AIE1902: Exploring Language Models with AI

Xiaopeng Li
CUHK(SZ)
SAI

March 16, 2026

Outline

From Unconditional to Conditional Language Models

Self-Attention

The Transformer Architecture

Tokenization: WordPiece

GPT-1: Decoder-Only Transformer

Recap: Unconditional Language Models

RNN & LSTM models the probability of a sequence

$$p(y_1, y_2, \ldots, y_T) = \prod_{t=1}^{T} p(y_t \mid y_1, \ldots, y_{t-1}).$$

Hidden state h_t carries information from the past.

Decoding: Given a trained model, we generate text via

- **Greedy decoding:** always pick $\arg\max_y \, p(y \mid y_1, \ldots, y_{t-1})$
- **Beam search:** keep top-k candidates at each step

Unconditional: the model produces text from scratch or a start token.

New question: what if we want to translate?

$$\underbrace{\text{我} \quad \text{喜欢} \quad \text{猫}}_{\text{input}} \Longrightarrow \underbrace{\texttt{I like cats}}_{\text{output}}$$

We need a model that generates text conditioned on an input sequence:

$$p(y_1, \ldots, y_T \mid x_1, \ldots, x_S).$$

Encoder-Decoder: Conditional Language Models

Idea: use two RNNs working together.

- **Encoder:** reads input $(x_1, \ldots, x_S)$, produces hidden states $h_1^{\text{enc}}, \ldots, h_S^{\text{enc}}$.
- **Decoder:** generates output $(y_1, \ldots, y_T)$ one token at a time, initialized with $h_0^{\text{dec}} = h_S^{\text{enc}}$.

Conditional: h_0^{dec} comes from the encoder instead of being $\mathbf{0}$.

$$p(y_t \mid y_1, \ldots, y_{t-1},\ x_1, \ldots, x_S) = \text{softmax}(W_y\, h_t^{\text{dec}} + b_y).$$

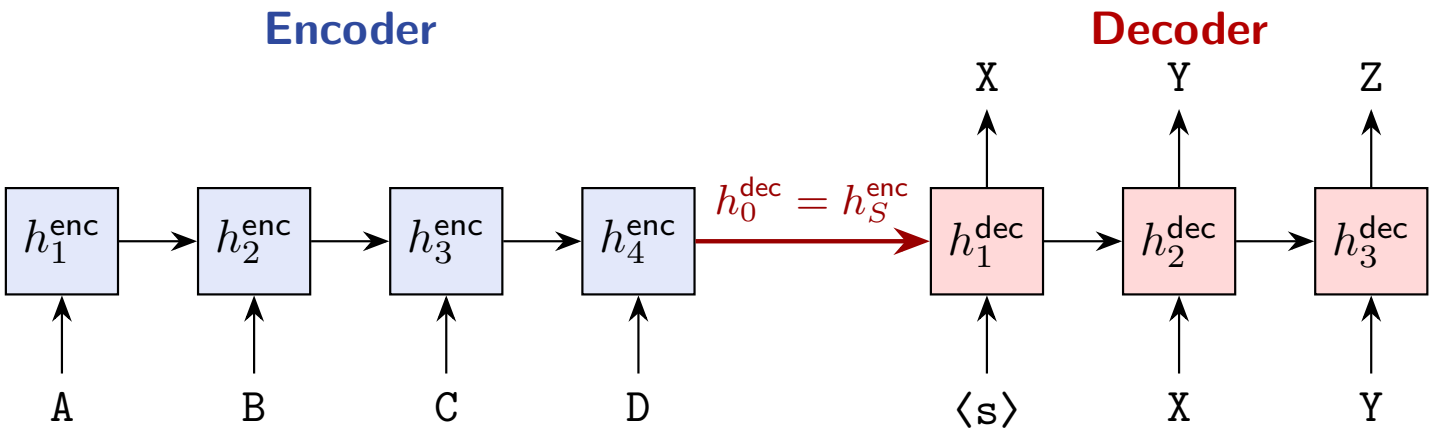
Training: same cross-entropy loss, but backpropagate through *both* decoder and encoder:

$$\mathcal{L} = -\frac{1}{T} \sum_{t=1}^{T} \log p(y_t \mid y_1, \ldots, y_{t-1},\ x_1, \ldots, x_S).$$

Terminology:

- **Language model**: models $p(y_1, \ldots, y_T)$, unconditional;
- **Seq2seq model**: models $p(y_1, \ldots, y_T \mid x_1, \ldots, x_S)$, conditional.

