How language models work: From n-grams to instruction-tuned LLMs

Dong Nguyen

2026

Utrecht University

What is a Large Language Model?

What is **a language model**?

"A language model is a model of the human brain's ability to produce natural language." *Wikipedia*

"A language model is a machine learning model that predicts upcoming words. More formally, a language model assigns a probability to each possible next word," *SLP3*

...

What is a Large Language Model?

What makes an LLM "**large**"?

Model	# **parameters**
BERT	*bert-base-cased **110M**; bert-large-uncased: **340M***
Llama	Llama 4 Scout **17B (active) 109B (total)**
GPT	GPT-3 (**175B**); (GPT-4/5 undisclosed)
Gemma	gemma-4-**26B-A4B**-it; gemma-4-**31B**-it

What is a Large Language Model?

What makes an LLM "**large**"?

But… it's not only about the number of parameters!

- Training on *lots* of data (also: increasingly multimodal)
- *Long* context windows (e.g., hundreds of thousands of tokens)
- *Strong* capabilities (e.g., reasoning)
- etc…

Paradigm shift

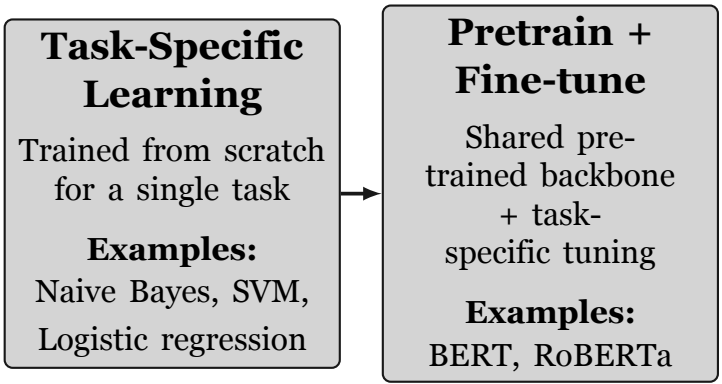
Task-Specific Learning

Trained from scratch for a single task

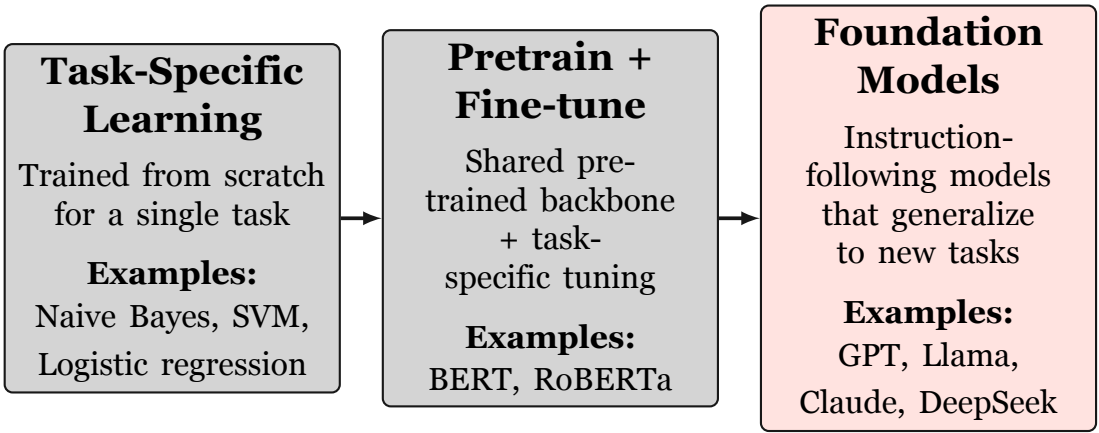
Examples:

