LLMs

A Complete 11-Chapter Data Science & ML Course

Topics covered:

Subword tokenization, embeddings & positional encoding, self-attention formalized

The full Transformer block, GPT/BERT/T5 architectures, pretraining & scaling laws

Fine-tuning, RLHF, context windows, and a full end-to-end prompt trace

Exercises: 33 hands-on scenarios with worked solutions

Format: A4 · Dark-theme code examples

Course 5 of 6 in the Data Science & ML subject

Philip Osztromok · Generated with Claude

Table of Contents

CHAPTER 1 OF 11

What an LLM Actually Is (Not Another Prompting Course)

LLMs

Chapter 1 · What an LLM Actually Is (Not Another Prompting Course)

This course exists to answer one question the site has never actually answered: what is happening, mechanically, inside the model between a prompt going in and a response coming out? Not how to phrase the prompt well — that's already covered, thoroughly, elsewhere.

What This Course Is Not

COURSE	WHAT IT ACTUALLY TEACHES
prompt1	How to phrase a prompt well — clarity, context, constraints, few-shot examples, chain-of-thought
claude-adv1	How to build with the Claude API — tool use, streaming, RAG, multi-agent patterns
llm1 (this course)	What's actually happening inside the model — tokenization, attention, pretraining, fine-tuning, why any of the above techniques work at all

`prompt1` and `claude-adv1` already own the practical territory in full — this course doesn't re-teach either one. It sits underneath both, the same way `imgai1` sat underneath image-generation prompting: that course didn't teach how to phrase an image prompt well either — it explained diffusion, the actual mechanism a prompt gets fed into. This course is the direct textual analogue.

Delivering on nlp1-10's Own Three Claims

`nlp1-10`'s own capstone closed the NLP course by naming three real differences between its own hand-built pipelines and an LLM — **scale**, **self-supervised pretraining taken further**, and **one flexible architecture instead of many task-specific pipelines**. That chapter asserted all three honestly, but didn't have the room to prove any of them mechanically. This course does exactly that, one claim at a time.

WHERE EACH CLAIM GETS PROVEN

- **Scale** — `llm1-7` (real pretraining data/parameter counts) and `llm1-8` (scaling laws — why bigger follows a predictable curve, not a guess)

- **Self-supervised pretraining taken further** — `llm1-7`, extending `nlp1-5`'s and `nlp1-9`'s own context-prediction training signal to an entire corpus, predicting the next token instead of a single missing word
- **One flexible architecture vs. many pipelines** — `llm1-6`, explaining why a single decoder-only Transformer can be steered toward tasks `nlp1-6` and `nlp1-7` each needed a separately built, separately trained pipeline to perform

Unpacking the Name — Large Language Model

- **Large** — a real, measurable claim about parameter count and training data volume, not marketing language. `llm1-8` covers exactly what "large" buys you and why.
- **Language** — trained primarily on text (many current models are genuinely multimodal too, an honest scope note this course doesn't chase further).
- **Model** — a function that computes a probability distribution over what token comes next, not a database. This is a genuine, important contrast with `nlp1-9`'s own GloVe: GloVe is a fixed lookup table — look up a word, get back a static vector, nothing more. An LLM is a function with billions of learned parameters that computes a fresh answer for every input; nothing is looked up, everything is computed.

A MODEL, NOT A LOOKUP TABLE

This distinction matters for the rest of the course: `nlp1-9`'s own out-of-vocabulary problem existed specifically because GloVe's vocabulary was a fixed table with gaps. A computed function has no such fixed table to run out of — `llm1-2` covers exactly how tokenization closes that gap for good.

A First, Deliberately Incomplete Preview

Take the prompt `"The capital of France is"`. At a high level, without yet explaining any of the mechanics: the text is broken into pieces (`llm1-2`), each piece is converted into a vector that also encodes its position in the sequence (`llm1-3`), those vectors pass through many stacked layers that let every piece attend to every other piece (`llm1-4` / `llm1-5`), and the model ultimately produces a probability distribution over every possible next piece — with `"Paris"` receiving, if training went well, the highest probability of all. This entire path is deliberately left abstract here — the capstone (`llm1-11`) walks it in full, once every piece has been built.

Why This Actually Helps With Prompting, Without Re-Teaching It

Knowing this mechanism doesn't replace `prompt1`'s own techniques — it explains why they work. Few-shot examples help because the model is a pattern-continuation engine at its core (`llm1-7` makes this precise). Chain-of-thought helps because generation is genuinely sequential — each token really does depend on

every token generated before it (`llm1-11` 's own trace makes this concrete). This course is a companion to `prompt1` , not a replacement for it.

Hands-On Exercises

EXERCISE 1

Explain the specific difference in scope between prompt1, claude-adv1, and this course, and explain why this course is described as the direct analogue of what imgai1 did for image-generation prompting.

View solution

EXERCISE 2

Explain why "Model" in "Large Language Model" is a meaningful, load-bearing word rather than filler, using the contrast between an LLM and nlp1-9's own GloVe lookup table to make the distinction concrete.

View solution

EXERCISE 3

For each of nlp1-10's own three named differences (scale, self-supervised pretraining taken further, one flexible architecture vs. many pipelines), identify which specific chapter of this course is responsible for proving it mechanically, and explain in one sentence what "proving it mechanically" means as opposed to simply asserting it.

View solution

CHAPTER 1 QUICK REFERENCE

- **This course's scope:** the mechanism underneath a prompt — not how to write one (prompt1) or how to build with an API (claude-adv1)
- **The imgai1 parallel:** that course explained diffusion underneath image prompting; this course explains the Transformer underneath text prompting
- **Model ≠ lookup table:** an LLM computes a fresh answer for every input, unlike nlp1-9's own fixed GloVe vectors
- **nlp1-10's three claims, and where each gets proven:** scale (llm1-7/8), self-supervised pretraining taken further (llm1-7), one flexible architecture vs. many pipelines (llm1-6)
- **Next chapter:** Subword Tokenization: Byte-Pair Encoding

CHAPTER 2 OF 11

Subword Tokenization: Byte-Pair Encoding

LLMs

Chapter 2 · Subword Tokenization: Byte-Pair Encoding

`nlp1-9` named a real, honest limitation of pretrained embeddings: a fixed vocabulary, with genuine gaps. This chapter shows how LLMs close that gap for good — not by building a bigger vocabulary, but by changing what the vocabulary is made of.

A Genuinely Different Philosophy From nlp1-1's Own Pipeline

`nlp1-1`'s preprocessing pipeline split text into whole words, stripped stopwords, and reduced words to their lemma — a linguistically-informed process, built around what a word "means." Byte-Pair Encoding (BPE) abandons that framing entirely. It doesn't ask what a word means — it asks what sequences of characters appear together often enough, across a massive corpus, to be worth their own token. No stopword removal, no lemmatization: raw frequency statistics decide what counts as a unit.

The BPE Algorithm

BPE started life as a data-compression algorithm and was repurposed for tokenization. The core loop:

1. Start with a vocabulary of individual characters (or bytes).
2. Count every adjacent pair of symbols across the training corpus.
3. Merge the single most frequent pair into one new symbol; add it to the vocabulary.
4. Repeat, thousands of times, until the vocabulary reaches a target size.

```
# simplified BPE training loop
vocab = set(all_characters_in_corpus)
corpus = split_into_characters(training_text)

for step in range(num_merges):
    pairs = count_adjacent_pairs(corpus)
    best_pair = max(pairs, key=pairs.get)     # the single most frequent pair
    vocab.add(merge(best_pair))               # e.g. ("t", "ion") -> "tion"
    corpus = apply_merge(corpus, best_pair)    # replace every occurrence
```

Common fragments like `"tion"`, `"ing"`, or `"un"` earn their own token early, since they occur constantly across a large corpus. Rare words never get merged into single tokens at all — they stay broken into smaller, more common pieces.