Encoder-Decoder: Architecture



Encoder: same RNN architecture, but no softmax, no loss, just produces representations. Thus, it is not a language model.

Decoder: same RNN language model as last lecture. The only new thing is where h_0 comes from.

Questions: The red arrow is the entire connection between encoder and decoder. Is this enough?

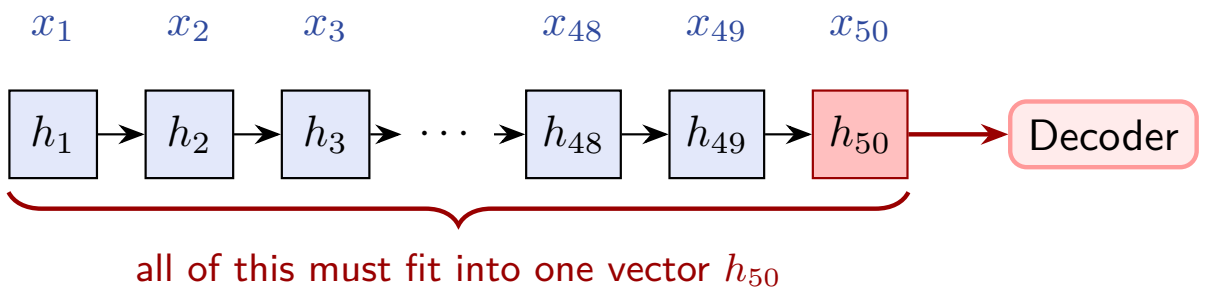
The Bottleneck Problem

Problem: all input must pass through a single vector $h_S^{\text{enc}} \in \mathbb{R}^d$.

Example: suppose we want to translate a 50-word sentence.

- The encoder processes all 50 words sequentially.
- By the end, h_{50}^{enc} must "remember" word 1, word 2, ..., word 50.
- But h_{50}^{enc} is a fixed-size vector (e.g., $d = 256$).

Problem: you already know from last lecture that RNNs suffer from vanishing gradients, so information from early time steps fades away.



Idea: what if the decoder could look back at all encoder hidden states $h_1, \ldots, h_S$, not just the last one?

Attention Mechanism

Setup: encoder produces hidden states $\bar{h}_1, \ldots, \bar{h}_S$. At decoder time step t, we have decoder hidden state h_t.

Score. Compute relevance between each encoder state and current decoder state:

$$\text{score}_{t,s} = w^\top \tanh(W_1\, h_t + W_2\, \bar{h}_s), \quad s = 1, \ldots, S.$$

Align. Apply softmax to get attention weights:

$$a_{t,s} = \frac{\exp(\text{score}_{t,s})}{\sum_{j=1}^{S} \exp(\text{score}_{t,j})}.$$

Context. Form a weighted sum of encoder states:

$$c_t = \sum_{s=1}^{S} a_{t,s}\, \bar{h}_s.$$

Predict. Combine context with decoder state to predict next token:

$$\tilde{h}_t = [c_t,\ h_t], \qquad y_t = \text{softmax}(W_y\, \tilde{h}_t + b_y).$$

Idea: at each step, the decoder chooses which input words to focus on. No more bottleneck!

Attention is a General Mechanism

General Pattern: every attention mechanism does three things:

1. Compute scores between **queries** Q and **keys** K.
2. Apply softmax to get weights.
3. Return weighted sum of **values** V.

Compact Formula:

$$\text{Attention}(Q, K, V) = \text{softmax}(\text{score}(Q, K))V.$$

Difference is where Q, K, V come from:

	Q	K, V	**Purpose**
Additive-att.	dec. h_t	enc. $\bar{h}_s$	connect two seq.
Self-att. (later)	same sequence		context in one seq.
Cross-att. (later)	decoder	encoder	connect two seq.

The Parallelism Problem

Even with attention, we still have RNNs underneath.

