Transformers (Part 2), Pretraining

Lucy He

2/27/26

Overview

- Transformers (continued)
- Contextual Word Embedding
- Pre-training technique
  - ELMo, BERT, GPT

Modeling Natural Language

Language Models: Assign a probability to any sequence of tokens.

e.g. p(dog) > p(turtle)

10^{-4} 10^{-6}

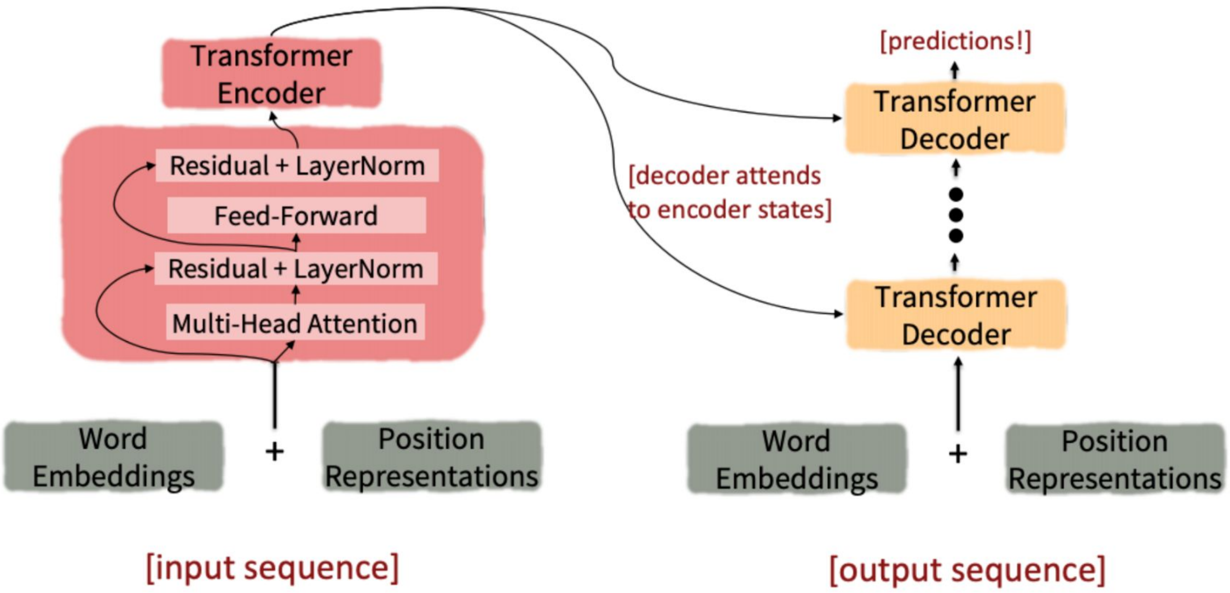
p(a turtle swims in the ocean) > p(a dog swims in the ocean)

10^{-11} 10^{-13}

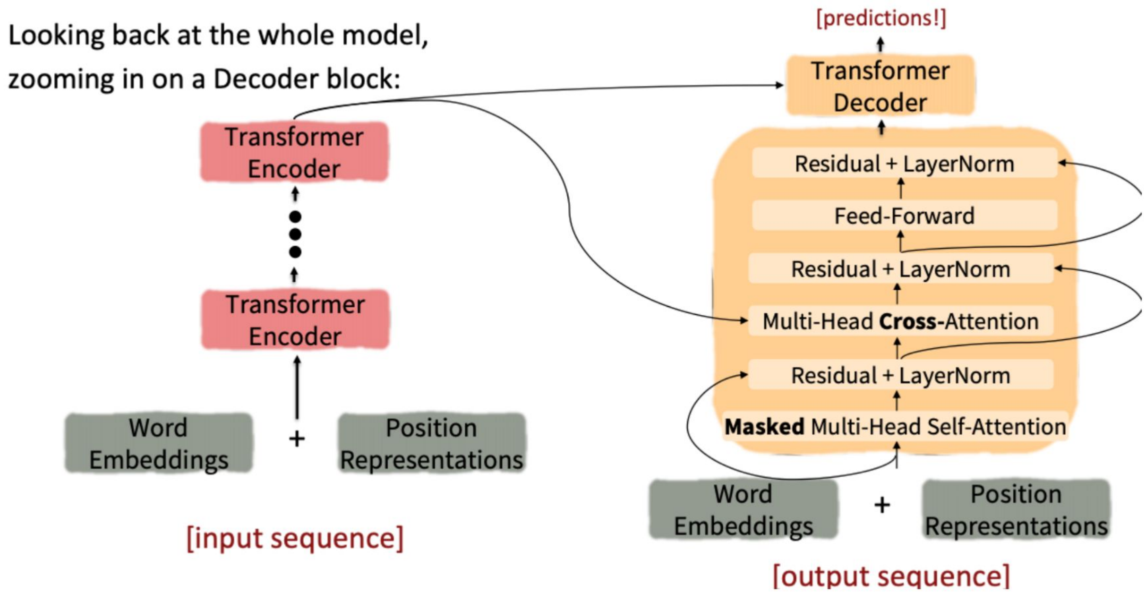
Left-to-right language models:

p(a turtle swims in the ocean) = p(a) p(turtle | a) p(swims | a turtle) p(in | a turtle swims) p(the | a turtle swims in) p(ocean | a turtle swims in the)

Transformer encoder-decoder (from class slides)



Transformer encoder-decoder (from class slides)



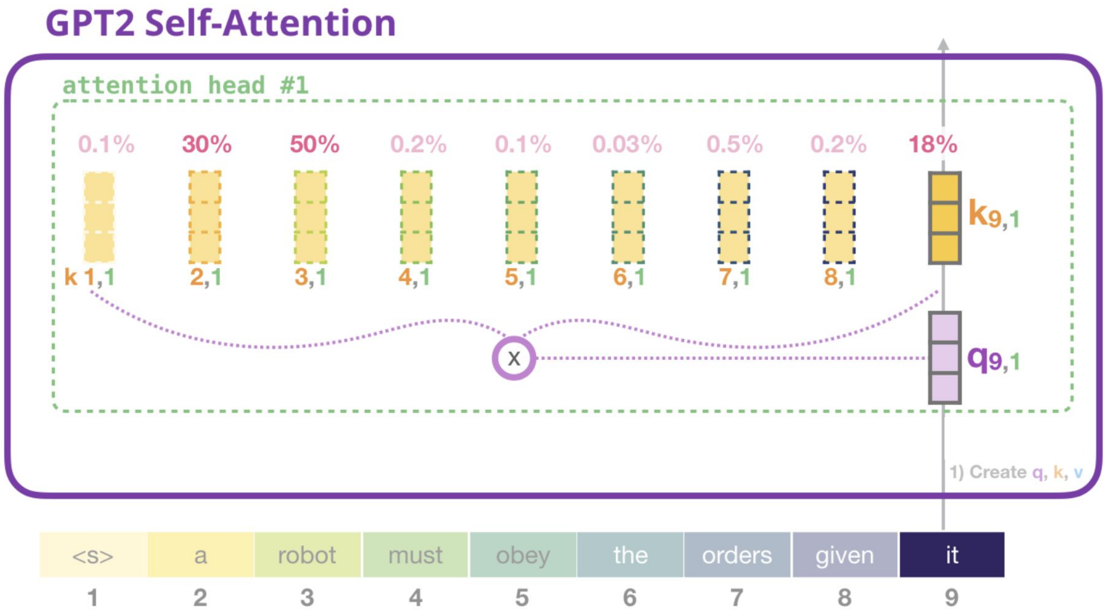
The Transformer Architecture: Embedding Layer

[from Alammar, The Illustrated GPT-2, https://jalammar.github.io/illustrated-gpt2/]