A Worked Example

INPUT WORD	LIKELY SUBWORD TOKENS
`tokenization`	`token` + `ization`
`unbelievability`	`un` + `believ` + `ability`
`zzyxplorb` (invented, never seen in training)	`zz` + `y` + `x` + `plor` + `b` — falls back toward individual characters

Resolving nlp1-9's Own Gap, For Good

NOT A BIGGER TABLE — A DIFFERENT KIND OF TABLE

`nlp1-9`'s own GloVe vectors were a fixed table of whole words: a word either has an entry or it doesn't, and a missing entry means no meaningful vector at all. BPE's vocabulary reaches all the way down to individual bytes as its ultimate fallback — so literally any string, including gibberish nobody has ever typed before, can always be decomposed into *some* sequence of known vocabulary pieces. Nothing is ever truly out of vocabulary, because the smallest unit in the vocabulary is small enough to represent anything.

Byte-Level BPE

GPT-2 and its successors operate on raw UTF-8 *bytes* rather than Unicode characters — a further guarantee. Since every possible string, in every script, emoji included, is representable as a sequence of bytes, byte-level BPE never encounters a character it fundamentally cannot represent, even before any merges are learned. This is what makes the "no true out-of-vocabulary input" guarantee absolute rather than merely "very good in practice."

Vocabulary Size — A Real Trade-Off

VOCABULARY SIZE	CONSEQUENCE
Too small	Text breaks into long sequences of tiny pieces — more tokens per sentence, more compute per input
Too large	Many tokens become rare and poorly trained, with diminishing returns on vocabulary growth

Real models settle in the tens of thousands to roughly one hundred thousand tokens — large enough that common words and word-fragments get their own single token, small enough that the vocabulary itself stays

learnable.

An Honest Limitation: Tokens Aren't Meaning Units

A REAL, WELL-KNOWN PRACTICAL QUIRK, EXPLAINED MECHANICALLY

Unlike `nlp1-1`'s own words or `nlp1-5`'s own word embeddings, a subword token like `"ing"` or `"plor"` carries no meaning of its own — it's a compression-driven engineering choice, not a linguistic one. This is the actual, mechanical reason LLMs are famously unreliable at tasks like counting letters within a word: the model frequently never "sees" the word as individual characters at all — it sees one or two opaque subword chunks, with no direct access to what letters compose them.

Hands-On Exercises

EXERCISE 1

Explain, using this chapter's own BPE training loop, why common fragments like "ing" or "tion" become their own tokens while rare words stay broken into smaller pieces.

View solution

EXERCISE 2

Explain specifically why BPE's own byte-level fallback resolves nlp1-9's out-of-vocabulary problem "for good" rather than just reducing how often it happens, contrasting BPE's vocabulary structure directly with GloVe's.

View solution

EXERCISE 3

Explain the mechanical reason LLMs are often unreliable at counting letters within a word, using this chapter's own explanation of what a subword token actually represents.

View solution

CHAPTER 2 QUICK REFERENCE

- **BPE** — repeatedly merge the most frequent adjacent symbol pair, building a vocabulary from the bottom up by frequency, not linguistics

- A genuinely different philosophy from nlp1-1's own word-level, linguistically-informed pipeline — no stopword removal, no lemmatization
- **The fallback:** vocabulary reaches down to individual bytes, so any string can always be represented — resolving nlp1-9's OOV gap for good, not just reducing it
- **Byte-level BPE** (GPT-2 onward) — operates on raw UTF-8 bytes, making the "no true OOV" guarantee absolute
- Vocabulary size is a real trade-off between sequence length and token rarity — real models use tens of thousands to ~100k tokens
- **Honest limitation:** subword tokens carry no inherent meaning, which is the real mechanical reason behind LLMs' own letter-counting unreliability
- **Next chapter:** Embeddings & Positional Encoding at Scale

CHAPTER 3 OF 11

Embeddings & Positional Encoding at Scale

LLMs

Chapter 3 · Embeddings & Positional Encoding at Scale

`nlp1-8` closed with an honest, deliberately unresolved gap: self-attention has no inherent sense of token order. This chapter delivers the fix in full — and along the way, revisits what an "embedding" even means once it's trained as part of one enormous model rather than as a separate step.

Token Embeddings — Trained End-to-End, Not Frozen

Every token ID produced by `llm1-2`'s own BPE vocabulary maps to a learned vector through an embedding lookup table — mechanically similar to `nlp1-5`'s own trainable `nn.Embedding`. The real difference is what trains it: `nlp1-9`'s own GloVe vectors were pretrained separately and then either frozen or fine-tuned as a distinct step. An LLM's own token embeddings are trained *jointly*, from the very start, alongside every other parameter in the entire network — there is no separate embedding-training phase at all.

APPROACH	HOW THE EMBEDDING TABLE IS TRAINED
nlp1-5's word2vec	Trained from scratch, as its own separate step, on a local corpus
nlp1-9's GloVe	Pretrained separately at scale, then used frozen or fine-tuned
An LLM's own embeddings	Trained jointly with the entire network, as one single end-to-end process

The Problem This Chapter Actually Solves

Per `nlp1-8`: swap two tokens in a sequence, and self-attention computes the exact same set of pairwise relationships either way. Nothing about the raw token embeddings, on their own, encodes *where* in the sequence a token appears. **Positional encoding** exists to inject that missing information before attention ever runs.

Sinusoidal Positional Encoding — The Original Fix

The original Transformer paper's own solution: a fixed, non-learned pattern of sine and cosine waves, one pair of frequencies per pair of embedding dimensions, added directly to each token's embedding based on its position.

```
PE(pos, 2i)   = sin(pos / 10000^(2i/d))
PE(pos, 2i+1) = cos(pos / 10000^(2i/d))

# pos = the token's position in the sequence (0, 1, 2, ...)
# i   = which pair of embedding dimensions this is
# d   = the total embedding dimension
```

Each position gets a unique, deterministic "fingerprint" — no two positions ever produce the same pattern. The specific choice of sine/cosine pairs has a genuinely elegant mathematical property: the encoding for any fixed relative offset (position *p*+*k* relative to position *p*) can be expressed as a simple linear transformation of the encoding at position *p*, which makes it easier for the model to learn to attend by *relative* position, not just absolute position.

Learned Positional Embeddings — GPT's Own Approach

GPT and many later models take a simpler route: instead of a fixed formula, learn a separate embedding vector for each position index (0 through some maximum), trained jointly with everything else, exactly like the token embeddings themselves.

APPROACH	TRADE-OFF
Sinusoidal (fixed)	No training required, well-behaved relative-offset structure — but not adapted to this model's own specific data
Learned	Adapts positional representations to the training data — but genuinely cannot represent a position beyond whatever maximum was seen during training

A REAL LIMITATION, REVISITED IN LLM1-10

A learned positional embedding table has a fixed size — it simply has no entry for position 8,193 if it was only ever trained up to position 8,192. This is one of the concrete mechanical reasons a context window has a hard limit, covered in full once `llm1-10` gets there.

Combining Token and Position

```
token_embedding = embedding_table[token_id]          # llm1-2's token, looked up
position_embedding = positional_encoding[position]    # sinusoidal or learned
input_vector = token_embedding + position_embedding    # what actually enters layer 1
```

Revisit `nlp1-4`'s own "dog bites man" vs. "man bites dog" example. Before any attention runs at all, `"dog"` at position 0 and `"dog"` at position 2 (in the reversed sentence) now receive genuinely different input

vectors, purely because their positional components differ — the exact information self-attention alone could never supply on its own, now supplied directly at the input.