Sequential processing: in an RNN, we *must* compute states in order:

$$h_0 \to h_1 \to h_2 \to h_3 \to \cdots \to h_T.$$

Consequence for training:

- RNN effective batch size = number of sequences.
- Each sequence is processed one token at a time, cannot parallelize across time steps.
- GPUs are good at parallelization, but RNNs cannot benefit!

Scalability: bigger models + more data $\implies$ better performance.
But RNNs are too slow to train at scale.

Question: can we get rid of the RNN and use only attention?

Yes! $\implies$ **The Transformer.**

Outline

Self-Attention: Query, Key, Value

Recall: attention needs queries Q, keys K, values V.

In self-attention, all three come from the same sequence $\{x_1, \ldots, x_T\}$.

Use learned matrices W^Q, W^K, W^V to project each token embedding:

$$q_t = W^Q x_t, \qquad k_t = W^K x_t, \qquad v_t = W^V x_t.$$

Intuition:

- q_t: "what am I looking for?"
- k_t: "what do I contain?"
- v_t: "what information do I provide if selected?"

Example: `The animal didn't eat because it was tired.`

- The query of "`it`" asks: *who am I referring to?*
- The key of "`animal`" answers: *I'm a noun, a candidate referent.*
- High score $q_{\texttt{it}}^\top k_{\texttt{animal}} \implies$ "`it`" attends strongly to "`animal`".

Scaled Dot-Product Attention

Score function: for one query q_t and all keys, use dot product:

$$\text{score}_{t,s} = q_t^\top k_s.$$

Problem: if $q, k \in \mathbb{R}^{d_k}$ with components $\sim N(0,1)$, then

$$\mathbb{E}[q^\top k] = 0, \qquad \text{Var}(q^\top k) = d_k.$$

Large $d_k \implies$ large variance $\implies$ softmax becomes extremely peaked.

Example: suppose we have 4 keys.

	Raw scores	Softmax
$d_k = 4$ (small var.)	$[1.2,\ -0.5,\ 0.8,\ -0.3]$	$[0.40,\ 0.07,\ 0.27,\ 0.09]$
$d_k = 64$ (large var.)	$[9.5,\ -4.2,\ 6.1,\ -2.8]$	$[0.97,\ 0.00,\ 0.03,\ 0.00]$
$d_k = 64$ (normalized)	$[1.2,\ -0.5,\ 0.8,\ -0.4]$	$[0.49,\ 0.09,\ 0.32,\ 0.10]$

When d_k is large, almost all weight goes to one token, and the model cannot learn from other positions.

Fix: divide by $\sqrt{d_k}$ to normalize variance to 1: $\text{score}_{t,s} \leftarrow \text{score}_{t,s}/\sqrt{d_k}$.

Scaled Dot-Product Attention: Full Formula

One query: compute scaled scores, softmax, weighted sum of values:

$$\text{output}_t = \sum_{s=1}^{T} \underbrace{\text{softmax}\left(\frac{q_t^\top k_s}{\sqrt{d_k}}\right)}_{a_{t,s}} v_s.$$

Matrix form: all T queries at once,

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

where $Q, K \in \mathbb{R}^{T \times d_k}$, $V \in \mathbb{R}^{T \times d_v}$.

Compare with RNNs:

- RNN: must compute $h_1 \to h_2 \to \cdots \to h_T$ sequentially.
- Self-attention: one matrix multiplication processes all tokens simultaneously.

Multi-Head Attention

Problem: with one set of W^Q, W^K, W^V, the model can only learn *one* type of relationship.

Idea: use h different sets $\{W_i^Q, W_i^K, W_i^V\}_{i=1}^h$, each learning a pattern!

$$\text{head}_i = \text{Attention}(XW_i^Q,\ XW_i^K,\ XW_i^V), \quad i = 1, \ldots, h.$$

Concatenate all heads and project back:

$$\text{MultiHead}(X) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)\, W^O.$$

What might different heads learn?

