvector** (the latter slots neatly into an existing Postgres backend).

3. **Retrieve:** top-k nearest neighbors for a query; add a reranker and optional hybrid keyword search.
4. **Generate:** construct a prompt with the retrieved context and call an LLM; cite sources in the answer.
5. **Serve:** expose a `/ask` REST endpoint (FastAPI); return the answer plus the source chunks.
6. **Evaluate:** build a small golden Q&A set; measure retrieval recall@k and answer faithfulness; iterate on chunking and retrieval.

Stretch: make it bilingual — test Chinese and English queries over mixed-language docs and compare retrieval quality and token costs (ties back to the tokenization section).

Phase 4

- **Stanford CS224n** — NLP with Deep Learning (the definitive course).
- **Andrej Karpathy** — "Let's build GPT" (builds a Transformer from scratch; pair with his earlier backprop video).
- **Hugging Face** free NLP course (huggingface.co/learn).
- **Jay Alammar** — "The Illustrated Transformer" (the clearest visual explainer).
- Papers: *Attention Is All You Need*; *BERT*; *GPT-1/2/3*; *LoRA*.

8 Phase 4 self-check

- Explain the vanishing-gradient problem and how LSTMs and attention each address it.
- Write scaled dot-product attention from memory and explain every term, including $\sqrt{d_k}$.
- Describe the full Transformer block and why residual connections and positional encodings are needed.
- Contrast encoder (BERT) vs. decoder (GPT) pretraining and when to use each.
- Explain BPE/WordPiece/SentencePiece and the specific quirks of CJK tokenization.
- Explain LoRA/QLoRA and why low-rank updates work.
- Draw the RAG architecture and name the most common failure mode.