CS498: Algorithmic Engineering

Lecture 26: The Transformer & Training

Elfarouk Harb

University of Illinois Urbana-Champaign

Week 14

Outline

1. The Transformer Architecture
2. Training
3. Generation

1 The Transformer Architecture

2 Training

3 Generation

We Have Attention. What's the Full Picture?

What we built last time:

- Multi-head attention (with causal masking)
- Positional encoding (RoPE)

What we still need:

- Normalization to keep activations stable
- Feed-forward network (FFN) for nonlinear computation per position
- Residual connections for gradient flow through deep stacks

Let's see how these combine into the transformer block.

Attention = Communication, FFN = Computation

Attention: tokens TALK to each other

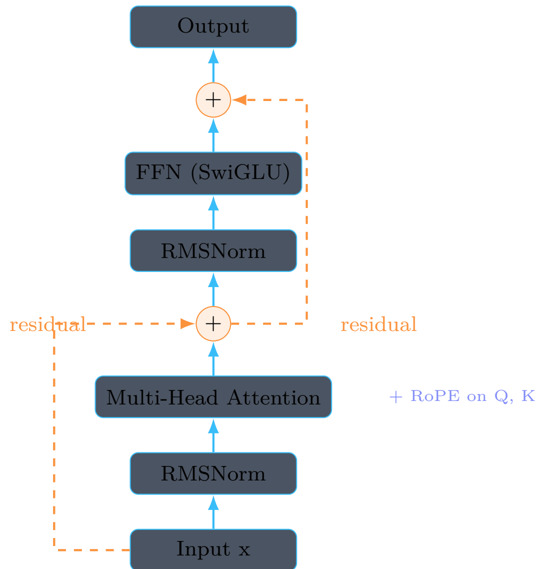
- Token i gathers information from other tokens
- "What context is relevant to me?"

FFN: each token THINKS independently

- Process the gathered information through a nonlinear network
- "What do I do with this context?"

Both are needed: you need to both gather context and process it. A transformer block alternates between communication (attention) and computation (FFN).

The Transformer Block



The FFN: SwiGLU

Original FFN (Vaswani et al., 2017):

$$\mathrm{FFN}(x) = W_2 \cdot \mathrm{ReLU}(W_1 x)$$

Two weight matrices, ReLU activation.

Modern FFN (SwiGLU) (Shazeer, 2020):

$$\mathrm{FFN}(x) = W_2 \cdot \big(\mathrm{SiLU}(W_1 x) \odot W_3 x\big)$$

- Three weight matrices, gated activation
- $\mathrm{SiLU}(z) = z \cdot \sigma(z)$ (smooth version of ReLU)
- $\odot$ is element-wise gating: $W_3 x$ controls which features pass through

FFN Dimensions

Typical hidden dimension of FFN: $d_{ff} \approx \frac{8}{3}d$ (for SwiGLU).

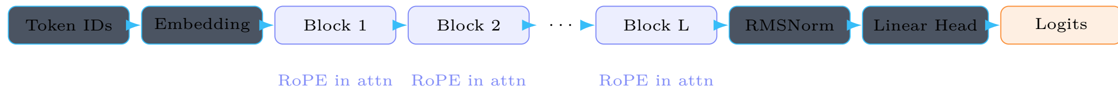
- $W_1 \in \mathbb{R}^{d_{ff} \times d}$, $W_3 \in \mathbb{R}^{d_{ff} \times d}$, $W_2 \in \mathbb{R}^{d \times d_{ff}}$
- Total: $3 \times d \times d_{ff} \approx 8d^2$ parameters

For Llama-3 8B ($d = 4096$, $d_{ff} = 14336$):

- $3 \times 4096 \times 14336 \approx 176M$ parameters per block in the FFN alone
- The FFN is the majority of the parameters in each block

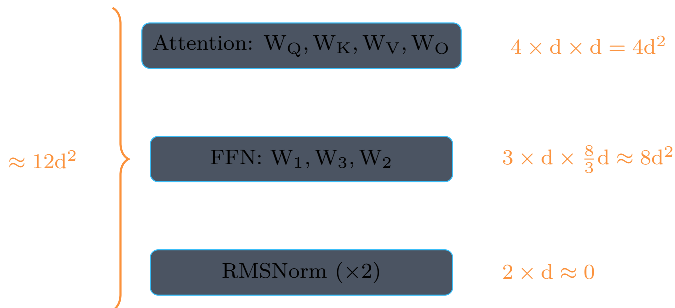
Intuition: the model needs a lot of capacity for “thinking” (FFN), not just “communication” (attention).