- Head 1: syntactic — "`it`" $\rightarrow$ "`animal`" (coreference)
- Head 2: positional — attend to neighboring words
- Head 3: semantic — "`tired`" $\rightarrow$ "`didn't eat`" (cause-effect)
- $\ldots$

Dimensions: if $d_{\text{model}} = 512$ and $h = 8$ heads:

- Each $W_i^Q, W_i^K, W_i^V \in \mathbb{R}^{512\times 64}$, so $d_k = d_v = d_{\text{model}}/h = 64$.
- Each $\text{head}_i \in \mathbb{R}^{T\times 64}$, concatenation $\in \mathbb{R}^{T\times 512}$.
- $W^O \in \mathbb{R}^{512\times 512}$ projects back to d_{model}.

Outline

Positional Encoding

Problem: self-attention is permutation-invariant.

`man bites dog` and `dog bites man`

produce the *same* attention output! Scores doesn't encode order.

Fix: add a **positional encoding** p_t to each token embedding:

$$x_t \leftarrow x_t + p_t.$$

How? The original Transformer uses sinusoidal functions:

$$p_{t,2j} = \sin\left(\frac{t}{10000^{2j/d_{\text{model}}}}\right), \qquad p_{t,2j+1} = \cos\left(\frac{t}{10000^{2j/d_{\text{model}}}}\right).$$

- Each dimension j oscillates at a different frequency.
- The model can learn to use these patterns to determine relative positions.

Residual Connections and Layer Normalization

Problem: stacking many layers makes training difficult because gradients can vanish or explode.

Residual connection: instead of computing output $= f(x)$, compute

$$\text{output} = x + f(x).$$

The input x has a shortcut path that skips the layer entirely.

Why? Even if $\frac{\partial f}{\partial x}$ is small, the gradient is at least $I \rightarrow$ no vanishing!

Layer normalization: normalize activations across the feature dimension:

$$\text{LayerNorm}(z) = \gamma \odot \frac{z - \mu}{\sigma + \epsilon} + \beta,$$

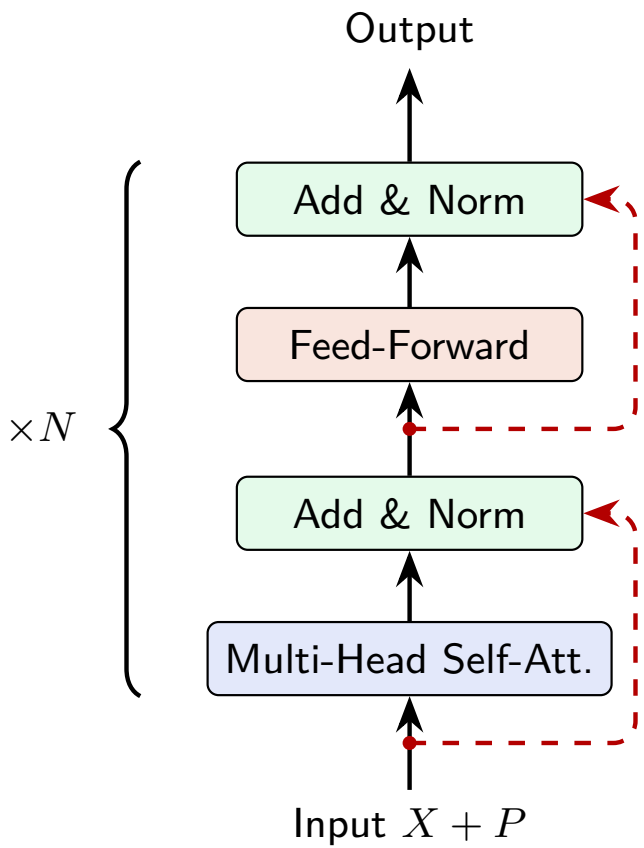
where μ, σ are mean and std of z, and γ, β are learned parameters.

Combined: used after every sublayer in the Transformer:

$$\text{output} = \text{LayerNorm}(x + f(x)).$$

The Encoder Block

One encoder layer: two sublayers, each with Add & Norm.



Sublayer 1: Multi-Head Self-Attention

$$X \leftarrow \mathrm{LN}(X + \mathrm{MH}(X))$$

Sublayer 2: Feed-Forward Network

$$X \leftarrow \mathrm{LN}(X + \mathrm{FF}(X))$$

where

$$\mathrm{FF}(x) = W_2\,\mathrm{ReLU}(W_1 x + b_1) + b_2$$

Projects: $d_{\text{model}} \to 4d_{\text{model}} \to d_{\text{model}}$.