Naive Bayes, SVM, Logistic regression

Paradigm shift



Paradigm shift

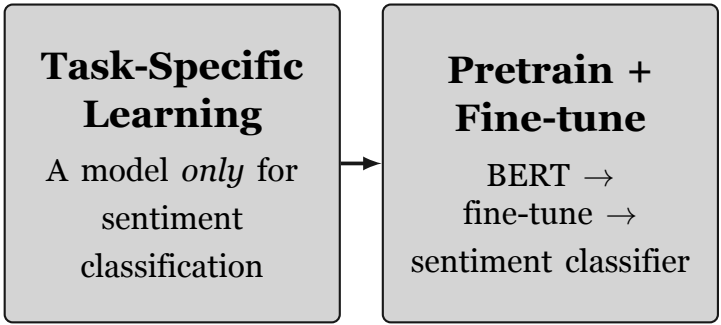


Paradigm shift

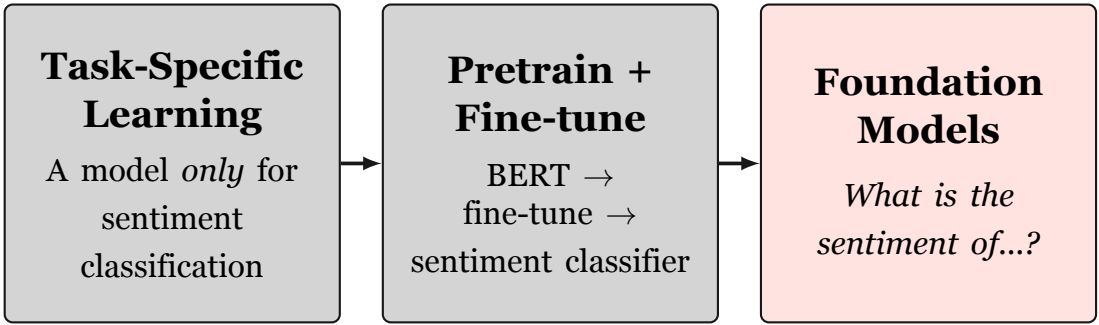
Task-Specific Learning

A model *only* for sentiment classification

Paradigm shift



Paradigm shift



Text generation

Key idea: Many tasks can be turned into the task of predicting words!

Text generation

Key idea: Many tasks can be turned into the task of predicting words!

Sentiment analysis:

P(*positive* | *The sentiment of "This is a great movie!" is: "*)
P(*negative* | ...)

Text generation

Key idea: Many tasks can be turned into the task of predicting words!

Math:

P(1 | *5* + *5* =)
P(10 | …)

Agenda

Goal: Introduce you to the core components of Large Language Models

Roadmap:

- Historical context
  - n-gram language models, RNNs
- Core components of Transformer models
  - tokenization, attention, decoding, etc.
- Transformer models
  - encoder, decoder, encoder-decoders
- LLMs
  - pre-training, instruction tuning

Historical Context

What is a Language Model (LM)?

A **language model** assigns probabilities to sequences of words.

It can estimate the probability of the **next word** given previous words:

$$p(\texttt{great} \mid \texttt{this is})$$

It can estimate the probability of an **entire text** (e.g., sentence, paragraph):

$$p(\texttt{this is great})$$

What is a Language Model (LM)?

A **language model** assigns probabilities to sequences of words.

It can estimate the probability of the **next word** given previous words:

$$p(\texttt{great} \mid \texttt{this is})$$

It can estimate the probability of an **entire text** (e.g., sentence, paragraph):

$$p(\texttt{this is great})$$

For example:

$$p(\texttt{Athens is the capital of Greece}) > p(\texttt{capital is the of Greece Athens})$$

What is a Language Model (LM)?

A **language model** assigns probabilities to sequences of words.

It can estimate the probability of the **next word** given previous words:

$$p(\texttt{great} \mid \texttt{this is})$$

It can estimate the probability of an **entire text** (e.g., sentence, paragraph):

$$p(\texttt{this is great})$$

For example:

$$p(\texttt{Athens is the capital of Greece}) >$$
$$p(\texttt{Amsterdam is the capital of Greece})$$

What is a Language Model (LM)?

A **language model** assigns probabilities to sequences of words.

It can estimate the probability of the **next word** given previous words:

$$p(\texttt{great} \mid \texttt{this is})$$

It can estimate the probability of an **entire text** (e.g., sentence, paragraph):

$$p(\texttt{this is great})$$

For example:

$$p(\texttt{Athens is the capital of Greece}) > p(\texttt{Atens is the capital of Greece})$$

What is a Language Model (LM)?

A **language model** assigns probabilities to sequences of words.

It can estimate the probability of the **next word** given previous words:

$$p(\texttt{great} \mid \texttt{this is})$$

It can estimate the probability of an **entire text** (e.g., sentence, paragraph):

$$p(\texttt{this is great})$$

Key idea: Modeling language = modeling word sequences probabilistically.