Stacking Blocks: The Full TransformerLM



- Embedding: token ID $\rightarrow$ vector in $\mathbb{R}^{d}$
- L transformer blocks: each applies attention (with RoPE on Q, K) then FFN
- Final RMSNorm: stabilize before output projection
- Linear head: project $\mathbb{R}^{d} \rightarrow \mathbb{R}^{|vocab|}$ to get logits

Counting Parameters: One Block



One transformer block $\approx 12d^2$ parameters.

For $d = 4096$ (Llama-3 scale): $12 \times 4096^2 \approx 201M$ per block.

Two thirds of the parameters are in the FFN. Attention is the smaller component.

Counting Parameters: Full Model

Full model: L blocks + embedding + output head

$$\text{Total} \approx \underbrace{L \times 12d^2}_{\text{transformer blocks}} + \underbrace{|\text{vocab}| \times d}_{\text{embedding}} + \underbrace{d \times |\text{vocab}|}_{\text{output layer}}$$

For large models, the $12Ld^2$ term dominates.

Model	Layers (L)	d	Heads	Params
GPT-2 Small	12	768	12	124M
GPT-2 XL	48	1600	25	1.5B
Llama-3 8B	32	4096	32	8B
Llama-3 70B	80	8192	64	70B
GPT-4 (est.)	120	12288	96	$\sim$1.8T

From Logits to Probabilities

The linear head outputs a vector of logits $z \in \mathbb{R}^{|\text{vocab}|}$.

Apply softmax to get a probability distribution:

$$p(w) = \frac{\exp(z_w)}{\sum_{w'} \exp(z_{w'})}$$

Example (vocab = {the, cat, mat, dog, sat, ...}):

- logits: $z_{mat} = 5.2$, $z_{cat} = 3.1$, $z_{dog} = 2.8$, ...
- After softmax: $p(\text{mat}) = 0.62$, $p(\text{cat}) = 0.08$, $p(\text{dog}) = 0.05$, ...

During training, we compare this distribution to the true next token using cross-entropy loss.
During generation, we sample from this distribution.

1 The Transformer Architecture

2 Training

3 Generation

Cross-Entropy Loss for Language Modeling

We have a chunk of tokens: $y_1, y_2, \ldots, y_T$

At every position i (from 1 to $T - 1$):

- Input: $y_1, \ldots, y_i$
- Target: y_{i+1}
- Loss at position i: $-\log p(y_{i+1} \mid y_1, \ldots, y_i)$

Total loss = average over ALL positions:

$$\mathcal{L} = -\frac{1}{T-1} \sum_{i=1}^{T-1} \log p(y_{i+1} \mid y_1, \ldots, y_i)$$

We predict the next token at every position, not just the last one. This is why causal masking matters: position i can only see positions 1 to i.

Cross-Entropy Loss: Concrete Example

Chunk: [The, cat, sat, on] $(T = 4)$

Position i	Sees	Predicts	Loss
1	The	cat	$-\log p(\text{cat} \mid \text{The})$
2	The, cat	sat	$-\log p(\text{sat} \mid \text{The, cat})$
3	The, cat, sat	on	$-\log p(\text{on} \mid \text{The, cat, sat})$

$$\mathcal{L} = \frac{1}{3}\left[-\log p(\text{cat} \mid \text{The}) - \log p(\text{sat} \mid \text{The,cat}) - \log p(\text{on} \mid \text{The,cat,sat})\right]$$

All three predictions happen in one forward pass thanks to causal masking.

Why Every Position Matters

A common misconception: "the model only predicts the last token."

No. The loss sums over all positions in the chunk.

Given chunk $[y_1, \ldots, y_T]$ with $T = 1024$:

- We get 1023 predictions from a single forward pass
- Position 1 predicts y_2 from just y_1
- Position 512 predicts y_{513} from $y_1, \ldots, y_{512}$
- Position 1023 predicts y_{1024} from all previous tokens

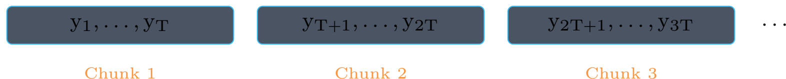
This is extremely efficient: one forward pass gives us 1023 training signals.

The Training Data

Step 1: Tokenize entire corpus → one long sequence $y_1, y_2, \ldots, y_N$

(N can be billions or trillions of tokens)

Step 2: Chunk into non-overlapping windows of length T (context length):



A few chunks is one training batch. Shuffle batch order (NOT TOKENS INSIDE!!).

Context length T is a hyperparameter:

- GPT-2: 1024 Llama-3: 8192 GPT-4: 128K

What Does the Data Look Like?