Hands-On Exercises

EXERCISE 1

Explain the specific difference in how nlp1-5's word2vec, nlp1-9's GloVe, and an LLM's own token embeddings are each trained, and explain why "trained jointly with the entire network" is a genuinely different approach from the other two, not just a difference of scale.

View solution

EXERCISE 2

Using this chapter's own "dog bites man" revisit, explain exactly how positional encoding gives self-attention the order information nlp1-8 showed it structurally lacks, and explain at what point in the pipeline this information is introduced.

View solution

EXERCISE 3

Explain the trade-off between sinusoidal and learned positional encoding, and explain specifically why a learned positional embedding table has a hard maximum sequence length while a sinusoidal one, in principle, does not.

View solution

CHAPTER 3 QUICK REFERENCE

- **Token embeddings** — a lookup table like nlp1-5's own nn.Embedding, but trained jointly with the whole network rather than separately (nlp1-9's own frozen/fine-tuned GloVe)
- **The problem solved:** nlp1-8's own named gap — self-attention alone has no sense of token order
- **Sinusoidal positional encoding** — fixed sine/cosine pattern per position, with a useful relative-offset property
- **Learned positional embeddings** — GPT's own approach; adapts to the data but has a hard maximum position (revisited in llm1-10)
- **Combining:** input vector = token embedding + positional embedding, added before any attention runs
- **Next chapter:** Self-Attention, Formalized: Query, Key & Value

CHAPTER 4 OF 11

Self-Attention, Formalized: Query, Key & Value

LLMs

Chapter 4 · Self-Attention, Formalized: Query, Key & Value

`nlp1-8` introduced attention conceptually — a score, a softmax weight, a blended context vector. That was enough to explain *why* attention works. This chapter shows *how* it's actually computed, with real matrices.

Recap: nlp1-8's Own Conceptual Version

```
# nlp1-8's own conceptual attention score
scores = [dot(decoder_state, encoder_state_i) for encoder_state_i in encoder_states]
weights = softmax(scores)
context = sum(w * s for w, s in zip(weights, encoder_states))
```

That version compared one decoder state against a set of encoder states. Self-attention needs something more general: every token comparing itself against every other token in the *same* sequence, including itself. This is where Query, Key, and Value come in.

Query, Key, Value — What Each One Actually Is

For every input vector (from `llm1-3` — a token embedding plus its positional encoding), the model computes three separate vectors by multiplying it against three separate, learned weight matrices:

```
Q = X @ W_Q    # Query — "what am I looking for?"
K = X @ W_K    # Key   — "what do I contain, for others to match against?"
V = X @ W_V    # Value — "what do I actually offer, if attended to?"
```

VECTOR	ROLE, IN PLAIN TERMS
Query	Represents what this token is currently trying to find out from the rest of the sequence
Key	Represents what this token has to offer, in a form other tokens' Queries can be compared against
Value	Represents the actual content this token contributes once it's been attended to

Every single token produces all three — its own Query, its own Key, its own Value — using the exact same three weight matrices (`W_Q`, `W_K`, `W_V`), shared across the whole sequence and learned during training like every other parameter.

Scaled Dot-Product Attention

```
scores = (Q @ K.T) / sqrt(d_k)    # every Query compared against every Key
weights = softmax(scores, axis=-1)  # normalized per row — one weight distribution per token
output = weights @ V                 # weighted blend of every token's Value
```

`Q @ K.T` computes a similarity score between every token's Query and every other token's Key, in one matrix multiplication — nlp1-8's own single dot-product score, generalized to every pair at once. Dividing by `sqrt(d_k)` (the square root of the Key dimension) keeps the scores from growing too large as dimensionality increases, which would otherwise push softmax into regions with vanishingly small gradients. Softmax turns each token's own row of scores into a proper probability distribution — exactly `nlp1-8`'s own attention weights, generalized from one decoder step to every token simultaneously. The final multiplication by `V` blends every token's Value according to those weights, producing one output vector per token.

Why Three Separate Matrices, Not One

SEPARATING "WHAT I'M LOOKING FOR" FROM "WHAT I CONTAIN"

A single token plays two genuinely different roles in every attention computation — the thing doing the looking (its Query) and the thing being looked at (its Key and Value). If a token used the same vector for both roles, it would be forced to represent "what I'm searching for" and "what I actually contain" as the identical vector, which is a real, unnecessary constraint. Separate learned projections let the model discover different representations for each role.

Self-Attention, Specifically

This is called *self*-attention because Q, K, and V are all derived from the *same* sequence — every token attends to every token in its own input, itself included. This is the exact mechanism `nlp1-8` introduced conceptually as "applying attention within a single sequence," now written out as real matrix operations rather than described in prose.

Hands-On Exercises

EXERCISE 1

Explain, in your own words, what Query, Key, and Value each represent, and explain why using three separate learned weight matrices is necessary rather than reusing one matrix for all three.

View solution

EXERCISE 2

Map each piece of this chapter's own scaled dot-product attention formula (Q @ K.T, the division by sqrt(d_k), softmax, the final multiplication by V) back to the corresponding piece of nlp1-8's own conceptual score/weight/blend mechanism.

View solution

EXERCISE 3

Explain why dividing by sqrt(d_k) is necessary, and explain specifically what would go wrong with softmax if this scaling step were removed, particularly for larger embedding dimensions.

View solution

CHAPTER 4 QUICK REFERENCE

- **Query** — what a token is looking for · **Key** — what a token offers, for matching · **Value** — what a token actually contributes
- Q, K, V are computed via three separate learned weight matrices, shared across the sequence: `Q = X @ W_Q`, etc.
- **Scaled dot-product attention:** `softmax((Q @ K.T) / sqrt(d_k)) @ V` — nlp1-8's own score/weight/blend mechanism, generalized to every token pair at once
- The `sqrt(d_k)` scaling keeps softmax's own gradients from vanishing as dimensionality grows
- **Self-attention** — Q, K, V all come from the same sequence; every token attends to every token, including itself
- **Next chapter:** Multi-Head Attention & the Full Transformer Block

CHAPTER 5 OF 11

Multi-Head Attention & the Full Transformer Block

LLMs

Chapter 5 · Multi-Head Attention & the Full Transformer Block

`nn1-9` and `nlp1-8` each previewed self-attention and each explicitly deferred "the full multi-head, multi-layer Transformer" to this course. This is that chapter — the single largest deferred-technical-depth thread either course carried.

Why One Attention Computation Isn't Enough

`llm1-4`'s own self-attention computes one set of relationships per sequence — one Query, Key, and Value projection, one resulting pattern of attention weights. Real language has many kinds of relationships happening at once inside the same sentence: which word a pronoun refers to, which verb a subject belongs to, which adjective modifies which noun. Forcing a single attention computation to capture all of these simultaneously is a real constraint — exactly the kind `llm1-4`'s own finding-box already flagged for forcing Query and Key into one shared vector.

Multi-Head Attention

The fix: split the embedding dimension into several smaller **heads**, each with its own independent set of learned `W_Q` / `W_K` / `W_V` matrices, each computing `llm1-4`'s own scaled dot-product attention independently and in parallel.

```
heads = []
for i in range(num_heads):
    Q_i = X @ W_Q[i]      # each head gets its own learned projections
    K_i = X @ W_K[i]
    V_i = X @ W_V[i]
    heads.append(scaled_dot_product_attention(Q_i, K_i, V_i))

multi_head_output = concat(heads) @ W_O    # concatenate, then project back down
```

Each head is free to specialize — one might learn to track subject-verb agreement, another might learn coreference (which noun a pronoun points back to), another might attend mostly to nearby, local context.

Concatenating every head's own output and projecting it back through `W_O` combines all these specialized views into a single vector per token, richer than any one head could produce alone.