The chain rule

Computing the probability of a whole sequence:

$$P(w_1, \ldots, w_N) = \prod_{i=1}^{N} P(w_i \mid w_1, \ldots, w_{i-1})$$

The chain rule

Computing the probability of a whole sequence:

$$P(w_1, \ldots, w_N) = \prod_{i=1}^{N} P(w_i \mid w_1, \ldots, w_{i-1})$$

For example:

$$P(\texttt{this is great}) = P(\texttt{this})\, P(\texttt{is} \mid \texttt{this})\, P(\texttt{great} \mid \texttt{this is})$$

The chain rule

Computing the probability of a whole sequence:

$$P(w_1, \ldots, w_N) = \prod_{i=1}^{N} P(w_i \mid w_1, \ldots, w_{i-1})$$

For example:

$$P(\texttt{I am teaching a lecture at AthensNLP on LLMs})$$
$$= P(\texttt{I})\, P(\texttt{am} \mid \texttt{I})\, P(\texttt{teaching} \mid \texttt{I am}) \cdots$$
$$\cdots P\big(\texttt{LLMs} \mid \texttt{I am teaching a lecture at AthensNLP on}\big)$$

Estimating probabilities by counting

We can estimate next-word probabilities by *counting* in a corpus:

$$P(\texttt{great} \mid \texttt{this is}) = \frac{\text{count}(\texttt{this is great})}{\text{count}(\texttt{this is})}$$

Estimating probabilities by counting

We can estimate next-word probabilities by *counting* in a corpus:

$$P(\texttt{great} \mid \texttt{this is}) = \frac{\text{count}(\texttt{this is great})}{\text{count}(\texttt{this is})}$$

Suppose our whole corpus is three sentences:

```
this is great
this is a lecture
this is great fun
```

What is $P(\texttt{great} \mid \texttt{this is})$?

Estimating probabilities by counting

We can estimate next-word probabilities by *counting* in a corpus:

$$P(\texttt{great} \mid \texttt{this is}) = \frac{\text{count}(\texttt{this is great})}{\text{count}(\texttt{this is})}$$

Suppose our whole corpus is three sentences:

```
this is great
this is a lecture
this is great fun
```

What is $P(\texttt{great} \mid \texttt{this is})$?

$$\text{count}(\texttt{this is}) = 3,$$
$$\text{count}(\texttt{this is great}) = 2 \quad \Rightarrow$$
$$P = \frac{2}{3}$$

Estimating probabilities by counting

We can estimate next-word probabilities by *counting* in a corpus:

$$P(\texttt{LLMs} \mid \texttt{I am teaching a lecture at AthensNLP on}) = \frac{\text{count}(\texttt{I am teaching a lecture at AthensNLP on LLMs})}{\text{count}(\texttt{I am teaching a lecture at AthensNLP on})}$$

Estimating probabilities by counting

We can estimate next-word probabilities by *counting* in a corpus:

$$P(\texttt{LLMs} \mid \texttt{I am teaching a lecture at AthensNLP on}) = \frac{\text{count}(\texttt{I am teaching a lecture at AthensNLP on LLMs})}{\text{count}(\texttt{I am teaching a lecture at AthensNLP on})}$$

How often do you think anyone has ever written this exact sentence?

Estimating probabilities by counting

We can estimate next-word probabilities by *counting* in a corpus:

$$P(\texttt{LLMs} \mid \texttt{I am teaching a lecture at AthensNLP on}) = \frac{\text{count}(\texttt{I am teaching a lecture at AthensNLP on LLMs})}{\text{count}(\texttt{I am teaching a lecture at AthensNLP on})}$$

How often do you think anyone has ever written this exact sentence?

Probably **never**! *Most well-formed sentences have never been written before.* 🤔

n-gram language modeling

We can approximate this by only looking at the previous $n-1$ words (*Markov* assumption):

$$P(w_i \mid w_1, \ldots, w_{i-1}) \approx P(w_i \mid w_{i-n+1}, \ldots, w_{i-1})$$

For a bigram model ($n = 2$) the context is a single word.

$$P(\texttt{LLMs} \mid \texttt{I am teaching a lecture at AthensNLP on}) \approx$$

$$P(\texttt{LLMs} \mid \texttt{on}) = \frac{\text{count}(\texttt{on LLMs})}{\text{count}(\texttt{on})}$$

Larger n means more context, but fewer observations per context.