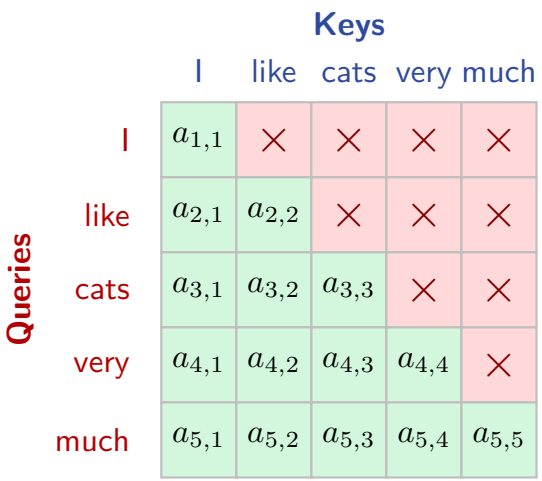
Repeat N times (original: $N = 6$).

Dashed red arrows = **residual connections**.

The Decoder: Masked Self-Attention

Problem: in a language model, we should predict y_t using only $y_1, \ldots, y_{t-1}$. We must not look ahead! Self-attention lets every token attend to every other token, including future ones.

Fix: set $\text{score}_{t,s} = -\infty$ for $s > t$ so that future token get 0 weight.



Reading the mask:

- Row = query, who is asking
- Column = key, who is being attended to
- Green : allowed, past and self
- Red × : blocked, future

Example:

- "cats" can attend to "I", "like", "cats"
- "cats" cannot attend to "very", "much"

The Decoder: Cross-Attention

After masked self-attention on its own tokens, the decoder attends to the encoder's output, just like additive attention, but without RNNs.

Queries come from the decoder, after masked self-attention:

$$Q = Y_{\text{masked-SA}}\, W^Q$$

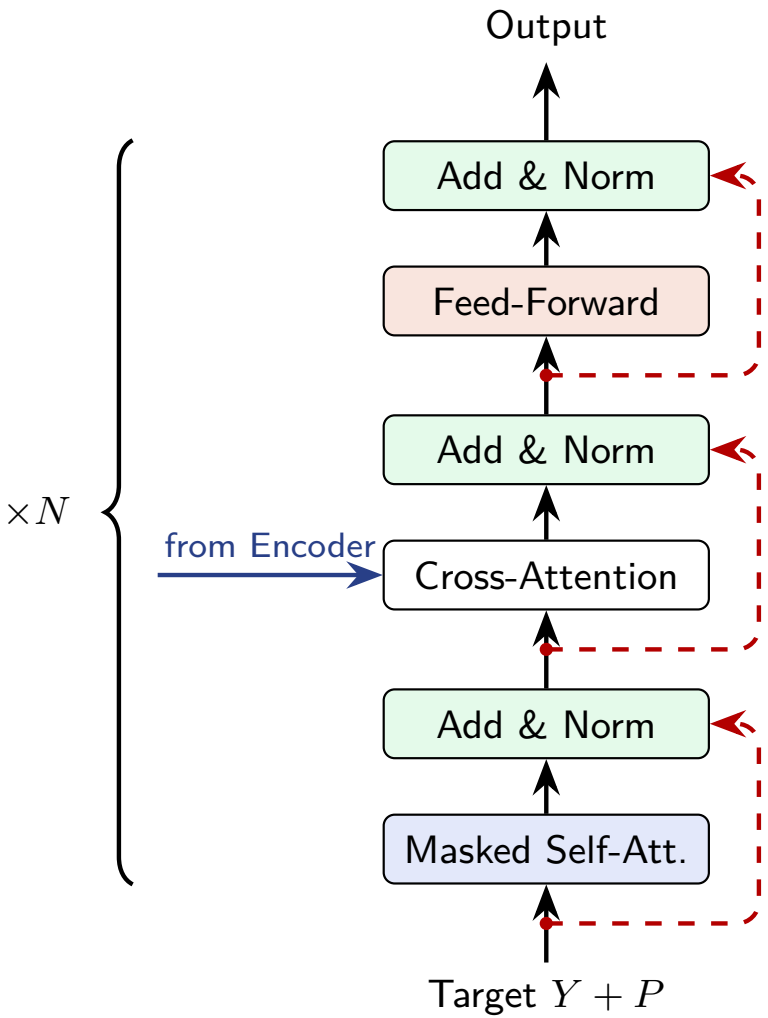
Keys and Values come from the encoder output:

$$K = X_{\text{enc}}\, W^K, \quad V = X_{\text{enc}}\, W^V$$

Then apply the same scaled dot-product attention:

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

Full decoder layer

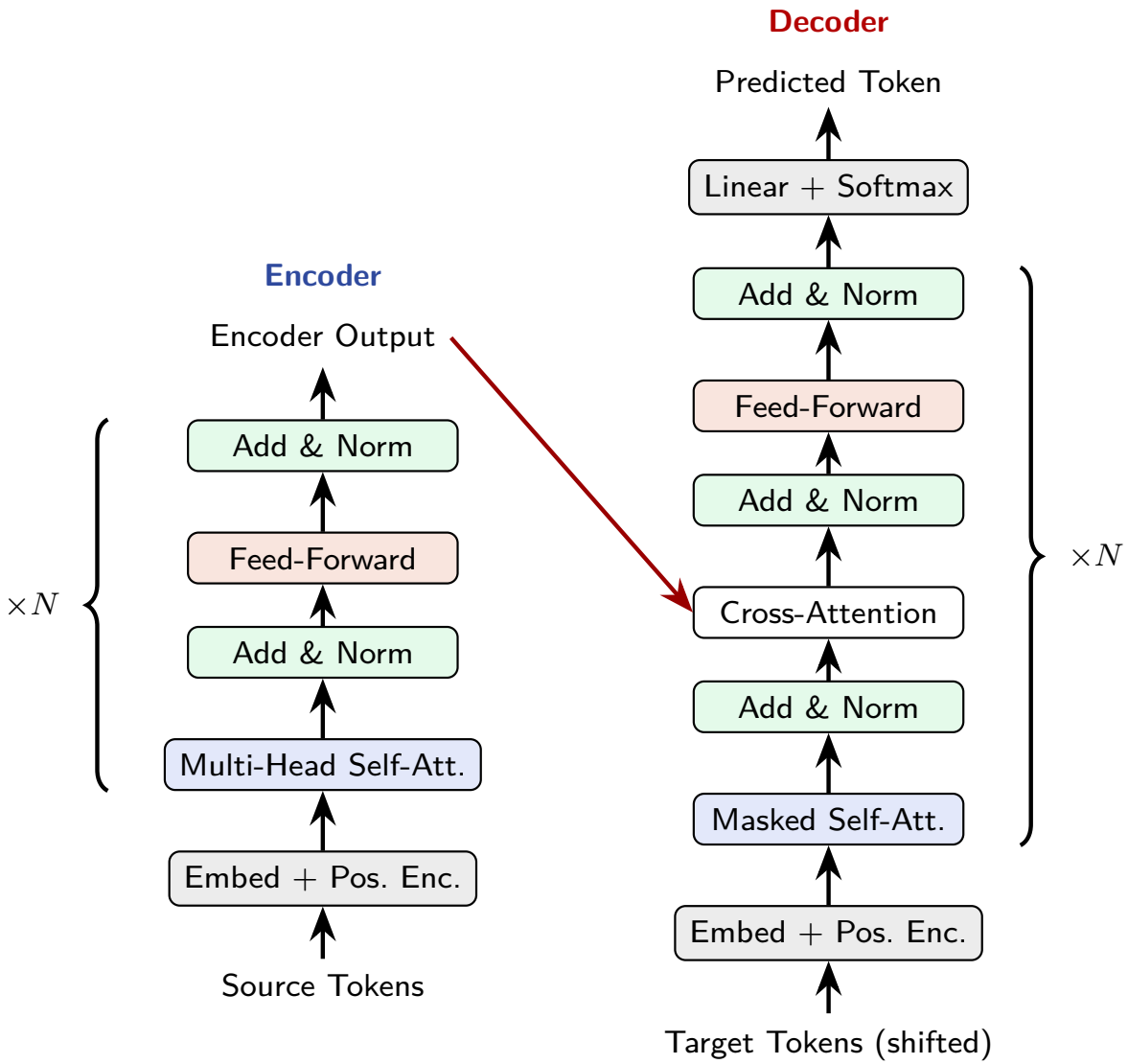


Three sublayers:

1. **Masked self-att:** attend to own past tokens
$$Y \leftarrow \mathsf{LN}(Y + \mathsf{MaskedMH}(Y))$$
2. **Cross-att:** attend to encoder output; Q from decoder, K, V from encoder
$$Y \leftarrow \mathsf{LN}(Y + \mathsf{CrossMH}(Y, X_{\mathsf{enc}}))$$
3. **Feed-forward:** same as encoder
$$Y \leftarrow \mathsf{LN}(Y + \mathsf{FF}(Y))$$

Compare: encoder has 2 sublayers, decoder has 3.

Putting It All Together



Transformer: Benefits and Drawbacks

Benefits:

- $O(1)$ sequential operations, full parallelism.
- $O(1)$ maximum path length between any two tokens.
- Can train much larger models much faster.

Drawbacks:

- Self-attention is $O(T^2)$ in memory and compute.
- Context window T must be fixed upfront.
- Requires substantial compute for large models.

	Sequential Ops	**Max Path Length**	**Complexity/Layer**
RNN	$O(n)$	$O(n)$	$O(n \cdot d^2)$
Self-Att.	$O(1)$	$O(1)$	$O(n^2 \cdot d)$

Takeaway: for most NLP, $n \ll d$, self-attention is cheap. For very long sequences, the $O(n^2)$ cost becomes expansive.

Outline

Why Not Just Words or Characters?

Word-level tokens: like Word2Vec

- Vocabulary is huge: 100K+ words in English.
- Any new or rare word becomes `[UNK]`: can't handle "unfriending", "ChatGPT", etc.

Character-level tokens:

- Every word can be spelled out.
- But sequences become very long: "cats" → `c`, `a`, `t`, `s`.
- Individual characters carry little semantic meaning.

A middle ground: subword tokenization.

- Common words stay whole: "the", "cat" → single tokens.
- Rare words split into meaningful pieces: "unhappiness" → `un`, `happi`, `ness`.
- Vocabulary size is manageable: 30K–50K tokens.

WordPiece: Training

Algorithm: start with individual characters, iteratively merge the highest-scoring pair.

$$\text{score}(t_1, t_2) = \frac{\text{freq}(t_1, t_2)}{\text{freq}(t_1) \cdot \text{freq}(t_2)}$$

Example corpus: [“hug”, “pug”, “pun”, “bun”, “hugs”]

Initial vocab: [`h`, `##u`, `##g`, `p`, `##n`, `b`, `##s`].

Pair (t_1, t_2)	$\text{freq}(t_1, t_2)$	$\text{freq}(t_1)$	$\text{freq}(t_2)$	**Score**
(h, ##u)	2	2	5	$2/10 = 0.20$
(p, ##u)	2	2	5	$2/10 = 0.20$
(b, ##u)	1	1	5	$1/5 = 0.20$
(##u, ##g)	3	5	3	$3/15 = 0.20$
(##u, ##n)	2	5	2	$2/10 = 0.20$
(##g, ##s)	1	3	1	$1/3 = 0.33$

Merge (##g, ##s) → `##gs`, add it into vocab and continue until vocab size limit is reached.

WordPiece: Encoding

Why did (##g, ##s) win over (##u, ##g) which has freq $= 3$?

- What we care is how often they co-occur *relative to chance*?
- Whenever `##s` appears, it *always* follows `##g`: strong relation.

Encoding rule: greedily match the longest token from left to right.

Examples:

- "hug" $\rightarrow$ [`hug`] $\rightarrow$ whole word in vocab
- "hugs" $\rightarrow$ [`hug` , `##s`] $\rightarrow$ longest match "hug", then "##s"
- "pug" $\rightarrow$ [`p` , `##u` , `##g`] $\rightarrow$ no longer matches
- "bum" $\rightarrow$ [`[UNK]`] $\rightarrow$ cannot decompose

The original Transformer used WordPiece with 32,000 tokens.

