Modern NLP Practice

Tokenization · Fine-Tuning · Retrieval-Augmented Generation

Ramaseshan Ramachandran

Subword Tokenization

WHY WORD-LEVEL TOKENS FAIL

- Heaps' law: the vocabulary keeps growing with the corpus. A word-level vocabulary is never complete
- Out-of-vocabulary (OOV) words are inevitable: new names, typos, code, morphology (*play, plays, played, playing, replayable*)
- A $|V|$-sized softmax over millions of words is expensive to train and store
- Morphologically rich and agglutinative languages (Tamil, Finnish, Turkish) explode the vocabulary far faster than English

---

Character-level tokens solve OOV but make sequences very long and force the model to relearn every word from scratch.

Subwords are the compromise: a fixed-size vocabulary with no OOV.

THE IDEA

- Choose a **fixed vocabulary size** (e.g., 32,000 or 100,000) *before* training
- Frequent words stay whole: `the`, `cricket`
- Rare words split into frequent pieces: `tokenization` $\rightarrow$ `token` + `ization`
- Worst case falls back to characters/bytes, so *any* string can be encoded
- This is Zipf's law used constructively: spend the vocabulary budget on the frequent units, compose the rare ones

The same idea appeared in FastText's character n-grams; subword tokenizers make it the *input representation* of every modern LLM.

BYTE-PAIR ENCODING (BPE)[1]

Training (learning the merges):

1. Start with the vocabulary of individual characters (or bytes)
2. Count all adjacent symbol pairs in the corpus
3. Merge the **most frequent pair** into a new symbol; add it to the vocabulary
4. Repeat until the vocabulary reaches the chosen size

Encoding (using the tokenizer): apply the learned merges to new text *in the order they were learned*.

The number of merges is the only hyperparameter: $|V| = |\text{characters}| + |\text{merges}|$.

[1] SENNRICH, HADDOW, AND BIRCH, "NEURAL MACHINE TRANSLATION OF RARE WORDS WITH SUBWORD UNITS", ACL 2016. ORIGINALLY A 1994 DATA-COMPRESSION ALGORITHM (GAGE).

BPE: A WORKED EXAMPLE

Corpus (word, frequency), with _ marking end-of-word:

`low_`:5 `lower_`:2 `newest_`:6 `widest_`:3

Step	Most frequent pair	New symbol
1	(e,s) 9×	es
2	(es,t) 9×	est
3	(est,_) 9×	est_
4	(l,o) 7×	lo
5	(lo,w) 7×	low
6	(e,w) 6×†	ew ···

After 5 merges: `lowest_` (never seen!) → `low est_`

- The unseen word *lowest* is encoded from learned pieces, so there is **no OOV**
- †Step 6 is a **tie** ((n,e), (e,w), (w,est_) all 6×). A fixed rule, here the lexicographically smallest pair, breaks it *deterministically*, so the tokenizer is reproducible.

THE TOKENIZER FAMILY

- **BPE** uses frequency-based merges, as in GPT-2/3/4 (byte-level BPE: base symbols are the 256 bytes, so even emoji and binary survive)
- **WordPiece** chooses merges by likelihood gain rather than raw frequency, as in BERT
- **Unigram LM** starts from a large candidate vocabulary and *prunes* it, keeping the subwords that maximize corpus likelihood, as in SentencePiece
- **SentencePiece** is a toolkit (Unigram or BPE) that treats the input as a raw character stream, so it needs no pre-tokenization. That matters for languages without spaces

Model	Tokenizer	$\lvert V\rvert$
BERT	WordPiece	30,522
GPT-2	byte-level BPE	50,257
Llama 2	SentencePiece (BPE)	32,000
Llama 3 / GPT-4o	BPE (tiktoken-style)	~128K–200K

WHY PRACTITIONERS MUST CARE

- **You pay per token.** API cost, context-window budget, and latency are all measured in tokens, not words
- **Fertility** = tokens per word. English ≈ 1.3; under an English-heavy tokenizer, Indian-language text can cost 3–8$\times$ more tokens for the same content. The same prompt is slower, costlier, and effectively has a smaller context window
- **Numbers and arithmetic**: `12345` may split as `123`+`45`. This is one reason LLMs are unreliable at digit-level arithmetic
- **Character puzzles**: a model that sees `straw`+`berry` cannot easily count the r's in *strawberry*
- **Code and whitespace**: modern tokenizers merge runs of spaces so indented code tokenizes efficiently