The Transformer Architecture: Attention Layer

⇒ Integrates information from previous tokens

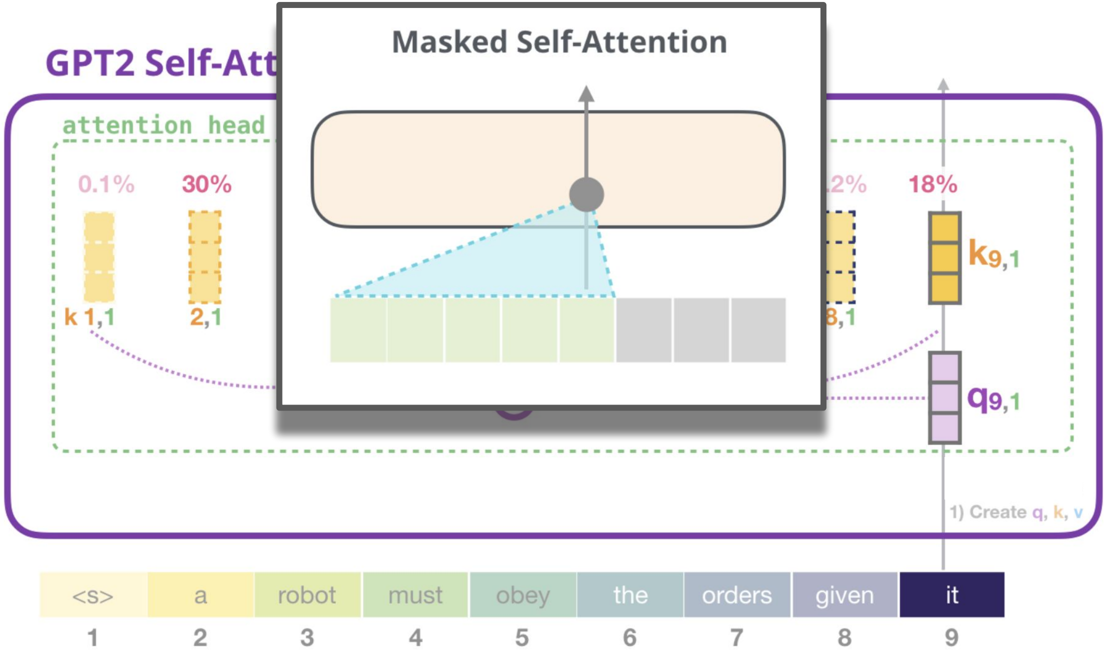
⇒ Can perform operations like *lookup* or *copy*



[from Alammar, The Illustrated GPT-2, https://jalammar.github.io/illustrated-gpt2/]

The Transformer Architecture: Attention Layer

⇒ Don't attend to future tokens



[from Alammar, The Illustrated GPT-2, https://jalammar.github.io/illustrated-gpt2/]

The Transformer Architecture: Masked Self-Attention

⇒ In the decoder, token *i* must not attend to future tokens *j* > *i*.

We add a mask **M** to the attention scores *before* softmax:

$$e_{i,j} = q_i \cdot k_j \,/\, \sqrt{d_k} + \mathbf{M}_{i,j}$$

where $\mathbf{M}_{i,j} = 0 \text{ if } j \leq i, \quad -\infty \text{ if } j > i$

After softmax, $e^{-\infty} \to 0$, so future tokens receive **zero weight**.
Key benefit: the full sequence can be processed in parallel during training — masking enforces causality without sequential computation.

Attention weight matrix (n = 5)

	the	cat	sat	on	mat
the	1.0	−∞	−∞	−∞	−∞
cat	.38	.62	−∞	−∞	−∞
sat	.18	.41	.41	−∞	−∞
on	.10	.28	.42	.20	−∞
mat	.07	.14	.34	.27	.18

allowed   masked (−∞)

The Transformer Architecture: Multi-Head Cross-Attention

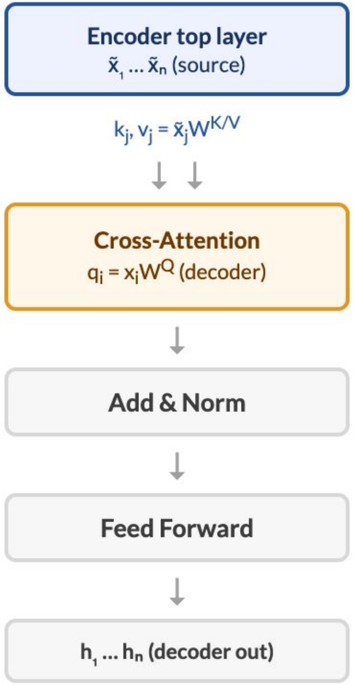
⇒ In an encoder-decoder model, the decoder needs to attend to the encoder's output.

> **Self-attention:** q_i, k_j, v_j all come from the **same sequence**
>
> **Cross-attention:** q_i from **decoder**, but k_j, v_j from **encoder top layer**
>
> $e_{i,j} = q_i \cdot k_j / \sqrt{d_k}$, $h_i = \Sigma_j \alpha_{i,j} v_j$

⇒ Decoder queries "look up" relevant encoder states for each output position

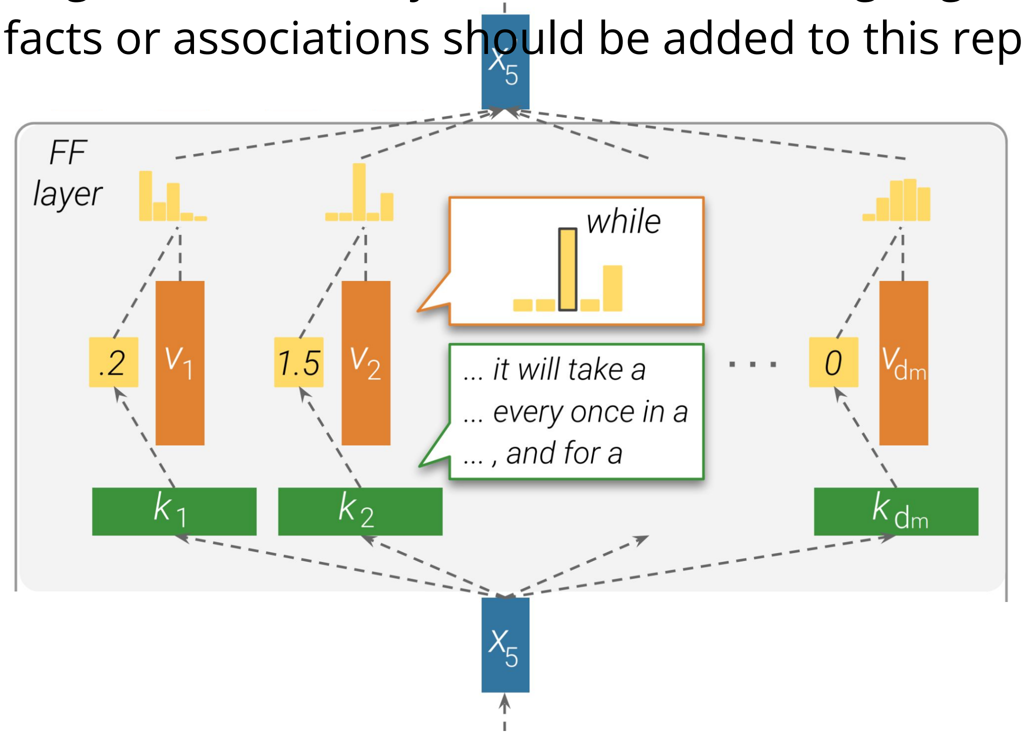
e.g., when generating "monde", query attends to "world" in the encoder

⇒ Each decoder layer has *both* masked self-attention (target seq) *and* cross-attention (source seq)



The Transformer Architecture: Feed-forward Layer

⇒ Store knowledge as a dictionary between embeddings. "given the current context, what facts or associations should be added to this representation."