Pre-training corpora are massive, diverse, and carefully curated:

Source	Example	Fraction
Web crawl	Wikipedia, blogs, forums	$\sim$70%
Code	GitHub repositories	$\sim$10%
Books	Scanned books, ebooks	$\sim$10%
Academic	ArXiv, PubMed	$\sim$5%
Other	Social media, news	$\sim$5%

The model learns language, code, math, science, history, etc. all from one objective: predict the next token.

Data Quality Matters More Than Quantity

Raw web crawls are full of garbage: duplicates, spam, boilerplate, toxic content.

A real data pipeline (e.g., Llama, FineWeb):

1. URL filtering: block known spam domains, adult content
2. Text extraction: strip HTML, ads, navigation menus
3. Language detection: keep only target languages
4. Quality filtering: perplexity filter (discard text a trained model finds nonsensical), length filter, character ratio checks
5. Deduplication: exact and near-duplicate removal (MinHash). Reduces CommonCrawl by $\sim$50%
6. PII scrubbing: remove emails, phone numbers, SSNs
7. LLM Slop Scrubing: Remove LLM slop/yap.
8. Mixing: blend sources at specific ratios (upweight books, code)

Data engineering is a huge part of LLM training.

Mixed Precision Training

Mixed precision:

- Modern GPUs have specialized hardware for 16-bit math (2x faster than 32-bit)
- Forward/backward pass in bfloat16, optimizer states in float32
- PyTorch: torch.autocast('cuda', dtype=torch.bfloat16)

Quantization (for inference):

- "4-bit quantized" $\rightarrow$ weights stored in 4 bits instead of 16

Format	Bits/param	7B model size
float32	32	28 GB
bfloat16	16	14 GB
int8	8	7 GB
int4	4	3.5 GB

The Training Recipe

Key ingredients:

- Optimizer: AdamW
- LR schedule: warmup + cosine decay
- Gradient accumulation: simulate large batches on limited GPU memory
- Gradient clipping: cap gradient norms to prevent spikes
- Epochs: usually just 1-3 passes through the data (for large corpora)

Gradient accumulation:

- Want effective batch of 128 sequences, but only 8 fit on each GPU
- Run 4 micro-batches, accumulate gradients, then step
- 4 GPUs $\times$ 8 per GPU $\times$ 4 accumulation steps = 128 effective batch

You will use this in the HW.

Training at Scale

Model	Data	Compute	Year
GPT-2	40 GB text	256 GPUs, ~1 week	2019
Llama-2 70B	2T tokens	1.7M GPU-hours	2023
Llama-3 405B	15T tokens	30M GPU-hours	2024
GPT-4 (est.)	>10T tokens	100K+ GPUs, months	2023

Cost (rough estimates):

- GPT-2: ~$50K
- Llama-2 70B: ~$2M
- Llama-3 405B: ~$30M
- GPT-4: ~$100M+

A natural question: if you have a fixed compute budget, how should you split it between model size and data?

Scaling Laws: Kaplan et al. (2020)

OpenAI found that loss follows power laws in model size, data, and compute:

$$L(N) \propto N^{-0.076}, \quad L(D) \propto D^{-0.095}, \quad L(C) \propto C^{-0.050}$$

- N = number of parameters, D = number of tokens, C = compute (FLOPs)
- Log-log plots are straight lines over many orders of magnitude

Key insight: performance improves predictably with scale. No plateaus in sight.

Kaplan's recommendation: scale the model fast, use relatively less data. This turned out to be wrong.

Chinchilla: Compute-Optimal Training (2022)

DeepMind's Chinchilla paper (Hoffmann et al., 2022) showed Kaplan was off.

The compute-optimal rule:

$$C \approx 6ND$$

where C = FLOPs, N = parameters, D = training tokens.

For a fixed compute budget, the optimal split is roughly:

$$D \approx 20N$$

20 tokens per parameter.

Model size	Optimal tokens	Compute
1B params	20B tokens	1.2×10^{20} FLOPs
7B params	140B tokens	5.9×10^{21} FLOPs
70B params	1.4T tokens	5.9×10^{23} FLOPs

Scaling Laws: Modern Caveats

Chinchilla optimizes for training compute. But that's not the only cost.

Inference cost matters too:

- A 70B model costs 7$\times$ more per query than a 10B model
- If you serve billions of queries, inference dominates total cost
- Better to overtrain a smaller model (more data than Chinchilla says)

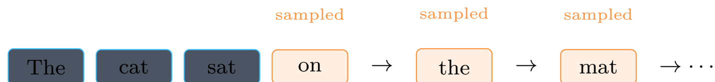
Llama-3 8B: trained on 15T tokens. Chinchilla says 160B would be optimal. That's 94x more data than Chinchilla-optimal. But the resulting 8B model is cheap to serve and very strong.