Bigram model: the whole sequence

Estimating the probability of the full sequence using a bigram model ($n = 2$):

$$P(\texttt{I am teaching a lecture at AthensNLP on LLMs}) \approx P(\texttt{I})\, P(\texttt{am} \mid \texttt{I})\, P(\texttt{teaching} \mid \texttt{am}) \cdots P(\texttt{LLMs} \mid \texttt{on})$$

Bigram model: the whole sequence

Estimating the probability of the full sequence using a bigram model ($n = 2$):

$$P(\texttt{I am teaching a lecture at AthensNLP on LLMs})$$
$$\approx P(\texttt{I})\, P(\texttt{am} \mid \texttt{I})\, P(\texttt{teaching} \mid \texttt{am}) \cdots P(\texttt{LLMs} \mid \texttt{on})$$

What happens if a bigram (e.g., "*on LLMs*") is never seen in the training data?

Smoothing

A single unseen bigram results in the entire product being zero 😕(i.e., the sentence is impossible).

With smoothing we redistribute the probability mass.

Laplace smoothing: pretend we saw every word one extra time in every context. For example when we have bigrams:

$$P(w_i \mid w_{i-1}) = \frac{\text{count}(w_{i-1}w_i) + 1}{\text{count}(w_{i-1}) + |V|}$$

Limitations of n-gram language modeling

- Fixed, short context window ($n - 1$). Longer dependencies can't be modeled

```
Leon, who moved here last year, just stopped by. He...
```

- Need to store all the frequency counts in the training data; model size grows as the training data grows.
- Only looks at *surface* forms. For example *teaching* vs. *presenting*.
- Smoothing is hand-engineered and has limitations.

Feed Forward Neural LM

Key idea: use a neural network to predict word probabilities! (remember: word embeddings)

Training data:

Seen: `I flew to Athens last summer.`

Never seen: `flew to Thessaloniki`

Test data:

`I flew to Thessaloniki last ...`

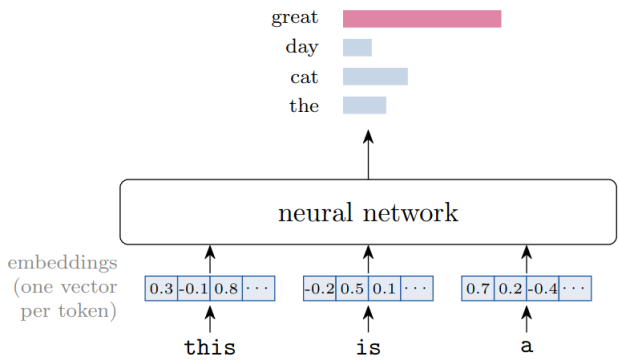
An n-gram LM cannot predict `summer` 😕

A neural LM can use the similarity between `Athens` and `Thessaloniki` to generalize 😄

Feed Forward Neural LM

A feedforward network that outputs the probability distribution over possible next words given the previous $n-1$ words (here, n=4).

Words are represented by learned embeddings, which are then concatenated.

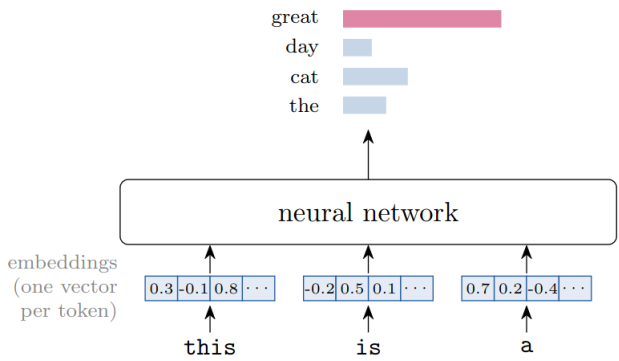


A Neural Probabilistic Language Model, Bengio et al., 2003 [link]

Feed Forward Neural LM

A feedforward network that outputs the probability distribution over possible next words given the previous $n-1$ words (here, n=4).

Words are represented by learned embeddings, which are then concatenated.