SUMMARY AND EXERCISES

- Subword tokenization gives a **fixed vocabulary with no OOV**. It is the practical answer to Heaps' law
- BPE = greedy merge of frequent pairs; WordPiece and Unigram are likelihood-based variants
- The tokenizer is part of the model: it shapes cost, context use, multilingual fairness, and failure modes

Exercises

1. Implement `bpe_train(corpus, num_merges)` and `bpe_encode(word)` by hand (no libraries) and reproduce the worked example.
2. Compute the fertility of an English and a Tamil (or Hindi) paragraph under the GPT-2 tokenizer. Explain the difference using Zipf's law.
3. Train SentencePiece with $|V| = 1\text{K}$, 8K, and 32K on the same corpus; plot average tokens per sentence vs. $|V|$.

Fine-Tuning and PEFT

THE ADAPTATION SPECTRUM

How do we make a general pre-trained model good at *our* task?

Method	Weights changed	When to use
Prompting / few-shot	none	task is common, examples fit in context
RAG	none	knowledge is external and changes often
PEFT (LoRA, . . .)	$< 1\%$	new *behavior/style/domain*, modest compute
Full fine-tuning	all	large data, maximal quality, big budget
Pre-training	all, from scratch	you are an AI lab

Rule of thumb: RAG adds **knowledge**; fine-tuning adds **behavior**. They are complementary, not competitors.

FINE-TUNING: THE OBJECTIVE

Take the pre-trained parameters θ_0 and continue training on task data $\{(x^{(i)}, y^{(i)})\}$ with the same cross-entropy loss:

$$\mathscr{L}(\theta) = -\sum_i \sum_t \log p_\theta\left(y_t^{(i)} \,\middle|\, x^{(i)}, y_{<t}^{(i)}\right), \qquad \theta \text{ initialized at } \theta_0 \tag{1}$$

- **Domain adaptation**: continue next-token prediction on in-domain text (legal, medical, code)
- **Instruction / supervised fine-tuning (SFT)**: (instruction, response) pairs. This is the first step of alignment we saw earlier
- **Task fine-tuning**: classification, extraction, translation pairs
- Use a **small learning rate** (10^{-5} vs. 10^{-3} from scratch): we want to *nudge* θ_0, not erase it

WHY FULL FINE-TUNING DOES NOT SCALE

For a 7B-parameter model in 16-bit precision:

Weights	14 GB
Gradients	14 GB
Adam optimizer states ($2\times$, in fp32)	56 GB
Total (before activations!)	~84 GB

- Beyond any single consumer GPU (24 GB)
- One full copy of the model *per task*. Ten tasks = 140 GB of checkpoints
- **Catastrophic forgetting**: aggressive updates overwrite general abilities learned in pre-training

Can we adapt the model by training only a *tiny* number of parameters?

LOW-RANK ADAPTATION (LORA)[2]

Hypothesis: the *change* needed to adapt a weight matrix has low intrinsic rank. It is the same low-rank intuition as SVD/LSA earlier in the course.

$$h = W_0 x + \Delta W x = W_0 x + \underbrace{B}_{d \times r} \underbrace{A}_{r \times k} x, \qquad r \ll \min(d, k) \tag{2}$$

- **Freeze** the pre-trained W_0; train only A and B
- A initialized with small random values, B initialized to zero $\Rightarrow \Delta W = BA = 0$ at the start: training begins exactly at the pre-trained model
- Typically applied to the attention projection matrices (W_Q, W_V, ...)
- At deployment, merge $W = W_0 + BA$ for **zero extra inference cost**

[2] HU ET AL., "LORA: LOW-RANK ADAPTATION OF LARGE LANGUAGE MODELS", ICLR 2022.

LORA: HOW MUCH DO WE SAVE?