The Transformer Architecture: Residual Connections & Layer Norms

Every sub-layer (attention, FFN) wraps its output in **Add & Norm**:

$$x' = \mathrm{LayerNorm}(\, x + \mathrm{Attention}(x)\,)$$

$$x'' = \mathrm{LayerNorm}(\, x' + \mathrm{FFN}(x')\,)$$

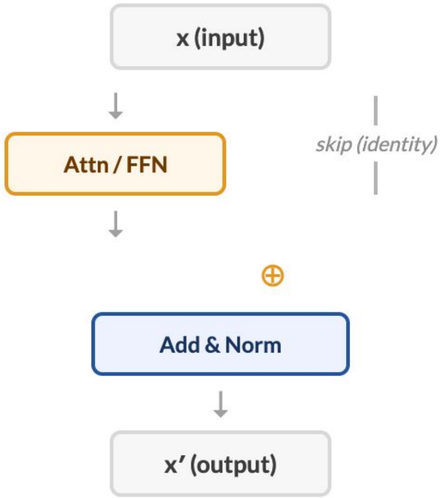
⇒ **Residual / skip connection** — gradient flows directly back through the "+1" term, preventing vanishing gradients across dozens of layers

∂L/∂x contains (1 + ∂Attn/∂x): the 1 always provides a clean gradient path

⇒ **Layer Norm** — normalizes each token's embedding to zero mean, unit variance across the feature dimension (not batch)

Stabilizes training; unlike BatchNorm, works with variable-length sequences

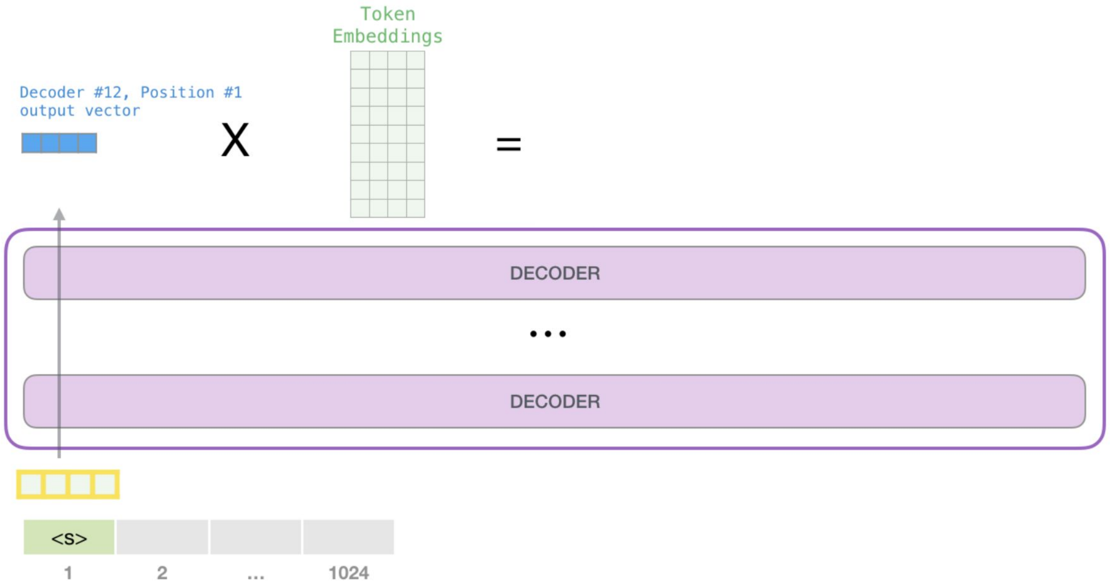
⇒ **Residual stream** view — each layer writes a small delta onto a shared "stream"; earlier information is never destroyed, just updated



Each of the two sub-layers in a Transformer block uses this pattern

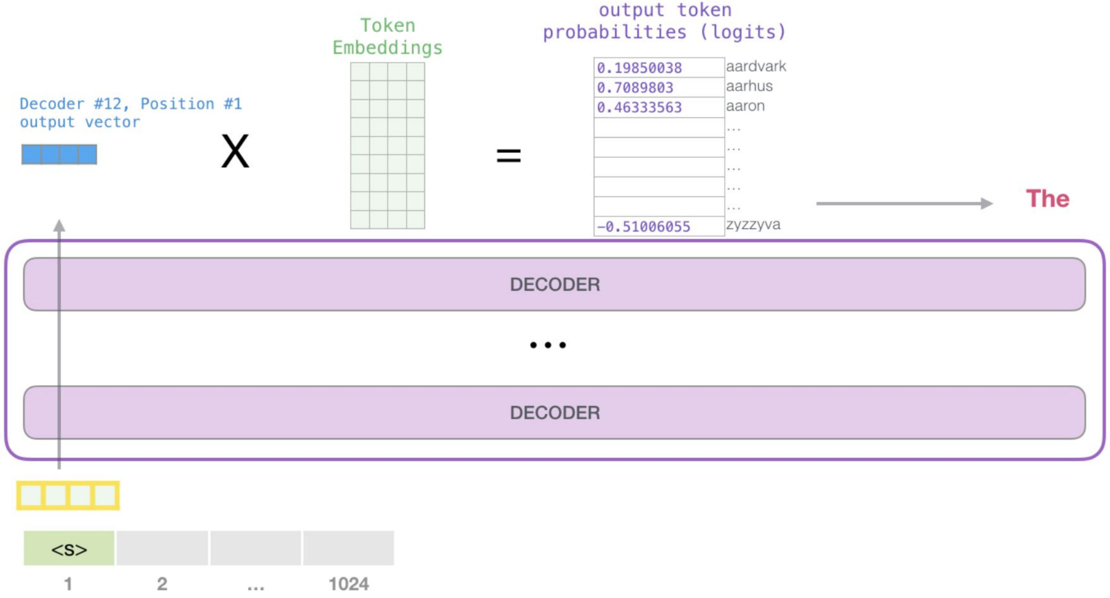
The Transformer Architecture: Prediction

1. Finally, compare the final embedding against all token embeddings



The Transformer Architecture: Prediction

1. Finally, compare the final embedding against all token embeddings
2. Obtain scores over next token candidates and convert to probabilities



The Transformer Architecture: Variants

Motivation: quadratic computation as a function of sequence length.

$Q = XW^Q \quad K = XW^K \quad V = XW^V$

$n \times d_q$ $\quad$ $d_k \times n$

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

$n \times d_v$

Need to compute n^2 pairs of scores (= dot product) $\quad O(n^2 d)$

RNNs only require $O(nd^2)$ running time:

$$\mathbf{h}_t = g(\mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t + \mathbf{b})$$

(assuming input dimension = hidden dimension = d)

The Transformer Architecture: Variants

$O = \text{Softmax}(QK^T)V$ — $O(n^2d)$ time & memory

In practice, the bottleneck is **memory bandwidth**:
the full n×n attention matrix must be read/written from slow GPU HBM.

⇒ **Approximate attention** (Sparse, Linear): fewer FLOPs by restricting which pairs are computed

Trade exact results for speed; work well on many tasks

⇒ **FlashAttention**: exact results, same FLOPs, but IO-aware — avoids materializing the full n×n matrix in HBM

Currently the standard in production LLMs (Llama, GPT-4, etc.)

Method	Complexity	Exact?
Standard attention	$O(n^2d)$	✓
Sparse (Longformer)	$O(nd)$	✗ approx.
Linear attention	$O(nd^2)$	✗ approx.
FlashAttention	$O(n^2d)$	✓ exact

Tay et al., "Efficient Transformers: A Survey," 2020

The Transformer Variants: Sparse Attention

Idea: don't let every token attend to every other token — restrict attention to a structured subset.

⇒ **Local window:** each token attends to its w nearest neighbors

Captures local syntactic patterns; O(nw) — linear if w ≪ n

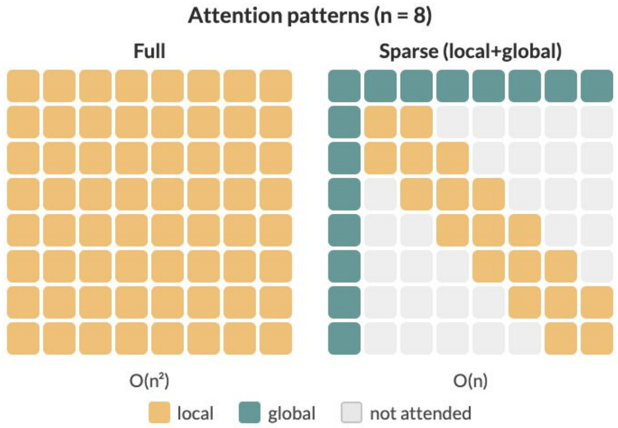
⇒ **Global tokens:** a few special tokens (e.g., [CLS]) attend to all positions

Enables long-range aggregation; e.g., [CLS] collects document-level info

⇒ **Random tokens** (BigBird only): each token also attends to a small random set

Provably equivalent to full attention under mild assumptions

Tradeoff: some token pairs never directly interact — information must "hop" via intermediate tokens across layers. Works best when *local context dominates* (e.g., NER, document classification).