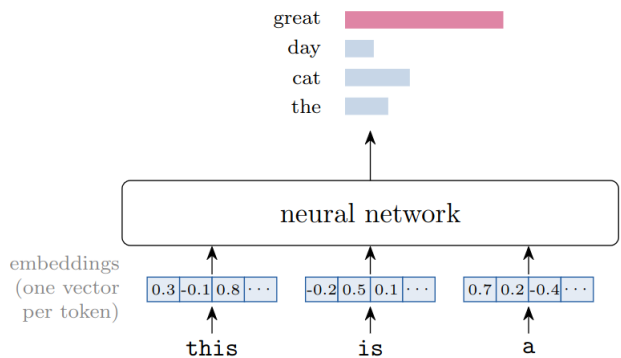
A Neural Probabilistic Language Model, Bengio et al., 2003 [link]

Feed Forward Neural LM

Can now generalize over context of similar words! 😄

Less interpretable 😕

Still a problem: fixed, short context window 😕



A Neural Probabilistic Language Model, Bengio et al., 2003 [link]

Recurrent Neural Networks (RNNs)

Process the sequence **one token at a time**, maintaining a hidden state $\mathbf{h}_t$ that summarizes everything seen so far:

$$\begin{aligned}\mathbf{e}_t &= \mathbf{E}\mathbf{x}_t \\ \mathbf{h}_t &= g\big(\mathbf{U}\mathbf{h}_{t-1} + \mathbf{W}\mathbf{e}_t\big) \\ \hat{\mathbf{y}}_t &= \text{softmax}\left(\mathbf{V}\mathbf{h}_t\right)\end{aligned}$$

- $\mathbf{E}$: embedding matrix; $\mathbf{h}_t$: the "memory" after t tokens
- g: a nonlinearity, e.g., tanh
- $\mathbf{W}, \mathbf{U}, \mathbf{V}$ are *the same at every step*, so the model can handle sequences of any length.

Feed Forward Neural LM vs. RNN

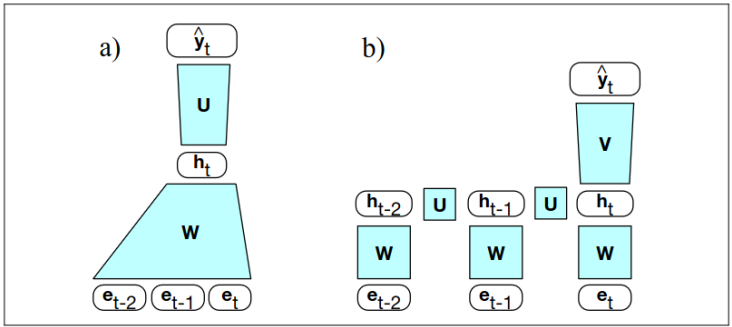


Figure 14.5 Simplified sketch of two LM architectures moving through a text, showing a schematic context of three tokens: (a) a feedforward neural language model which has a fixed context input to the weight matrix **W**, (b) an RNN language model, in which the hidden state $\mathbf{h}_{t-1}$ summarizes the prior context.

Figure: Figure from Speech and Language Processing, Jurafsky and Martin, Chapter 14

Different RNN-based architectures

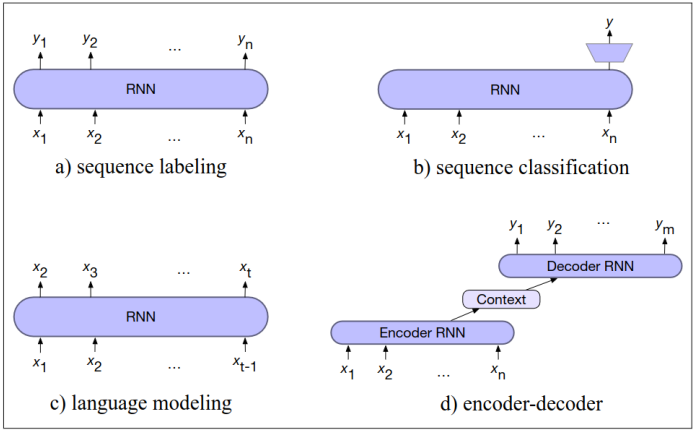
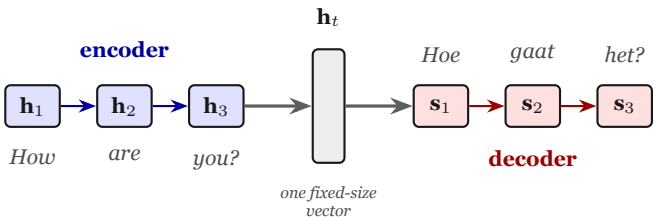