One attention projection in Llama-2-7B: $d = k = 4096$.

$$\text{Full } \Delta W : 4096 \times 4096 \approx 16.8\text{M params}$$
$$\text{LoRA } (r = 8) : 4096 \times 8 + 8 \times 4096 \approx 65\text{K params} \quad (\approx 0.4\%)$$

- Across the whole model, LoRA typically trains 0.1–1% of the parameters
- Optimizer states shrink proportionally. A 7B model becomes trainable on a single 24 GB GPU
- Each task's adapter is a few **megabytes**: keep one base model, swap adapters per task/customer
- Freezing W_0 also mitigates catastrophic forgetting

THE PEFT FAMILY

- **LoRA** is a low-rank update to frozen weights (the current default)
- **QLoRA** quantizes the frozen base model to 4-bit and trains LoRA adapters in 16-bit on top: fine-tune a 65B model on one 48 GB GPU[3]
- **Adapters** are small bottleneck layers inserted between transformer sublayers (they add inference latency, unlike LoRA)
- **Prompt / prefix tuning** learns continuous "virtual token" embeddings prepended to the input, leaving weights untouched
- **BitFit** trains only the bias terms

All share one design: **freeze the knowledge, train a small steering wheel.**

[3] Dettmers et al., "QLoRA: Efficient Finetuning of Quantized LLMs", NeurIPS 2023.

A PRACTICAL FINE-TUNING RECIPE

1. **Try prompting first.** Fine-tune only when few-shot prompting demonstrably falls short
2. **Data before compute**: 1,000 clean, diverse, deduplicated examples beat 100,000 noisy ones
3. Hold out a test set *before* looking at the data (the same discipline as every model in this course)
4. Start with LoRA: $r \in \{8, 16\}$, learning rate $\sim 10^{-4}$, 1–3 epochs
5. Evaluate on the held-out set *and* on a general benchmark. Watch for forgetting and regression
6. Compare against the RAG alternative: if the failure is *missing knowledge*, fine-tuning is the wrong tool

SUMMARY AND EXERCISES

- Fine-tuning continues pre-training on task data from θ_0; full fine-tuning is memory-hungry and forgets
- LoRA trains a low-rank $\Delta W = BA$ on frozen weights: $< 1\%$ of parameters, mergeable, swappable
- Choose by failure mode: knowledge $\rightarrow$ RAG; behavior/format/style $\rightarrow$ PEFT

Exercises

1. Compute total LoRA parameters for Llama-2-7B with $r = 16$ applied to W_Q and W_V in all 32 layers. What fraction of 7B is that?
2. Fine-tune a small model (e.g., 1–3B) with LoRA on 500 instruction pairs; report held-out accuracy vs. the zero-shot baseline.
3. Ablate $r \in \{2, 8, 64\}$: plot quality vs. trainable parameters. Where does it saturate, and why is that consistent with the low-rank hypothesis?

Retrieval-Augmented Generation (RAG)

An LLM's knowledge is frozen in its weights (*parametric* knowledge). This fails when:

- **Knowledge cutoff** means events after training are simply unknown
- **Private data** is invisible: your institution's documents were never in the training set
- **Hallucination** makes the model fluently invent facts it does not have
- **Attribution** fails because weights cannot cite a source. Retrieved passages can

---

Idea[4]: don't store the facts in the weights. Instead **retrieve** the relevant text at query time and put it *in the context window*:

$$p(y \mid x) \approx p_{\text{LLM}}\big(y \mid x, \text{retrieve}(x)\big) \tag{3}$$

[4] Lewis et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks", NeurIPS 2020.

THE RAG PIPELINE

Offline indexing:

1. Split documents into **chunks** (e.g., 200–500 tokens, with overlap)
2. Embed each chunk into a vector: $c_j = E(\text{chunk}_j)$
3. Store the vectors in a **vector index**

Online answering:

1. Embed the query: $q = E(x)$
2. Retrieve top-k chunks by similarity: top-k_j $\cos(q, c_j)$
3. (Optional) **rerank** the candidates with a stronger model
4. Build the prompt: instructions + retrieved chunks + question
5. The LLM generates an answer *grounded* in the retrieved text, ideally with citations

RETRIEVAL: THIS IS OUR COURSE COMING FULL CIRCLE

- **Sparse retrieval** uses TF-IDF and its tuned successor **BM25**. Exact-term matching needs no training and is still a strong baseline
- **Dense retrieval** embeds query and passage into the *same* vector space and uses cosine similarity. It is the distributional hypothesis, now at sentence scale
- Sentence embeddings are trained *contrastively*: pull (query, relevant passage) pairs together, push random passages apart. This is the same positive/negative structure as word2vec's negative sampling
- Dense retrieval matches *meaning* ("heart attack" $\leftrightarrow$ "myocardial infarction"); sparse retrieval matches *strings* (exact IDs, rare names)
- **Hybrid** (BM25 + dense, scores fused) is the common production choice

Searching millions of vectors uses approximate nearest-neighbour indexes (e.g., HNSW, FAISS). These give sub-linear search at a small recall cost.