Beltagy et al., Longformer, 2020 | Zaheer et al., BigBird, 2020

The Transformer Variants: Flash Attention

Key insight: the bottleneck is not FLOPs, but **memory bandwidth** — reading/writing the n×n matrix from slow GPU HBM.

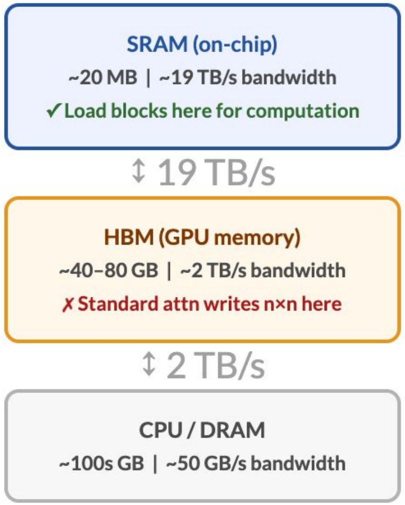
⇒ **Tiling:** compute attention in blocks that fit in fast SRAM, avoiding HBM reads/writes of the full score matrix

Decompose softmax normalization across blocks using the log-sum-exp trick

⇒ **Recomputation:** don't save the n×n matrix during the forward pass — recompute it during backprop from the smaller saved inputs

Uses more FLOPs, but far fewer HBM reads/writes → net speedup

Result: exact same output as standard attention, but 2–4× wall-clock speedup and $O(n)$ HBM memory usage (vs $O(n^2)$). Now standard in Llama, GPT-4, Gemini, etc.



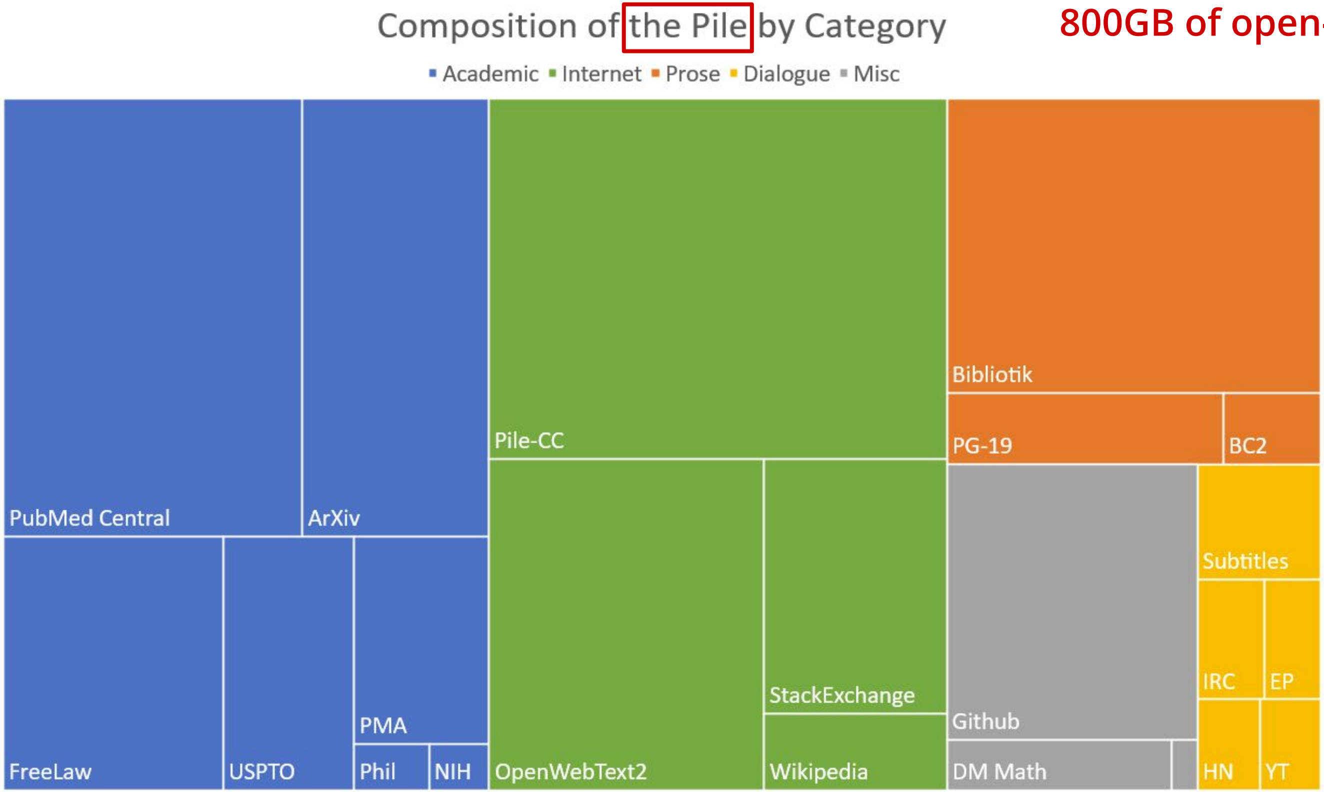
Flash loads blocks block-by-block into SRAM, never writing the full n×n to HBM.

Dao et al., FlashAttention, NeurIPS 2022

PLACEHOLDER

Training a LLM: Data

800GB of open-source data



[from Gao et al., The Pile: An 800GB Dataset of Diverse Text for Language Modeling]

Pretraining data- what one does with these data?

In the case of Llama 3.1:

"To train the best language model, the curation of a large, high-quality training dataset is paramount."

- PII and safety filtering
- Text extraction and cleaning from raw HTML pages
- De-duplication: URL, document, line-level, …
- Heuristic filtering:
    - Remove lines that consist of repeated content (e.g., n-gram coverage ratio)
    - Dirty word counting
    - KL divergence of token-distribution compared "high-quality corpus"
- **Model-based quality classifier: important and new trend!**
- Code, reasoning, and multilingual data

Training a LLM: Optimization

Now we can compute the LLM's guess for p(next token | previous tokens)
How can we improve the model?

Training a LLM: Optimization

Now we can compute the LLM's guess for p(next token | previous tokens)
How can we improve the model?

Check the probability of the correct token and move the model parameters a tiny bit to make the correct token more likely (stochastic gradient descent)

Training a LLM: Optimization

Now we can compute the LLM's guess for p(next token | previous tokens)
How can we improve the model?

Check the probability of the correct token and move the model parameters a tiny bit to make the correct token more likely (stochastic gradient descent)

How do we know which parameters need to change? ⇒ Compute the gradient of each parameter.

Training a LLM: Optimization

Now we can compute the LLM's guess for p(next token | previous tokens)
How can we improve the model?

Check the probability of the correct token and move the model parameters a tiny bit to make the correct token more likely (stochastic gradient descent)

How do we know which parameters need to change? ⇒ Compute the gradient of each parameter.

Always consider a batch of 10^3-10^6 of predictions and average gradients. ⇒ Parameters move to produce the best effect overall.

Training a LLM: Optimization

Now we can compute the LLM's guess for p(next token | previous tokens)

How can we improve the model?

Check the probability of the correct token and move the model parameters a tiny bit to make the correct token more likely (stochastic gradient descent)

How do we know which parameters need to change? ⇒ Compute the gradient of each parameter.

Always consider a batch of 10^3-10^6 of predictions and average gradients. ⇒ Parameters move to produce the best effect overall.

🔁 Repeat millions of times. ⟶ **Pre-Training!**

Contextual Word Embedding

Why Static Embeddings Fall Short

Static embeddings (word2vec, GloVe): one fixed vector per word type — regardless of context.