The Feed-Forward Sublayer

Attention's own job is mixing information *across* tokens — deciding what each token should attend to. After attention, every token's own resulting vector independently passes through a small two-layer network, applied identically and separately to each position:

```
FFN(x) = max(0, x @ W1 + b1) @ W2 + b2   # (or GELU instead of the ReLU shown here)
```

This is where genuine non-linear transformation of each token's own representation happens, independent of every other token — attention decides *what to look at*; the feed-forward layer decides *what to do with it*, position by position.

Residual Connections

Real Transformers stack dozens of these blocks — GPT-3 uses 96 layers. Very deep stacks risk exactly the kind of degraded gradient flow `nn1-6`'s own training material warned about for deep networks generally. The fix here: each sublayer's output is *added to* its own input, rather than replacing it outright.

```
x = x + MultiHeadAttention(x)
x = x + FeedForward(x)
```

> **A SHORTCUT FOR GRADIENTS**
>
> Because the original input `x` is always present in the sum, gradients during backpropagation have a direct path all the way back through the network via these shortcuts, rather than only flowing through however many transformative sublayers sit in between. Extending `nn1-6`'s own training-stability theme to networks dozens of layers deep.

Layer Normalization

Each sublayer's input is also normalized — rescaled to have stable mean and variance across its own feature dimension, per token — keeping training numerically well-behaved at depth. Most modern models apply this normalization *before* each sublayer (Pre-LN) rather than after (Post-LN, the original Transformer paper's own choice), since Pre-LN has proven more stable to train at very large depth.

Assembling One Full Transformer Block

```
def transformer_block(x):
    x = x + multi_head_attention(layer_norm(x))
    x = x + feed_forward(layer_norm(x))
    return x

for _ in range(num_layers):        # dozens of these, stacked
    x = transformer_block(x)
```

This is the complete picture: `llm1-3`'s own embedded, position-encoded input vectors enter at the bottom, pass through this exact block repeated dozens of times, and emerge the other end having been repeatedly reshaped by multi-head attention and per-token feed-forward transformation, with residual connections and layer norm keeping the whole stack trainable.

Hands-On Exercises

EXERCISE 1

Explain why a single attention computation forcing every kind of token relationship into one set of weights is a real constraint, and explain specifically how splitting into multiple heads addresses it.

View solution

EXERCISE 2

Explain the specific difference in job between the multi-head attention sublayer and the feed-forward sublayer within one Transformer block, and explain why both are needed rather than just one or the other.

View solution

EXERCISE 3

Explain what residual connections actually do mechanically (x = x + Sublayer(x)), and explain why this becomes especially important once dozens of Transformer blocks are stacked, connecting your answer to nn1-6's own training-stability material.

View solution

CHAPTER 5 QUICK REFERENCE

- **Multi-head attention** — several independent attention computations in parallel subspaces, each free to specialize, concatenated and projected back down
- **Feed-forward sublayer** — a small two-layer network applied independently per token, adding non-linear transformation after attention's own cross-token mixing
- **Residual connections** — `x = x + Sublayer(x)`, giving gradients a direct shortcut through dozens of stacked layers
- **Layer normalization** — stabilizes activations per token; modern models apply it before each sublayer (Pre-LN)
- **One Transformer block** = multi-head attention + feed-forward, each wrapped in a residual connection and layer norm — stacked dozens of times in a real model
- This delivers nn1-9's and nlp1-8's own jointly-deferred "full multi-head, multi-layer Transformer" in full
- **Next chapter:** Decoder-Only vs. Encoder-Decoder: GPT vs. BERT vs. T5

CHAPTER 6 OF 11

Decoder-Only vs. Encoder-Decoder: GPT vs. BERT vs. T5

LLMs

Chapter 6 · Decoder-Only vs. Encoder-Decoder: GPT vs. BERT vs. T5

`llm1-5` built one Transformer block. The 2017 "Attention Is All You Need" paper `nlp1-8` already cited actually proposed stacking these blocks into an **encoder-decoder** pair, for machine translation specifically — `nlp1-8`'s own real motivating task. This chapter surveys the three architecture families that emerged from arranging Transformer blocks differently, and explains why one of them became dominant.

Three Families, Three Attention Patterns

FAMILY	ATTENTION DIRECTION	TRAINING OBJECTIVE	TYPICAL USE
Encoder-only (BERT)	Bidirectional — every token attends to every other token, both directions	Masked language modeling — predict randomly hidden tokens from surrounding context	Classification, understanding tasks
Encoder-decoder (T5)	Encoder bidirectional; decoder causal, cross-attending to the encoder	Text-to-text — map an input sequence to an output sequence	Translation, summarization
Decoder-only (GPT)	Causal — each token attends only to itself and earlier tokens	Next-token prediction over one continuous sequence	General-purpose generation, the dominant modern shape

Encoder-Only: BERT's Bidirectionality

BERT (2018) stacks only encoder-style Transformer blocks, where every token can attend freely to every other token in the sequence, in both directions. This is the exact Transformer-era counterpart of `nlp1-7`'s own bidirectional LSTM — both exist to let a token's own representation be informed by context on both sides, not just what came before. BERT trains via **masked language modeling**: randomly hide some tokens in the input and predict them using the full surrounding context, generalizing `nlp1-5`'s own context-prediction training signal to a full Transformer stack.

Encoder-Decoder: T5 and the Original Transformer

This is the architecture `nlp1-8` already introduced conceptually — an encoder reads the entire input sequence (bidirectionally), and a decoder generates the output sequence one token at a time, at each step cross-attending back to the encoder's own output. T5 generalizes this into a "text-to-text" framing, treating every task — translation, summarization, classification — as converting one string of text into another.

Decoder-Only: Causal Masking

GPT's own family uses no separate encoder at all — one single stack of Transformer blocks, with a critical modification to `llm1-4`'s own attention formula: a **causal mask** that forces each token to attend only to itself and tokens before it, never tokens that come later.

```
scores = (Q @ K.T) / sqrt(d_k)
scores = scores.masked_fill(future_positions, -inf)   # block attention to later tokens
weights = softmax(scores, axis=-1)                      # -inf becomes exactly 0 after softmax
output = weights @ V
```

Setting future-position scores to negative infinity before softmax guarantees their resulting attention weight is exactly zero — a token generating its own prediction can never "peek" at tokens that haven't been generated yet. This single change is what makes autoregressive generation coherent: predicting each next token uses only what has genuinely already been produced.

Why Decoder-Only Became the Dominant General-Purpose Shape

> **DELIVERING NLP1-10'S OWN THIRD CLAIM, MECHANICALLY**
>
> `nlp1-10`'s own capstone named this directly: `nlp1-6`'s sentiment classifier and `nlp1-7`'s NER tagger were genuinely separate pipelines, each with its own architecture and training objective. A decoder-only model needs neither. Nearly any task can be reframed as "generate the correct continuation of this text" — sentiment becomes "Review: ... Sentiment:", NER becomes "Sentence: ... Named entities:". The *same* trained weights, the *same* architecture, perform both — steered entirely by how the input text is framed, not by swapping out the model.

This also explains the simplicity advantage: one stack, one training objective (predict the next token), rather than BERT's separate masked-prediction objective or T5's split encoder/decoder training. Because generating a coherent next token requires the model to have effectively "understood" everything so far, useful general-purpose representations emerge as a side effect of the generative objective alone — no separate understanding-specific objective needs to be designed in.

The GPT Lineage

This decoder-only, causally-masked architecture is the actual mechanism behind `historyai3-7`'s own GPT-1 through GPT-3 lineage namecheck — the architecture story, not yet the training story. How pretraining at real scale actually works is `llm1-7`'s own job; how GPT-3 became ChatGPT through fine-tuning and RLHF is `llm1-9`'s.

Hands-On Exercises

EXERCISE 1

Explain the specific attention-direction difference between BERT, T5, and GPT, and explain why BERT's own bidirectionality is described as the Transformer-era counterpart of nlp1-7's bidirectional LSTM.