BM25, PRECISELY

The workhorse of sparse retrieval scores a document d for a query q over a collection of N documents:

$$\text{score}(q, d) = \sum_{t \in \text{unique}(q)} \text{idf}(t) \cdot \frac{\text{tf}\,(k_1 + 1)}{\text{tf} + k_1 \left(1 - b + b\,\frac{\text{dl}}{\text{avgdl}}\right)}, \quad \text{idf}(t) = \ln\!\left(\frac{N - \text{df} + 0.5}{\text{df} + 0.5} + 1\right) \tag{4}$$

with tf = occurrences of t in d, df = documents containing t, dl = length of d, avgdl = average length. Standard $k_1 = 1.5$, $b = 0.75$.

- **idf**: rare terms dominate the score. This is exactly TF-IDF's instinct, smoothed
- **tf saturation** (k_1): the 10th occurrence of a term adds far less than the 1st. Raw TF-IDF instead grows linearly
- **Length normalisation** (b): long documents match more terms by luck alone; dividing by dl/avgdl removes that advantage. *Dropping this term is the classic implementation bug*

You will implement precisely this formula in Assignment 8. You will also compute one term of it by hand.

DESIGN CHOICES THAT DECIDE RAG QUALITY

- **Chunking** has two failure modes. Too small: context is lost. Too large: the relevant sentence is diluted and the prompt budget is wasted. Respect document structure (sections, paragraphs); overlap chunks
- With k **and context budget**, more chunks $\neq$ better answers. Irrelevant chunks actively mislead the model
- **Reranking** uses a cross-encoder that reads (query, passage) *together* and re-orders the top-100 into a sharp top-5
- **Query transformation** rewrites conversational questions into standalone queries and decomposes multi-hop questions
- **Prompting** instructs the model to answer *only* from the provided context and to say "not found" otherwise

EVALUATING A RAG SYSTEM

Evaluate the two stages separately. A wrong answer can be a retrieval failure or a generation failure:

- **Retrieval**:
  - **Recall@k** asks whether the gold passage is among the top k
  - **MRR / nDCG** asks how high it is ranked
- **Generation** (given the retrieved context):
  - **Faithfulness / groundedness** asks whether every claim is supported by the retrieved text
  - **Answer relevance** asks whether it actually answers the question
  - Scored with exact match/F1 where possible, LLM-as-judge with human spot-checks otherwise

Typical failures: the answer exists but was never retrieved (fix retrieval), or it was retrieved and the model ignored it (fix prompting/reranking).

RAG VS. FINE-TUNING VS. LONG CONTEXT

	Strength	Weakness
RAG	fresh/private knowledge, citations	retrieval quality is the ceiling
Fine-tuning	behavior, format, style, domain voice	does not add reliable facts
Long context	small, static corpora; just paste it	cost per query; "lost in the middle"

- Million-token contexts shrink RAG's territory but don't remove it: corpora are terabytes, and attention over everything on every query is not economical
- Production systems combine them: a fine-tuned model, behind a retrieval layer, with a generous context window
- Agentic variants let the model *decide* when to search and to issue follow-up queries (multi-hop)

SUMMARY AND EXERCISES

- RAG moves knowledge out of the weights and into a searchable index consulted at query time
- Retrieval reuses this course's foundations: TF-IDF/BM25, embeddings, cosine similarity, now at passage scale
- Quality lives in the unglamorous parts: chunking, reranking, prompt discipline, and two-stage evaluation

Exercises

1. Build BM25 and dense retrieval over the COVID-19 abstracts; report Recall@5 on 20 questions each. Find two queries where sparse beats dense, and two where dense beats sparse. Explain why.
2. Ablate chunk size $\in \{128, 256, 512\}$ tokens with fixed k: measure answer F1.
3. Construct 5 questions whose answers are *not* in the corpus. Does your system say "not found" or hallucinate? Fix it via prompting and report before/after.

Thank you

Linkedin:Ramaseshan
Email:Ramaseshan Ramachandran