The word **"play"** has a single vector v_{play}, even though it means different things:

```
"a spectacular play on Alusik's grounder"  → sports
"a Broadway play for Garson"               → theater
"play an important role in cognition"       → verb
```

⇒ **Polysemy problem:** one vector can't capture multiple word senses

v(bank) must average over "river bank" and "financial bank"

⇒ **Solution:** compute embeddings **conditioned on the full input sentence**

$f: (w_1, \ldots, w_n) \rightarrow x_1, \ldots, x_n \in \mathbb{R}^d$

Static vs. Contextual

Static (word2vec)

`play` → `[0.22, −0.14, …]` *always same*

Contextual (ELMo/BERT)

"**play** on grounder" → [0.81, 0.32, …] (sports)

"Broadway **play**" → [−0.12, 0.67, …] (theater)

Peters et al., ELMo, NAACL 2018

ELMo

Key idea: train a *bidirectional* LSTM language model, then use the hidden states from *all layers* as the word representation.

⇒ **Two stacked BiLSTMs** trained on 1B Word Benchmark

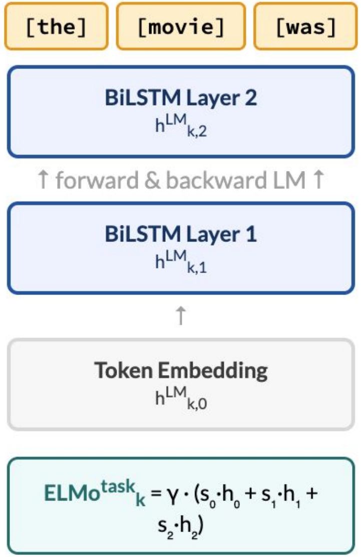
Forward LM: $p(w_t \mid w_1 \ldots w_{t-1})$ & Backward LM: $p(w_t \mid w_{t+1} \ldots w_n)$

⇒ **Contextual vector** = task-weighted average of all layer representations:

$$\text{ELMo}^{task}_k = \gamma^{task} \cdot \Sigma_j\, s^{task}_j \cdot h^{LM}_{k,j}$$

s^{task}_j: softmax-normalized weights learned per task | γ: overall scale

Why average all layers, not just the top? Lower layers encode *syntax* (POS, constituency), upper layers encode *semantics* (word sense, coreference). Different tasks benefit from different layers.

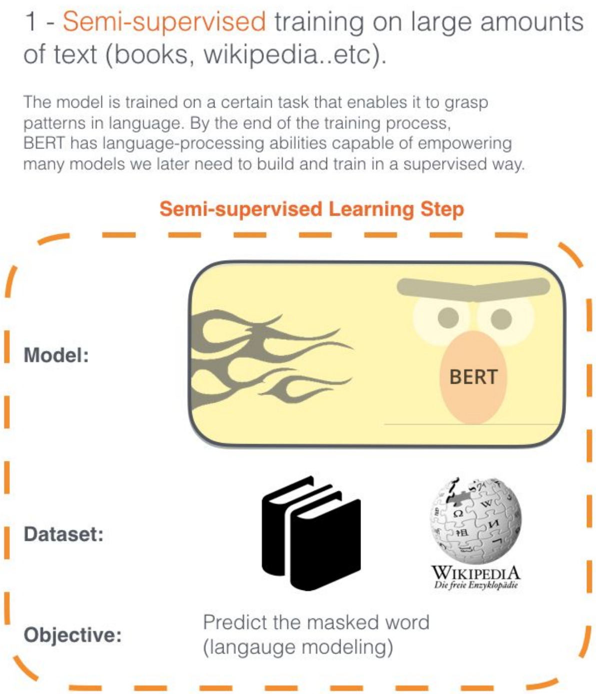


Peters et al., "Deep Contextualized Word Representations," NAACL 2018 | Released Feb 2018

BERT (Devlin et al., 2019)

- Bidirectional Encoder Representations from Transformers
- Key idea: Use a transformer to leverage **bidirectional context**
- Two objectives/loss optimization
  - Masked language modeling (MLM)
  - Next sentence prediction (NSP)
- Impact: one of the first works in NLP showing strong performance using a pre-trained transformer

BERT Pre-training



We first pre-train the model on a lot of data to learn basic language abilities.

[from Alammar, The Illustrated BERT, http://jalammar.github.io/illustrated-bert/]

BERT Pre-training

Key design choice: use a **bidirectional Transformer encoder** instead of a left-to-right decoder.

> **Why can't we just train a bidirectional LM?**
> A standard LM predicts the next token, so it can't look right. BERT's solution: predict **randomly masked** tokens using *both* left and right context.

⇒ **Masked LM (MLM):** mask 15% of tokens; predict them from bidirectional context

80% → [MASK], 10% → random token, 10% → unchanged

⇒ **Next Sentence Prediction (NSP):** predict if sentence B follows sentence A

Later ablations (Liu et al., 2019) show NSP often doesn't help — removed in RoBERTa

BERT Model Sizes

	BERT-Base	BERT-Large
Layers	12	24
Hidden size	768	1024
Attn heads	12	16
Parameters	110M	340M

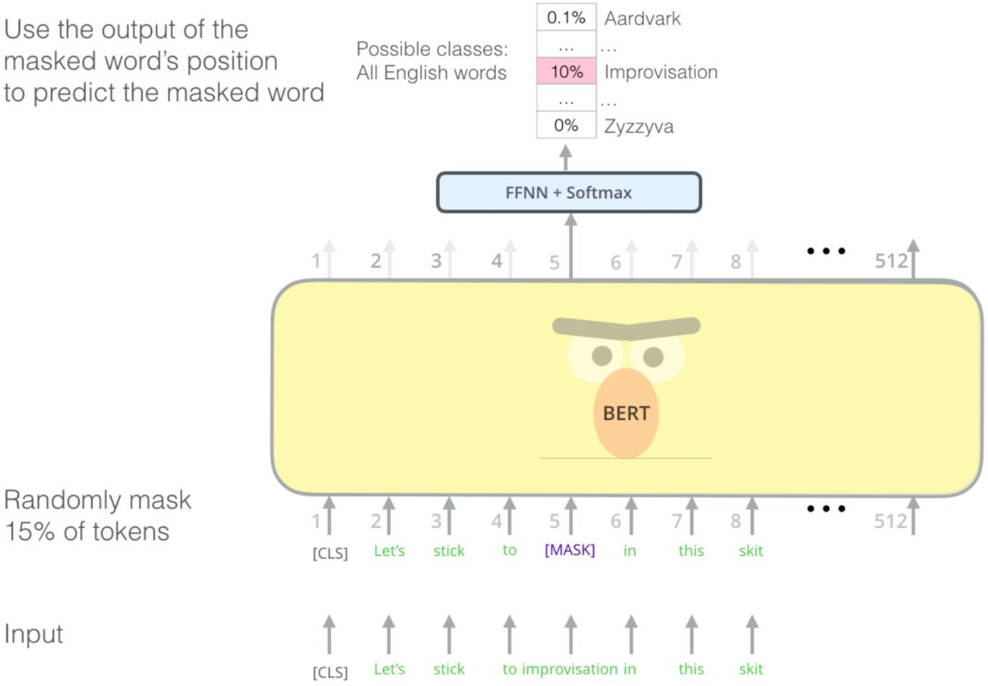
Training Setup

Corpus	Wikipedia (2.5B) + BooksCorpus (0.8B)
Max seq len	512 wordpiece tokens
Vocab size	30,000 wordpieces
Training	1M steps, batch 128K

Devlin et al., "BERT: Pre-training of Deep Bidirectional Transformers," NAACL 2019 | Released Oct 2018

Masked Language Modeling (MLM)

Learn to recover a masked word using the context



[from Alammar, The Illustrated BERT, http://jalammar.github.io/illustrated-bert/]

MLM- The 80-10-10 Strategy

Of the 15% selected tokens, the corruption applied varies:

%	Action	Why?
80%	Replace with `[MASK]`	Forces model to infer from context
10%	Replace with random token	Prevents over-reliance on [MASK]; model must check all positions
10%	Keep unchanged	Biases representation toward the true token

Train/inference mismatch: [MASK] tokens never appear at fine-tuning time. The 10/10 mix closes this gap — the model learns that *any* position may need correction, not just [MASK] positions.