View solution

EXERCISE 2

Explain mechanically how causal masking works, using this chapter's own code, and explain why setting future-position scores to negative infinity (rather than, say, zero) is the correct way to block attention to future tokens.

View solution

EXERCISE 3

Using this chapter's own sentiment/NER reframing example, explain precisely why a decoder-only model can perform both nlp1-6's and nlp1-7's own tasks without needing two separately built pipelines, and explain what specifically changes between the two cases (and what doesn't).

View solution

CHAPTER 6 QUICK REFERENCE

- **Encoder-only (BERT)** — bidirectional attention, masked language modeling, understanding tasks; the Transformer-era counterpart of nlp1-7's BiLSTM
- **Encoder-decoder (T5)** — nlp1-8's own original architecture, generalized to a text-to-text framing
- **Decoder-only (GPT)** — causal masking via `-inf` on future positions, next-token prediction, the dominant general-purpose shape
- Causal masking is what makes autoregressive generation coherent — no peeking at ungenerated tokens
- Decoder-only wins on flexibility: one architecture, reframed via text, replaces nlp1-6's and nlp1-7's own separately-built pipelines — nlp1-10's own third claim, delivered mechanically

- The mechanism behind historyai3-7's own GPT lineage — training story deferred to llm1-7 (pretraining) and llm1-9 (fine-tuning/RLHF)
- **Next chapter:** Pretraining at Scale: Self-Supervised Learning

CHAPTER 7 OF 11

Pretraining at Scale: Self-Supervised Learning

LLMs

Chapter 7 · Pretraining at Scale: Self-Supervised Learning

`llm1-6` named next-token prediction as GPT's own training objective without unpacking it. This chapter does — and delivers, with real numbers, `nlp1-10`'s own "self-supervised pretraining taken further" claim.

The Self-Supervised Thread, Traced From nlp1-5

`nlp1-5`'s word2vec and `nlp1-9`'s GloVe both trained on a self-supervised signal — predict a word from its surrounding context, with no hand-labeled data required. LLM pretraining is the same underlying idea, scaled from "predict one missing word in a local window" to "predict every next token across an entire corpus," using the full Transformer stack `llm1-5` built rather than a shallow lookup table.

Causal Language Modeling — The Actual Objective

Given a training sequence, the model's job at *every single position* is to predict the token that actually comes next, using only the tokens before it — exactly the constraint `llm1-6`'s own causal masking enforces.

```
sequence = ["The", "capital", "of", "France", "is", "Paris"]

# one training sequence yields a prediction target at every position:
# "The"                                -> predict "capital"
# "The capital"                        -> predict "of"
# "The capital of"                     -> predict "France"
# "The capital of France"              -> predict "is"
# "The capital of France is"           -> predict "Paris"
```

A single sequence of *N* tokens yields *N* separate next-token training signals, all computable in one forward pass — because `llm1-6`'s own causal mask already prevents any position from seeing ahead, every position's prediction can be trained simultaneously, in parallel, rather than one at a time. This is a genuine efficiency advantage over `nlp1-6`'s own strictly sequential LSTM training, where each step had to wait for the one before it.

The Loss Function

At each position, the model produces a probability distribution over the *entire* vocabulary (`llm1-2`'s own tens of thousands of subword tokens) via softmax, and cross-entropy loss compares that distribution against the actual next token.

NLP1-7'S OWN SOFTMAX, AT VASTLY LARGER SCALE

This is directly the same softmax-over-multiple-classes mechanism `nlp1-7` introduced for NER tagging — except instead of choosing among a handful of entity tags, the model chooses among an entire vocabulary of tens of thousands of possible next tokens, at every single position in every training sequence.

Contrast: Causal LM vs. BERT's Masked LM

	CAUSAL LM (GPT)	MASKED LM (BERT)
Training signal per sequence	Every position — dense	Only the ~15% of randomly masked positions
Context used	Only earlier tokens (causal)	Both directions (bidirectional)
Usable for left-to-right generation?	Yes — training and generation are consistent	No — assumes the whole sequence already exists

This is the concrete, mechanical reason `llm1-6`'s own architecture split holds: BERT's bidirectional context is exactly what makes coherent left-to-right generation impossible for it — predicting a token using information from tokens that come *after* it has no equivalent at real generation time, when those later tokens don't exist yet. GPT's causal objective has no such mismatch.

What "Scale" Actually Means

Real pretraining corpora span hundreds of billions to trillions of tokens — web text, books, code, and more. Real models range from hundreds of millions to hundreds of billions of parameters. This is `nlp1-10`'s own "scale" claim, given its first real numbers; `llm1-8` covers *why* more of each helps, and by how much, in a predictable way.

WHY SELF-SUPERVISION IS WHAT MAKES THIS SCALE POSSIBLE AT ALL

Hand-labeling even a fraction of a trillion tokens is not remotely feasible — compare this to `nlp1-2` / `nlp1-3`'s own small, manually-inspectable datasets. Self-supervision sidesteps this entirely: the "label" for every position is simply the next token already present in the raw text. Any text scraped from the web, digitized from books, or pulled from code repositories becomes usable training data with zero manual annotation — the exact property that makes training at this scale achievable in the first place, not just theoretically desirable.

An Honest Note: This Produces a Base Model, Not an Assistant

A model trained purely on next-token prediction learns to continue text plausibly — it has no built-in notion of "follow this instruction" or "refuse this harmful request." Turning a raw pretrained base model into something that behaves like an assistant is `llm1-9`'s own job: fine-tuning and RLHF.

Hands-On Exercises

EXERCISE 1

Using this chapter's own "The capital of France is Paris" example, explain why a single training sequence of N tokens produces N separate training signals, and explain why this can be computed in parallel rather than one position at a time.

View solution

EXERCISE 2

Explain why BERT's own masked language modeling objective cannot be used for coherent left-to-right generation, while GPT's causal language modeling objective can, using this chapter's own compare-table.

View solution

EXERCISE 3

Explain specifically why self-supervised training is what makes training on trillions of tokens feasible at all, contrasting this with nlp1-2/nlp1-3's own small, hand-inspectable datasets.

View solution

CHAPTER 7 QUICK REFERENCE

- **Causal language modeling** — predict the next token at every position, using only earlier tokens; N tokens yields N training signals, computed in parallel
- Cross-entropy loss over the full vocabulary — nlp1-7's own softmax mechanism, at a far larger scale
- **Causal LM vs. masked LM (BERT):** dense per-position signal vs. sparse masked-position signal; only causal LM stays consistent with real generation
- **Scale:** hundreds of billions to trillions of training tokens, hundreds of millions to hundreds of billions of parameters — nlp1-10's own "scale" claim, given real numbers
- Self-supervision (no hand-labeling required) is the specific property that makes this scale achievable at all

- **Honest note:** pretraining alone produces a base model, not an assistant — llm1-9 covers the fine-tuning/RLHF step that changes that
- **Next chapter:** Scaling Laws

CHAPTER 8 OF 11

Scaling Laws

LLMs

Chapter 8 · Scaling Laws

`llm1-7` gave real numbers for what "scale" means — but not why more of it helps, or by how much. This chapter closes that gap: scale isn't a vague intuition, it's a measured, predictable relationship.

Loss Improves as a Power Law, Not a Guess

As model size (parameters, *N*), dataset size (tokens, *D*), and training compute (*C*) each increase, the cross-entropy loss from `llm1-7`'s own training objective decreases smoothly, following a **power-law** relationship:

```
Loss(N) ≈ (N_c / N) ^ alpha_N

# plotted on a log-log axis, this relationship is a straight line —
# loss keeps falling predictably as N grows, across many orders of magnitude
```

This is what "scaling laws" literally means: laws, discovered empirically (Kaplan et al., 2020), that govern how scale relates to performance. Not a diminishing-returns cliff, not a random walk — a genuinely predictable curve.