Figure: Figure 14.15 from Speech and Language Processing, Jurafsky and Martin, Chapter 14

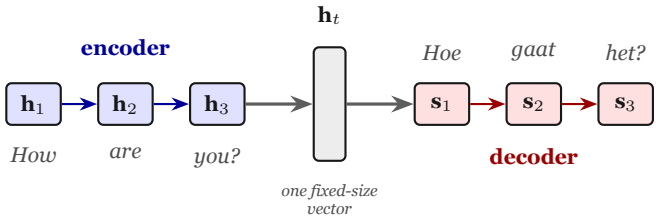
Example: Machine translation with RNNs

- The **encoder** RNN reads the English sentence
- Its final hidden state ($\mathbf{h}_t$) is passed to the **decoder** RNN
- The decoder generates the Dutch translation (token by token)



Example: Machine translation with RNNs

- The **encoder** RNN reads the English sentence
- Its final hidden state ($\mathbf{h}_t$) is passed to the **decoder** RNN
- The decoder generates the Dutch translation (token by token)



All information about the input has to fit in this final hidden state... 🤔

One vector bottleneck

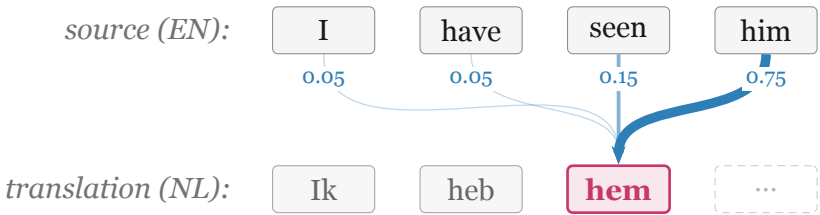
One vector has to remember everything 😕

- $\mathbf{h}_t$ has the same size regardless of whether it summarizes 5 or 1000 tokens; early information is often not retained well
- The training signal is propagated back step by step and gets weaker along the way (*vanishing gradients*)

Note: In practice people used LSTMs, a variant with gates that is better in managing the context (what to keep, what to remove).

Attention!

Instead of compressing everything into one vector, let the decoder look back at *all* encoder states and put more weight on the ones that matter.



note: attention was invented *for* RNNs, for the task of machine translation

Neural Machine Translation by Jointly Learning to Align and Translate, Bahdanau et al., 2015 [link]

Quick check

1. Why can an n-gram model not handle "*The researchers who published this dataset **have** moved to industry*"?

Quick check

1. Why can an n-gram model not handle "*The researchers who published this dataset **have** moved to industry*"?
   the context window is too short

Quick check

1. Why can an n-gram model not handle "*The researchers who published this dataset **have** moved to industry*"?
   the context window is too short
2. What do neural LMs have that n-gram models don't?

Quick check

1. Why can an n-gram model not handle "*The researchers who published this dataset **have** moved to industry*"?
   the context window is too short
2. What do neural LMs have that n-gram models don't?
   a notion of *similarity* through embeddings

Quick check

1. Why can an n-gram model not handle "*The researchers who published this dataset **have** moved to industry*"?
   the context window is too short
2. What do neural LMs have that n-gram models don't?
   a notion of *similarity* through embeddings
3. Why was attention originally introduced?

Quick check

1. Why can an n-gram model not handle "*The researchers who published this dataset **have** moved to industry*"?
   the context window is too short
2. What do neural LMs have that n-gram models don't?
   a notion of *similarity* through embeddings
3. Why was attention originally introduced?
   to remove the one-vector bottleneck in translation

Transformer Basics

Agenda

Goal: Introduce you to the core components of Large Language Models

Roadmap:

- Historical context
- Core components of Transformer models
- Transformer models
- Training LLMs: pretraining and instruction tuning

Transformers

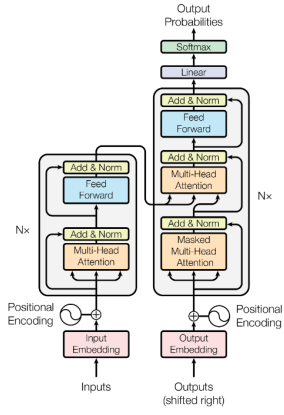


Figure 1: The Transformer - model architecture.