Original sentence:

`[CLS]` `the` `man` `went` `to` `the` `store` `[SEP]`

After 80/10/10 corruption:

`[CLS]` `the` `man` `[MASK]` `to` `the` `running` `[SEP]`

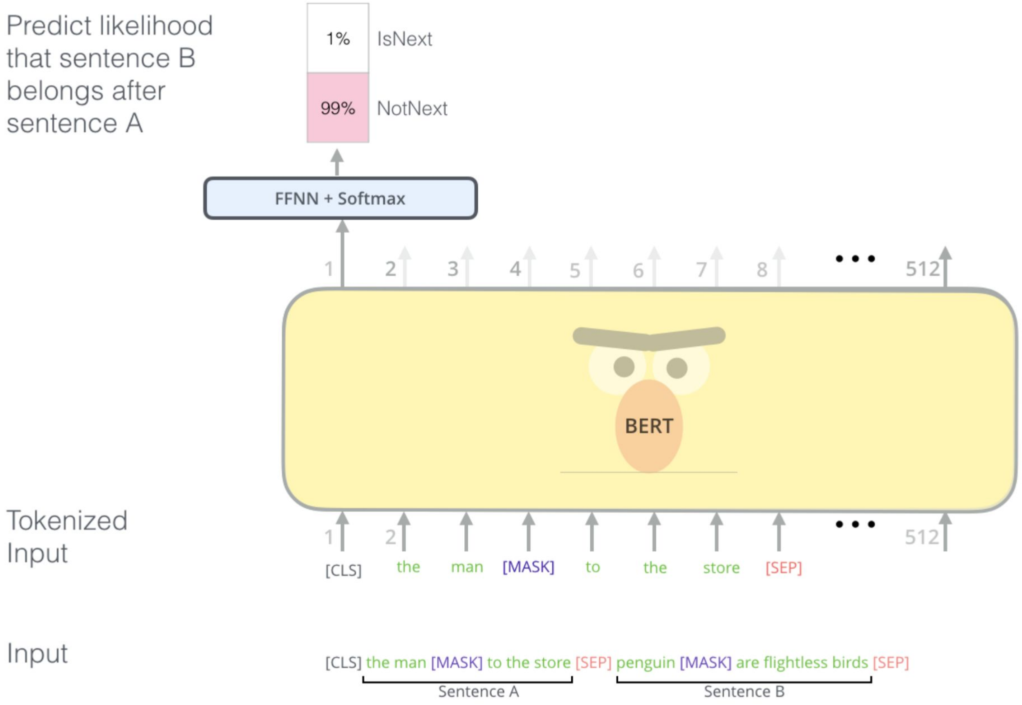
"went" → [MASK] (80%)

"store" → "running" (10%, random word)

Loss computed only at masked positions.

predict: `"went"` and `"store"`

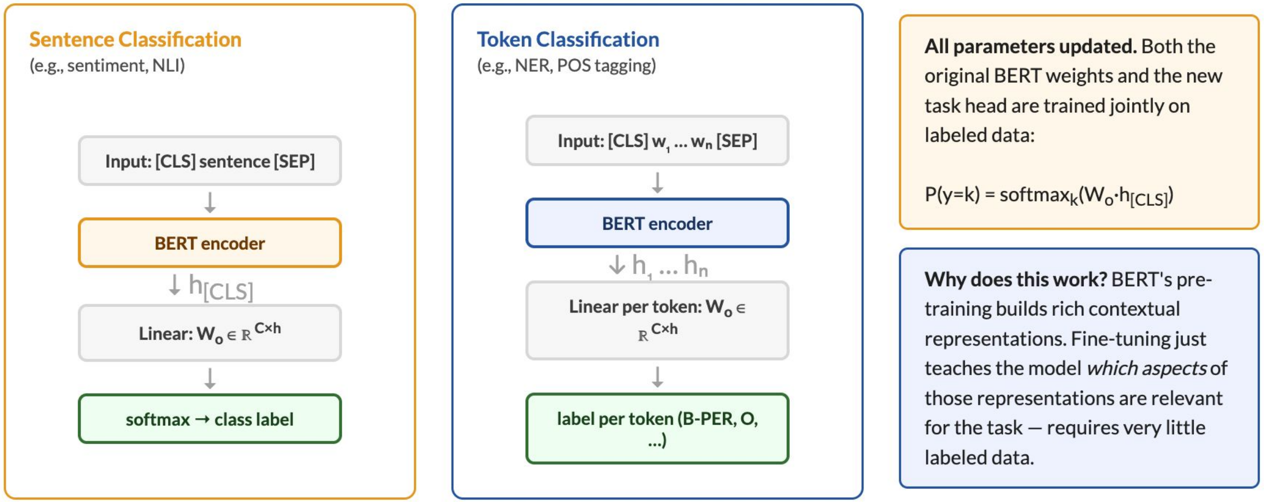
Next Sentence Prediction (NSP)



Later works showed that this doesn't always help (Liu et al., 2019)!

[from Alammar, The Illustrated BERT, http://jalammar.github.io/illustrated-bert/]

How to use BERT? Fine-tuning



"Pretrain once, fine-tune many times." — A single task head is added on top of the frozen-then-updated BERT encoder.

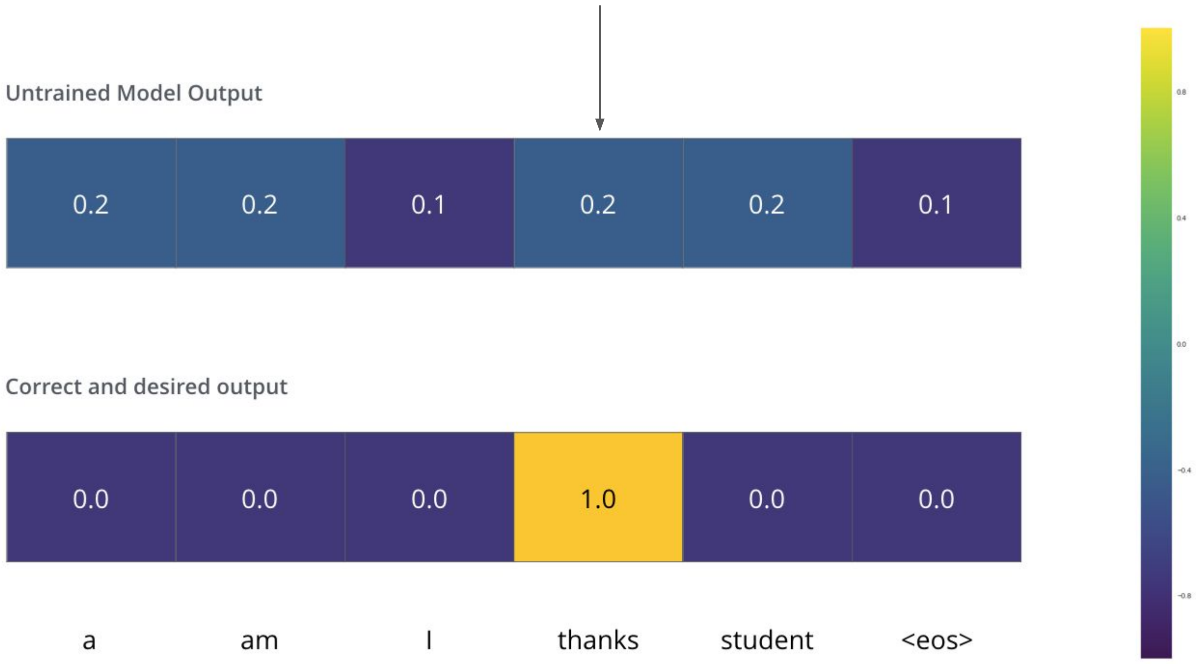
GPT (Radford et al., 2018)