Takeaway: Chinchilla gives a useful baseline, but real decisions depend on your deployment scenario. If inference is cheap (few queries), follow Chinchilla. If inference is expensive (millions of users), overtrain a smaller model.

1. The Transformer Architecture
2. Training
3. Generation

Autoregressive Generation

The idea: model predicts p(next | context). Sample a token. Append. Repeat.



This is inherently sequential: each new token depends on all previous ones.

Training: process all positions in parallel (teacher forcing).
Generation: must go one token at a time.

This asymmetry is why training is much faster (per token) than generation.

Greedy vs. Sampling

Greedy decoding: always pick the highest-probability token.

- Deterministic: same input always gives same output
- Often repetitive: “The cat sat on the mat. The cat sat on the mat. The cat...”

Sampling: draw from the probability distribution.

- Stochastic: different output each time
- More diverse and interesting text
- But sometimes incoherent: might sample a very unlikely token

We need ways to control the randomness: temperature and top-p.

Temperature

Divide logits by T before softmax:

$$p_i = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)}$$

- $T = 1$: standard distribution
- $T < 1$: sharper (more confident, more repetitive)
- $T > 1$: flatter (more random, more creative)

Example: logits [3.0, 1.0, 0.5]

T	p_1	p_2	p_3
0.5	0.98	0.02	0.01
1.0	0.82	0.11	0.07
2.0	0.60	0.22	0.17

Top-p (Nucleus) Sampling

Problem: even with temperature, the model might sample very unlikely tokens.

Top-p sampling (Holtzman et al., 2020):

1. Sort tokens by probability (descending)
2. Take the smallest set whose cumulative probability $\geq$ p
3. Re-normalize and sample from this set

Example: $\mathrm{p} = 0.9$, probabilities $[0.5, 0.3, 0.1, 0.05, 0.03, 0.02]$

- Cumulative: $0.5, 0.8, 0.9, \ldots$
- Keep first 3 tokens (cumulative $= 0.9$). Discard the rest.
- Re-normalize: $[0.556, 0.333, 0.111]$ and sample.

$\mathrm{p} = 1.0$: sample from full distribution. $\mathrm{p} = 0.9$: ignore the long tail.

Example Outputs

Prompt: "Once upon a time"

T = 0.3 (low temperature, very focused):

"Once upon a time there was a little girl named Lily. She liked to play with her toys. One day, she went to the park."

T = 1.0 (standard):

"Once upon a time there was a brave knight who discovered a hidden library under the castle. The books could talk and each one told a different story."

T = 2.0 (high temperature, chaotic):

"Once elephant dancing quarterly the suspicious marble fountain echoing butterfly sand..."

Temperature controls the creativity-coherence tradeoff.

Temperature and Top-p: Working Together

In practice, you use both temperature and top-p:

Use case	T	top-p
Code generation	0.2	0.95
Chat / dialogue	0.7	0.9
Creative writing	1.0	0.95
Brainstorming	1.2	1.0

- Temperature controls the shape of the distribution
- Top-p truncates the tail
- Together they give fine-grained control over generation quality

Stopping Generation

When does the model stop generating?

- Max tokens: stop after generating N tokens (API-enforced budget)
- EOS token: the tokenizer has a special <|endoftext|> token. When sampled, stop.
- Stop strings: stop when a specific pattern appears (e.g., \n\n)

You've probably seen errors like:

API Error: response exceeded the 32000 output token maximum

This is an API-side limit, not a hard architectural wall. The provider caps generation to control costs. The model could technically keep going.

Context Length: Why Can't We Just Keep Going?

Attention is variable-length. Nothing in $\mathrm{softmax}(QK^\top/\sqrt{d_k})V$ needs a fixed T.

The real bottleneck is positional encoding:

Learned positional embeddings (GPT-2): the model has embeddings for positions $1, \ldots, L$. Beyond L, there is literally no embedding. Hard stop.

RoPE (Llama, Mistral): the rotation formula works at any position. No hard stop. But the model was trained on positions up to L. Beyond that, it degrades.

Why? Context length is a training distribution problem:

- The model only learned attention patterns across windows of size $\leq L$
- It only learned to locate relevant evidence within L tokens
- Going beyond L is out-of-distribution, like asking someone trained to read 10 pages to reason over 10,000 pages. This is why extending context length (RoPE scaling, YaRN, etc.) is an active research area