Why Predictability Is the Practically Important Part

EXTRAPOLATING BEFORE SPENDING MILLIONS

Because the relationship is a smooth, measurable curve, labs can run many small, cheap training runs at modest scale, fit the power-law curve to those results, and predict with real accuracy how a proposed much larger — and far more expensive — training run will perform, *before* committing the compute budget to actually run it. This is the load-bearing, practical use of scaling laws, not an academic curiosity.

Kaplan (2020) vs. Chinchilla (2022) — A Real Correction

Kaplan et al.'s original 2020 scaling laws suggested that, for a fixed compute budget, model size should be scaled up much faster than dataset size — leading to very large models trained on comparatively modest

amounts of data. GPT-3 is a real example of this era's approach. DeepMind's 2022 Chinchilla paper (Hoffmann et al.) found this was actually suboptimal.

MODEL	PARAMETERS	APPROACH	RESULT
Gopher	280B	Kaplan-style — large model, comparatively less data	Undertrained relative to its own size
Chinchilla	70B	Model size and dataset size scaled roughly equally, same compute budget as Gopher	Outperformed the far larger Gopher

The real finding: for a given compute budget, there's an *optimal ratio* of model size to dataset size — not "bigger model always wins," and not "more data always wins," but a genuine balance. Many large models from the Kaplan-influenced era were significantly undertrained relative to their own parameter count.

Completing nlp1-10's "Scale" Claim

`nlp1-10` asserted that scale matters; `llm1-7` gave it real numbers; this chapter supplies the actual mathematical relationship — a measured power law, with a real, historically-corrected understanding of how to spend a compute budget wisely across model size and data size, rather than "throw more of everything at it."

An Honest Debate: Emergent Abilities

Some capabilities — multi-step arithmetic, certain reasoning tasks — appear to show up suddenly at a scale threshold, rather than improving gradually. This was initially presented as evidence that scale produces genuinely new, qualitatively different capabilities in a discontinuous way.

A GENUINE, UNRESOLVED DEBATE — BOTH SIDES STATED HONESTLY

Later work (Schaeffer et al., 2023, "Are Emergent Abilities a Mirage?") showed that at least some apparent emergence is a **measurement artifact**. If a discontinuous, all-or-nothing metric is used (like exact-match accuracy on a multi-step problem, which scores 0 unless every step is correct), performance can look like it jumps suddenly at some threshold. Switching to a smooth metric (like token-level accuracy or log-likelihood) on the exact same models often reveals the underlying improvement was gradual all along — the "emergence" was partly an artifact of how the capability was measured, not necessarily a real discontinuity in the model itself. This doesn't settle the debate entirely — some researchers maintain certain abilities do show genuine discontinuities — but it's a real, important caveat that shouldn't be skipped.

What Scaling Laws Don't Cover

Scaling laws describe how *pretraining loss* improves with scale — next-token-prediction accuracy, nothing more. They say nothing directly about usefulness as an assistant, safety, or alignment with human intent. A model with excellent pretraining loss is still, per `llm1-7`'s own honest note, only a base model. Turning predictable loss improvement into a genuinely useful, well-behaved assistant is `llm1-9`'s own job.

Hands-On Exercises

EXERCISE 1

Explain what it means for loss to follow a "power law" as scale increases, and explain specifically why this predictability is practically useful to a lab deciding whether to fund an expensive large-scale training run.

View solution

EXERCISE 2

Using this chapter's own Gopher/Chinchilla comparison, explain what the Chinchilla paper actually found, and explain why "bigger model, same compute" turned out to be the wrong lesson to draw from Kaplan's original scaling laws.

View solution

EXERCISE 3

Explain how a discontinuous, all-or-nothing evaluation metric can make a genuinely gradual improvement look like sudden "emergence," and explain why this chapter treats the emergent-abilities question as a real, unresolved debate rather than settled in either direction.

View solution

CHAPTER 8 QUICK REFERENCE

- **Scaling laws** — loss decreases as a smooth power law with model size, data size, and compute; a straight line on a log-log plot
- Predictability lets labs extrapolate small, cheap experiments to forecast large, expensive training runs before funding them
- **Kaplan (2020):** favored scaling model size faster than data — GPT-3-era models, often undertrained
- **Chinchilla (2022):** model size and data size should scale roughly equally for a fixed compute budget — 70B Chinchilla beat 280B Gopher
- **Emergent abilities:** a genuine, honestly unresolved debate — some apparent "sudden" capability jumps are measurement artifacts of discontinuous metrics, not necessarily proof of true discontinuity

- Scaling laws govern pretraining loss only — not usefulness, safety, or alignment; that's llm1-9's own job
- **Next chapter:** Fine-Tuning & RLHF: From GPT-3 to ChatGPT

CHAPTER 9 OF 11

Fine-Tuning & RLHF: From GPT-3 to ChatGPT

LLMs

Chapter 9 · Fine-Tuning & RLHF: From GPT-3 to ChatGPT

`llm1-7` ended with an honest note: pretraining alone produces a base model — a raw next-token predictor with no notion of following an instruction or refusing a harmful request. This chapter covers the real, three-step mechanism that turns a base model into an assistant — the actual process behind `historyai3-7`'s own GPT-3-to-ChatGPT narrative.

Resolving nlp1-9's Own Distinction, at Full Scale

EXACTLY THE SHAPE NLP1-9 PREDICTED

`nlp1-9` named frozen-vs-fine-tuned for a lookup table of word embeddings, and explicitly said this shape mirrors "an entire pretrained network" being either used as-is or further adapted — "only the scale differs." This chapter is where that promise is delivered: everything below is fine-tuning, applied not to an embedding table but to the entire pretrained Transformer from `llm1-5`, using the identical underlying logic nlp1-9 already introduced — start from a broadly-trained representation, then adapt it further using a smaller, more targeted signal.

Step 1: Supervised Fine-Tuning (SFT)

Humans write high-quality examples of what a good response to an instruction looks like — a much smaller, carefully curated dataset than `llm1-7`'s own massive raw pretraining corpus. The base model continues training on these prompt/response pairs, using the identical next-token-prediction objective from `llm1-7` — nothing about the architecture changes (`llm1-6`'s own flexibility point resurfaces here), only the data being trained on.

```
# same objective as llm1-7, different data
# pretraining data:     raw internet text, books, code
# SFT data:              curated (instruction, ideal_response) pairs
```

SFT shifts the model's own behavior toward instruction-following, but it only teaches imitation of the specific examples shown — it can't, by itself, capture the subtler, harder-to-write-down human preferences about what makes one otherwise-reasonable response genuinely *better* than another.

Step 2: Reward Modeling

To capture those preferences, a separate model is trained. The SFT model generates several different responses to the same prompt; humans rank them by preference; a **reward model** is trained to predict that ranking — given a prompt and a response, output a single scalar score estimating how much a human would prefer it.

```
responses = [sft_model.generate(prompt) for _ in range(k)]   # several candidate responses
ranking = human_rank(responses)                               # humans order them by preference
reward_model.train(prompt, responses, ranking)                # learn to predict the ranking
```

Step 3: Reinforcement Learning from Human Feedback (RLHF)

The SFT model is fine-tuned further, this time via reinforcement learning: it generates a response, the trained reward model scores it, and the model's own parameters are updated (via Proximal Policy Optimization, PPO) to increase the likelihood of generating higher-reward responses in the future.

> **A REAL SAFEGUARD AGAINST GAMING THE REWARD MODEL**
> A KL-divergence penalty against the original SFT model is typically included in the RLHF objective — it discourages the model from drifting too far from its own SFT starting point purely to exploit quirks in the reward model rather than genuinely satisfying human preference. Without it, a model can learn to produce responses that score artificially well on the reward model without actually being better in any way a human would recognize.