- Generative Pre-training
- Similar to BERT, GPT also uses transformers, but the key difference is that we process text in an **unidirectional** fashion (left-to-right)
- One objectives/loss optimization
  - Language modeling or next token prediction
- Impact: the most impressive generative language model at the time

Next token prediction

Minimize the negative loglikelihood of the next token

We teach the model to predict tokens through the probability outputs



Next token prediction

Target Model Outputs

Output Vocabulary:	a	am	I	thanks	student	<eos>
position #1	0.0	0.0	1.0	0.0	0.0	0.0
position #2	0.0	1.0	0.0	0.0	0.0	0.0
position #3	1.0	0.0	0.0	0.0	0.0	0.0
position #4	0.0	0.0	0.0	0.0	1.0	0.0
position #5	0.0	0.0	0.0	0.0	0.0	1.0
	a	am	I	thanks	student	<eos>

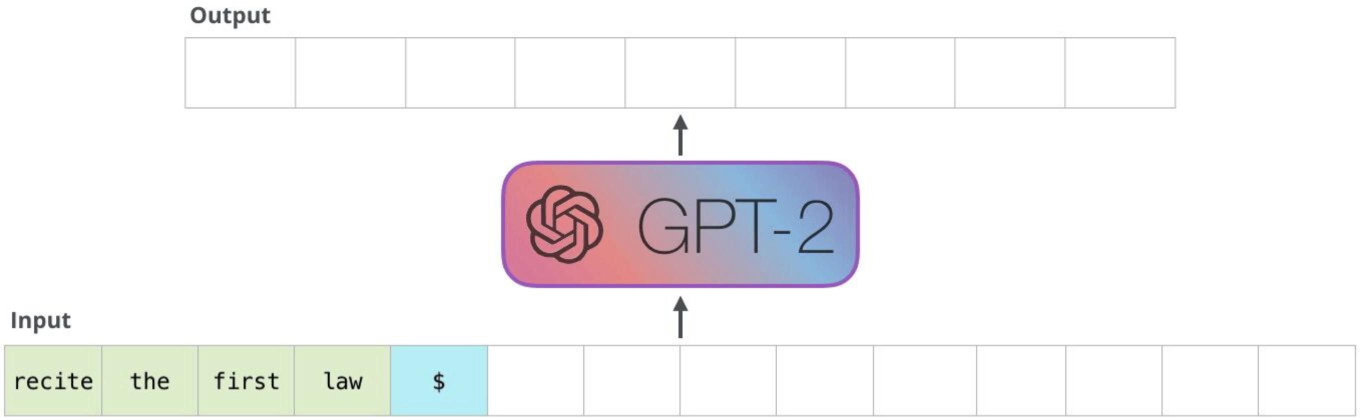
Trained Model Outputs

Output Vocabulary:	a	am	I	thanks	student	<eos>
position #1	0.01	0.02	0.93	0.01	0.03	0.01
position #2	0.01	0.8	0.1	0.05	0.01	0.03
position #3	0.99	0.001	0.001	0.001	0.002	0.001
position #4	0.001	0.002	0.001	0.02	0.94	0.01
position #5	0.01	0.01	0.001	0.001	0.001	0.98
	a	am	I	thanks	student	<eos>

Text Generation

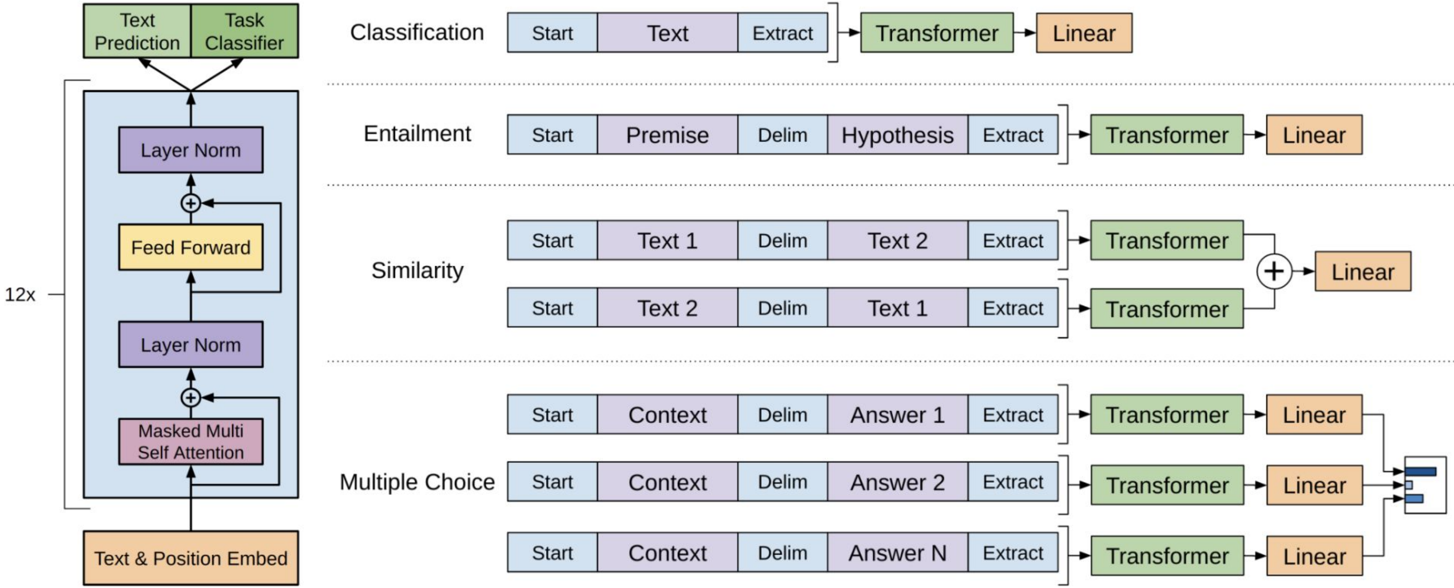
1. Sample a token from ~ p(next token | previous tokens)
2. Append the token to the input
3. Run the new input through the transformer

… and so on…



[from Alammar, The Illustrated GPT-2, https://jalammar.github.io/illustrated-gpt2/]

How to use GPT-2? Fine-tuning



To summarize: how is the pretrained model used downstream?

Feature-Based (ELMo)

Pre-trained model weights are **frozen**. Its hidden states are used as *input features* to a separately trained task model.

Pre-trained BiLSTM
(frozen ❄️)

↓ embeddings

Task model
(trained from scratch)

✓ Can use any task architecture
✗ Pre-trained representations not adapted to the task

Fine-Tuning (GPT, BERT)

Pre-trained model weights are used as initialization and **updated** together with a small task-specific head during training.