Byte-Pair Encoding (BPE)

Similar to WordPiece, but simpler score: just raw frequency.

$$\text{score}(t_1, t_2) = \text{freq}(t_1, t_2)$$

Training: learn ordered merge rules.

1. ("u", "g") → "ug" (most frequent pair)
2. ("u", "n") → "un"
3. ("h", "ug") → "hug"
4. ...

Encoding (at inference time): apply merge rules in learned order.

- "bug" → [`b`, `ug`] (rule 1 applies: "u"+"g" → "ug")
- "thug" → [`[UNK]`, `hug`] (rules applied in order)
- "unhug" → [`un`, `hug`] (rule 2 then rule 3)

GPT-1 used BPE with 40,000 merges.

Outline

Do We Need Both Encoder and Decoder?

The original Transformer was **encoder-decoder** for translation.

But what if we just want a powerful unconditional language model?

$$p(y_1, \ldots, y_T) = \prod_{t=1}^{T} p(y_t \mid y_1, \ldots, y_{t-1})$$

Same goal as the RNN: just predict the next token.

Idea: we only need the decoder half.

- Keep: masked self-attention + feed-forward layers.
- Remove: the encoder and the cross-attention layer.

This is GPT-1 (Generative Pre-trained Transformer).

GPT-1: Architecture & Pretraining

Architecture:

- 12 Transformer decoder layers
- $d_{\text{model}} = 768$
- Feed-forward inner dimension: $4 \times 768 = 3072$
- 12 attention heads, $d_k = 64$
- $\sim$117M parameters

Pretraining data: BookCorpus ($\sim$7,000 books).

Pretraining objective: same cross-entropy loss as the RNN:

$$\mathcal{L} = -\frac{1}{T}\sum_{t=1}^{T} \log p(y_t \mid y_1, \ldots, y_{t-1};\ \theta)$$

Given "`The man walks fast`", predict "`man walks fast home.`"

Tokenization: BPE with 40,000 merges.

GPT-1: Fine-Tuning for Downstream Tasks

The pretrain-then-fine-tune paradigm:

1. **Pretrain** on large corpus (unsupervised, cheap): learn general language patterns $\rightarrow$ parameters θ.
2. **Fine-tune** on small labeled dataset (supervised): add a classifier head with new parameters ϑ, train with task-specific loss $\mathcal{L}_{\text{task}}$.

Minimal architecture changes, just reformat the input:

Task	Input Format
Classification	`[START]` text `[EXTRACT]` $\rightarrow$ Linear
Similarity	Run both orderings, add representations
Multiple Choice	Run each option separately, compare scores

Idea: the same pretrained model handles all tasks, only the input formatting and a small output head change.

GPT-1: Results & The Scaling Story

Results: GPT-1 achieved state-of-the-art on 9 of 12 benchmarks, with the same architecture and just different input formatting.

Core lesson: a large pretrained Transformer captures general language knowledge that transfers powerfully to many downstream tasks.

The scaling trajectory:

Model	Parameters	Data	Year
GPT-1	117M	BookCorpus	2018
GPT-2	1.5B	WebText (40GB)	2019
GPT-3	175B	Common Crawl + books	2020
GPT-4	???	???	2023

Same fundamental architecture. Bigger models + more data $\Longrightarrow$ dramatically better performance.

Key Takeaways

Encoder-decoder extends language models to conditional generation (translation); the decoder is a conditional language model.

Attention lets the decoder focus on relevant parts of the input, solving the bottleneck problem.

Self-attention replaces recurrence entirely, enabling parallelism.

The Transformer combines self-attention, cross-attention, residual connections, layer norm, and feed-forward layers together.

Tokenization (e.g. BPE) balances vocabulary size and coverage.

GPT-1 shows that a decoder-only Transformer, pretrained as a language model and fine-tuned, achieves much better performance.

Next Week

Thank you for your attendance!

Questions?

Next week (Lab session):

- Build a GPT-1 language model in PyTorch
- Implement masked self-attention, multi-head attention, and decoder blocks
- Train on Hemingway text with cross-entropy loss
- Generate "fake Hemingway" with your trained model
- Experiment with scaling: `d_model`, `n_head`, `n_layer`