The Actual Mechanism Behind the ChatGPT Moment

`historyai3-7` already named the real, historical November 2022 ChatGPT launch and its RLHF namecheck. This three-step pipeline — SFT, then reward modeling, then RLHF/PPO — is that mechanism, made concrete: GPT-3 (a base model, exactly per `llm1-7`'s own honest note) became ChatGPT through this pipeline, not through additional pretraining or additional scale. `llm1-8`'s own scaling laws describe how pretraining loss improves with scale — a completely separate axis from this chapter's own instruction-following and preference-alignment transformation.

An Honest Limitation

> **REWARD HACKING — A REAL, DOCUMENTED RISK**
> RLHF optimizes a model toward whatever the reward model rewards — and the reward model is itself only an imperfect proxy for genuine human preference, trained on a finite set of human rankings. A model can learn to exploit specific quirks or blind spots in the reward model (a real instance of Goodhart's Law — when a measure becomes a

target, it stops being a good measure) rather than genuinely improving in the way human raters actually intended. RLHF measurably improves helpfulness and reduces harmful outputs in practice, but it is not a guarantee of perfect alignment with human intent.

Hands-On Exercises

EXERCISE 1

Explain precisely why this chapter's own SFT/reward-model/RLHF pipeline counts as "fine-tuning applied to an entire pretrained network," directly connecting this to nlp1-9's own frozen-vs-fine-tuned distinction for embeddings.

View solution

EXERCISE 2

Explain why supervised fine-tuning alone is insufficient to capture human preferences between two otherwise-reasonable responses, and explain specifically what the reward model adds that SFT cannot provide on its own.

View solution

EXERCISE 3

Explain what reward hacking is, why it's a real risk specific to RLHF's own design, and explain how the KL-divergence penalty against the original SFT model helps mitigate it.

View solution

CHAPTER 9 QUICK REFERENCE

- **SFT** — continue training the base model on curated (instruction, ideal response) pairs, same objective as llm1-7, different data
- **Reward modeling** — a separate model trained to predict human preference rankings between candidate responses
- **RLHF/PPO** — fine-tune the SFT model to maximize the reward model's own score, with a KL penalty against drifting too far from the SFT starting point
- This three-step pipeline is the real mechanism behind historyai3-7's own GPT-3-to-ChatGPT narrative — not more pretraining, not more scale (llm1-8)
- Resolves nlp1-9's own frozen-vs-fine-tuned distinction at full LLM scale, exactly as that chapter predicted
- **Honest limitation:** reward hacking — the reward model is an imperfect proxy for genuine human preference, a real instance of Goodhart's Law

- **Next chapter:** Context Windows, Quadratic Attention & Honest Limitations

CHAPTER 10 OF 11

Context Windows, Quadratic Attention & Honest Limitations

LLMs

Chapter 10 · Context Windows, Quadratic Attention & Honest Limitations

`llm1-3` flagged, and deferred, exactly why a context window has a hard limit. This chapter delivers that in full — and gives two of the most commonly hand-waved LLM behaviors, hallucination and knowledge cutoff, real mechanical explanations instead.

What a Context Window Actually Is

The maximum number of tokens (`llm1-2`'s own BPE tokens) a model can process in one forward pass. Two genuinely separate mechanisms both cap this number:

1. **Positional encoding's own limit** — `llm1-3`'s own deferred warning: a learned positional embedding table simply has no entry beyond whatever maximum position it was trained with.
2. **The computational cost of self-attention itself** — this chapter's own new material, and the deeper, more fundamental constraint.

Quadratic Attention Cost

Recall `llm1-4`'s own scaled dot-product attention: `Q @ K.T` computes a similarity score between *every* pair of tokens. For a sequence of length N, that's an $N \times N$ matrix of scores — both the compute and the memory required scale with N^2, not N.

SEQUENCE LENGTH (N)	SCORE PAIRS (N^2)	RELATIVE TO N=1,000
1,000	~1,000,000	1×
2,000 (2× longer)	~4,000,000	4×
10,000 (10× longer)	~100,000,000	100×

A REAL ARCHITECTURAL CONSTRAINT, NOT A PRODUCT DECISION

Doubling the context window doesn't double the cost — it roughly quadruples it. This is why context windows historically grew slowly, and why any given amount of available compute and memory imposes a genuine, hard ceiling

on sequence length. `llm1-5`'s own multi-head attention computes this same N×N score matrix independently per head, multiplying the cost further by the number of heads on top of the underlying N^2 scaling.

REAL MITIGATIONS EXIST — BRIEFLY, HONESTLY

Techniques like sparse attention patterns, sliding-window attention, and memory-efficient exact implementations (such as FlashAttention) reduce the practical cost of running attention, and are genuinely used in real systems. None of them eliminate the fundamental N^2 relationship for full, dense attention over the whole sequence — they trade completeness or memory layout for efficiency in various ways. A full technical treatment of these techniques is beyond this course's own scope.

Hallucination, Explained Mechanically

Per `llm1-7`, a model is trained to predict the most probable next token given context — nothing more. It has no built-in mechanism for verifying factual truth, only for producing plausible-sounding continuations shaped by patterns learned during pretraining.

NO BUILT-IN "I DON'T KNOW"

When the model has no strong pattern support for the true continuation, its core mechanism has no fallback behavior for that case — it simply continues producing *some* plausible-sounding sequence of tokens, drawn from the same probability distribution it always samples from, whether or not the result is actually true. `llm1-9`'s own fine-tuning/RLHF pipeline can and does train a model to say "I don't know" more often in situations where that response is reward-preferred — but that's a learned behavioral pattern layered on top via SFT and RLHF, not a fact-checking mechanism built into the base architecture from `llm1-5` onward.

Knowledge Cutoff, Explained Mechanically

Per `llm1-7`, pretraining uses a fixed, finite corpus, collected up to some point in time. Everything the model "knows" is entirely encoded in its trained parameters, shaped exclusively by whatever text existed in that corpus. Nothing about the pretraining process gives the model any live, ongoing access to information published after its own training data was collected — there is no built-in internet connection, no update mechanism running in the background.

Anything that happened after the corpus's own collection date is simply absent from the patterns the model learned — not because the model is vaguely "unaware," but because that information never existed in any data used to shape its parameters at all. Retrieval-augmented generation, already covered in `claude-adv1`, works around this by injecting live external information directly into the prompt at inference time — a way of supplying missing context, not a way of updating the model's own trained parameters.

Closing the Loop Back to Chapter 1

`llm1-1` opened by claiming this course would explain *why* `prompt1`'s own techniques work, not replace them. Techniques like providing sources in a prompt, or explicitly instructing a model to say "I don't know" when uncertain, work precisely *because* of the mechanisms covered in this chapter — they compensate for a model with no built-in fact-checker and no live knowledge feed, exactly as described here, mechanically, rather than as an unexplained quirk.

Hands-On Exercises

EXERCISE 1

Using this chapter's own table, explain why doubling a context window's length roughly quadruples attention's own computational cost, and explain why this is described as a real architectural constraint rather than an arbitrary product limit.

View solution

EXERCISE 2

Explain the mechanical reason hallucination happens, tracing it back to llm1-7's own training objective, and explain why RLHF-trained "I don't know" responses (llm1-9) don't contradict this explanation.

View solution

EXERCISE 3

Explain why knowledge cutoff is a direct, structural consequence of how pretraining works (llm1-7), and explain why retrieval-augmented generation is described as working around this limitation rather than fixing it.