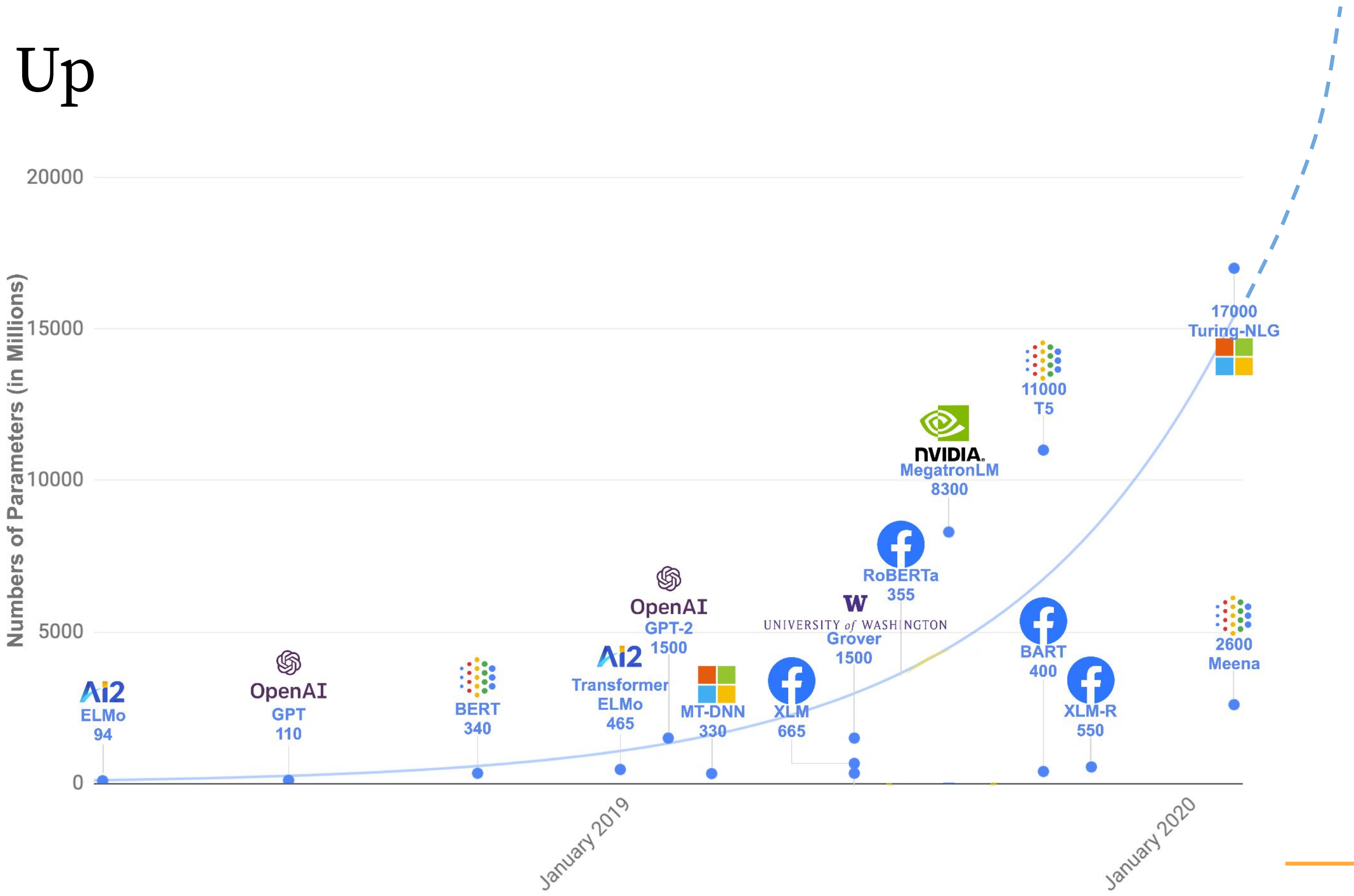
Pre-trained Transformer
(fine-tuned 🔥)

↓ [CLS] or token reps

Small task head
(C×h matrix)

✓ Representations adapt to the task
✓ "Pretrain once, fine-tune many times"

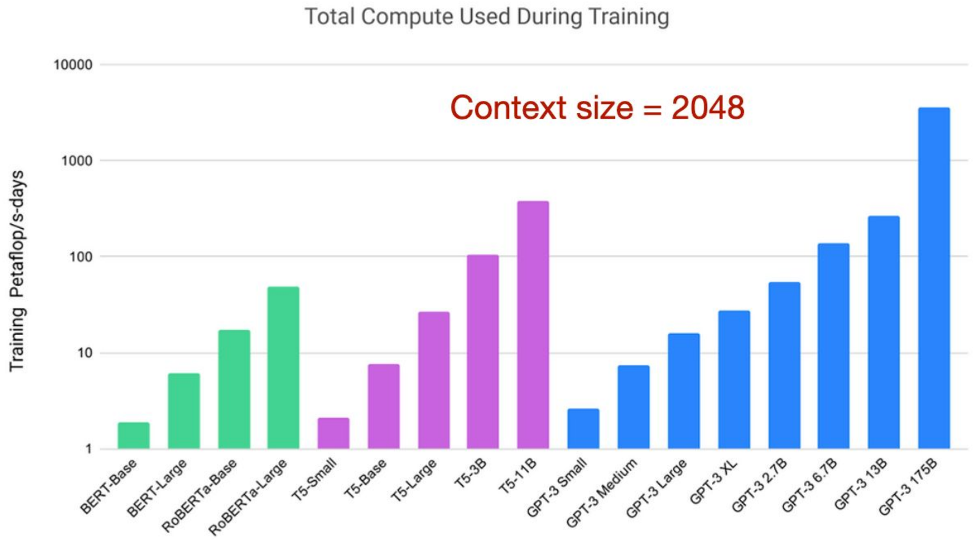
Scaling Up



[adapted from https://huggingface.co/course/chapter1/4]

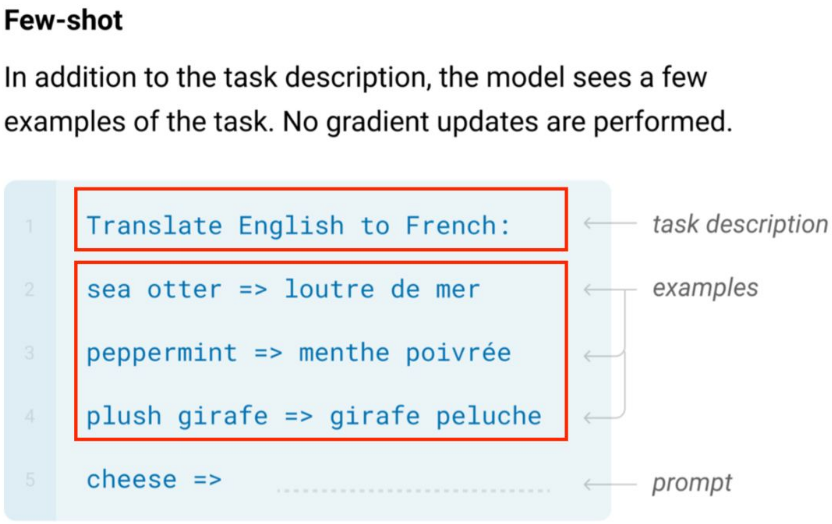
GPT-3 (preview for next class)

What's new? More parameters and more data.



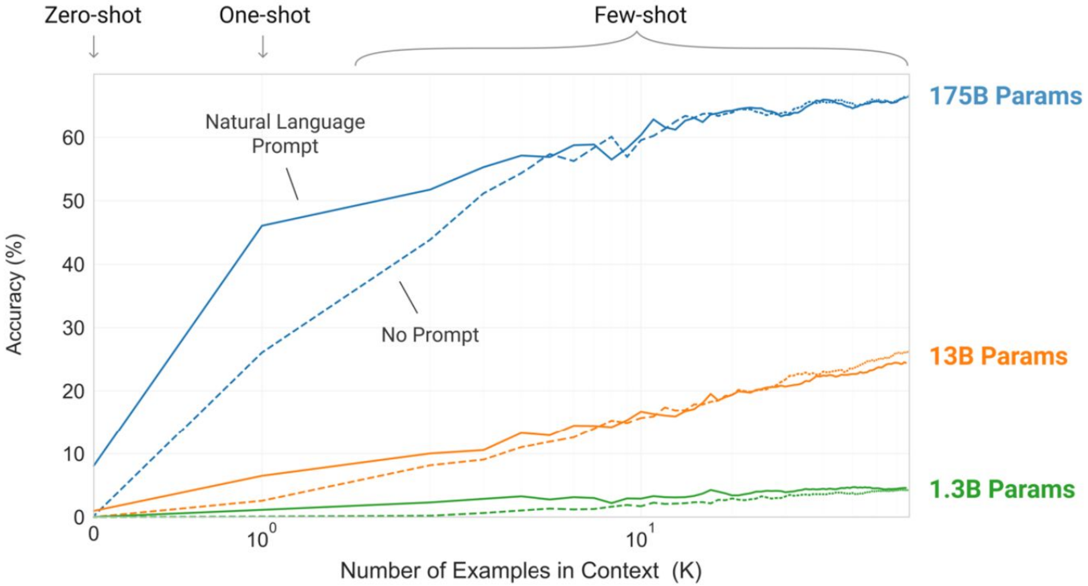
How to use GPT3? In-context learning

Fine-tuning is too expensive to such large models.



How to use GPT3? In-context learning

More examples = better performance



Questions?