View solution

CHAPTER 10 QUICK REFERENCE

- **Context window limit** — capped by both positional encoding's own max length (llm1-3) and attention's own quadratic cost (this chapter)
- **Quadratic attention cost:** N tokens → N^2 score pairs; doubling length roughly quadruples cost — a real, hard architectural ceiling
- Real mitigations (sparse/sliding-window attention, FlashAttention) reduce practical cost but don't eliminate N^2 scaling for full dense attention

- **Hallucination:** the model has no built-in fact-checker — only next-token plausibility (llm1-7); RLHF (llm1-9) layers "I don't know" as a learned behavior, not a core capability
- **Knowledge cutoff:** a direct consequence of training on a fixed, finite corpus (llm1-7) — RAG supplies live context at inference time, without updating the model's own parameters
- Closes the loop to llm1-1: this is exactly why prompt1's own techniques (sourcing, "say if unsure") work
- **Next chapter:** Capstone — Tracing a Prompt Through a Real LLM, End to End

CHAPTER 11 OF 11

Capstone: Tracing a Prompt Through a Real LLM, End to End

LLMs

Chapter 11 · Capstone: Tracing a Prompt Through a Real LLM, End to End

`llm1-1` opened with a deliberately incomplete preview of `"The capital of France is"`, promising the full trace would wait until every piece was built. Every piece is now built. Here is the full trace.

Step 1 — Tokenization (llm1-2)

```
prompt = "The capital of France is"
tokens = ["The", " capital", " of", " France", " is"]   # BPE subword tokens
token_ids = [464, 3139, 286, 4881, 318]                    # looked up in the vocabulary
```

`llm1-2`'s own Byte-Pair Encoding breaks the raw prompt string into subword tokens, each mapped to an integer ID in the model's fixed vocabulary — no out-of-vocabulary risk, per that chapter's own byte-level fallback guarantee.

Step 2 — Embedding & Positional Encoding (llm1-3)

```
x = embedding_table[token_ids]              # each token ID -> its learned vector
x = x + positional_encoding[0:5]              # position 0..4 added, per token
```

Each token ID is looked up in the jointly-trained embedding table, then combined with its own positional encoding — the exact mechanism that gives the model any sense of order at all, resolving `nlp1-8`'s own deferred gap. This produces the actual vectors that enter the first Transformer block.

Step 3 — Multi-Head Self-Attention Across Stacked Layers (llm1-4 / llm1-5 / llm1-6)

```
for layer in range(num_layers):                    # dozens of stacked blocks
    x = x + multi_head_attention(layer_norm(x), causal_mask=True)
```

```
    x = x + feed_forward(layer_norm(x))
```

At every layer, each token's own vector is reshaped by causally-masked (`llm1-6`) multi-head attention (`llm1-5`), built from the Query/Key/Value mechanism (`llm1-4`). The token at position 4 (`" is"`) can attend to itself and every earlier token — never anything later, since nothing later exists yet in this generation step. By the final layer, its own vector has been shaped by learned relevance to `"The"`, `"capital"`, `"of"`, and `"France"`, repeatedly, across every stacked block.

Step 4 — Next-Token Prediction (llm1-7 / llm1-8)

```
logits = final_layer_output[-1] @ output_projection    # project to vocabulary size
probabilities = softmax(logits)
# probabilities["Paris"] ≈ highest, if pretraining (llm1-7) at sufficient scale (llm1-8) went well
```

The final vector at the last position is projected into a probability distribution over the *entire* vocabulary — `llm1-7` 's own causal language modeling objective, at generation time. If the model was pretrained on enough data, at enough scale, following `llm1-8` 's own measured power-law relationship, `"Paris"` receives the highest probability of any token in the vocabulary.

Step 5 — The Autoregressive Loop (llm1-6)

The predicted token is appended to the sequence, and the entire process — embed, attend across every layer, predict — repeats to generate the token after that, one token at a time, each one causally valid given only what has genuinely been generated so far.

An Honest Aside: Real Deployments Aren't Raw Completions

> **LLM1-9'S OWN FINE-TUNING, IN PRACTICE**
>
> A real assistant deployment doesn't send the raw prompt shown above — it wraps it in a conversation format (system/user/assistant turn markers) the model was specifically trained to follow via `llm1-9` 's own SFT and RLHF pipeline. The trace above shows the base mechanism; real assistant behavior is this exact mechanism, operating on text that's been formatted according to what fine-tuning taught the model to expect.

Chapter Attribution Table

CHAPTER	WHAT IT CONTRIBUTED TO THIS TRACE
llm1-1	The scope-setting preview this capstone now completes
llm1-2	BPE tokenization — turning the raw prompt into token IDs
llm1-3	Token embeddings and positional encoding — the input vectors
llm1-4	Query/Key/Value and scaled dot-product attention — the core computation
llm1-5	Multi-head attention, feed-forward layers, residuals, and layer norm — one full Transformer block
llm1-6	Causal masking and the decoder-only architecture — why generation stays coherent
llm1-7	The next-token prediction objective — why "Paris" becomes the most probable output
llm1-8	Why scale makes that prediction reliable — the measured power law behind pretraining quality
llm1-9	Why a real deployment behaves like a helpful assistant rather than a raw text-completion engine
llm1-10	Why this trace has a maximum possible length, and why the model might not "know" the answer at all

Closing the Loop on nlp1-10's Three Claims

DELIVERED, NOT JUST ASSERTED

- **Scale** — real numbers (llm1-7) and a real, measured power law (llm1-8), not a vague intuition.
- **Self-supervised pretraining taken further** — llm1-7's own next-token objective, extending nlp1-5's/nlp1-9's own context-prediction signal across an entire corpus.
- **One flexible architecture vs. many task-specific pipelines** — llm1-6's own decoder-only reframing, proven with a worked example replacing nlp1-6's and nlp1-7's own genuinely separate pipelines.

This is also the LLM analogue of what `imgai1` did for image models, exactly as `llm1-1` set out — a full mechanism, traced end to end, sitting underneath `prompt1`'s and `claude-adv1`'s own practical territory rather than replacing it.

Hands-On Exercises

EXERCISE 1

Using this chapter's own code, trace each step of the pipeline back to the specific chapter it came from, and explain why the trace stops being a "preview" (as in llm1-1) and becomes a genuine mechanical account here.

View solution

EXERCISE 2

Explain why the token at position 4 ("is") can attend to every earlier token but nothing later, connecting your answer to llm1-6's own causal masking mechanism, and explain why this restriction is essential for the autoregressive generation loop in Step 5 to make sense.

View solution

EXERCISE 3

For each of nlp1-10's own three named differences between its own pipelines and an LLM, explain specifically what in this capstone's own trace demonstrates that claim concretely, rather than merely restating it abstractly.

View solution

Scope Note — What This Course Deliberately Doesn't Cover

HONESTLY OUT OF SCOPE

- No from-scratch training of a production-scale model — this course explains the mechanism, not a runnable training pipeline at real scale.
- No deployment or serving infrastructure — that's a separate, substantial engineering discipline of its own.
- No coverage of any specific commercial API's own features or pricing — that territory belongs to `claude-adv1`, exactly as `llm1-1` scoped from the start.

CHAPTER 11 QUICK REFERENCE — COURSE SUMMARY

- **The full trace:** tokenize (2) → embed + position (3) → multi-head causal attention across stacked layers (4/5/6) → predict next token (7, reliable per 8) → repeat (6) → shaped into an assistant (9), within a hard length limit (10)
- **What was diagnosed and fixed across the course:** nlp1-9's OOV gap (2), nlp1-8's missing order (3) and undelivered full Transformer (5), nlp1-6/nlp1-7's separate pipelines (6), nlp1-10's three asserted-but-unproven claims (7/8/6)
- This completes the LLMs course (11 chapters) — the fifth of six courses in the Data Science & ML subject
- **Remaining in the subject:** Data Science & ML Projects (dsproj1/dsproj2)