Figure: Figure from Attention Is All You Need, Vaswani et al., 2017 [link]

Basics

	Components	What it does
input	tokenization, embeddings	from text to vectors
Transformer block ($\times N$)	attention, positional embeddings, layer normalization, feedforward layer, residual connections	building contextual representations
Output	LM head, decoding	making word predictions

Tokenization

Tokenizer: splits a text into *tokens*

Where is Utrecht?

Tokenization

Tokenizer: splits a text into *tokens*

Where is Utrecht?

```
[_Where, _is, _U, tre, cht, ?]
```

The tokenizer is fitted on a dataset before the actual model training starts!

Byte Pair Encoding (BPE)

Start with characters:
Vocab: {l,m,s,g,r,e,c,a,t}

l l m s

g r e e c e

g r e a t

Byte Pair Encoding (BPE)

Start with characters:
Vocab: {l,m,s,g,r,e,c,a,t}

l l m s

g r e e c e

g r e a t

Count adjacent pairs and merge the most frequent one
The pair (g, r) occurs in both greece and great (tie with (r, e)).
g + r ⟶ gr

Byte Pair Encoding (BPE)

Start with characters:
Vocab: {l,m,s,g,r,e,c,a,t}

l l m s

g r e e c e

g r e a t

Count adjacent pairs and merge the most frequent one
The pair (g, r) occurs in both greece and great (tie with (r, e)).
g + r ⟶ gr

After merging:

l l m s

gr e e c e

gr e a t

Updated vocabulary:

{l, m, s, g, r, e, c, a, t, gr}

Byte Pair Encoding (BPE)

Start with characters:
Vocab: {l,m,s,g,r,e,c,a,t}

l l m s

g r e e c e

g r e a t

Count adjacent pairs and merge the most frequent one
The pair (g, r) occurs in both greece and great (tie with (r, e)).
g + r ⟶ gr

After merging:

l l m s

gr e e c e

gr e a t

Updated vocabulary:

{l, m, s, g, r, e, c, a, t, gr}

Repeat until the desired vocabulary size is reached.

Byte Pair Encoding (BPE)

Start with characters:
Vocab: {l,m,s,g,r,e,c,a,t}

l l m s

g r e e c e

g r e a t

Count adjacent pairs and merge the most frequent one

After merging:

l l m s

gr e e c e

gr e a t

Updated vocabulary:

{l, m, s, g, r, e, c, a, t, gr}

Question: What would be the next merge?

Byte Pair Encoding (BPE)

Start with characters:
Vocab: {l,m,s,g,r,e,c,a,t}

l l m s

g r e e c e

g r e a t

Count adjacent pairs and merge the most frequent one

After merging:

l l m s

gr e e c e

gr e a t

Updated vocabulary:

{l, m, s, g, r, e, c, a, t, gr}

Question: What would be the next merge? gr + e ⟶ gre

Byte Pair Encoding (BPE)

Start with characters:
Vocab: {l,m,s,g,r,e,c,a,t}

l l m s

g r e e c e

g r e a t

Count adjacent pairs and merge the most frequent one

After merging:

l l m s

gr e e c e

gr e a t

Updated vocabulary:

{l, m, s, g, r, e, c, a, t, gr}

Question: What would be the next merge? gr + e ⟶ gre
Question: What happens to a word the tokenizer has never seen before?

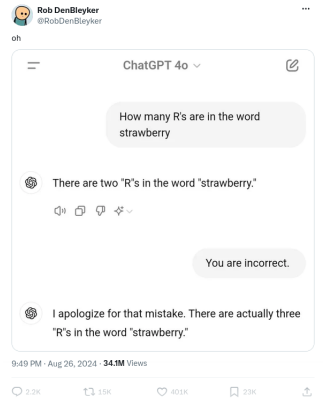
Tokenization: fairness

The number of tokens required to encode the "same" text is not equal across languages due to how the tokenizer was trained! E.g., the tokenizer by ChatGPT/GPT-4 uses $\approx$ 3 times more tokens to encode the same text in Arabic compared to English (Petrov et al. 2023).
Consequently:

- More computation required
- Higher API costs
- Less content fits in the context window

Do all languages cost the same? Tokenization in the era of commercial language models, Ahia et al., EMNLP 2023 [url]
Language model tokenizers introduce unfairness between languages, Petrov et al., NeurIPS 2023 [url]

How many r's are in strawberry?

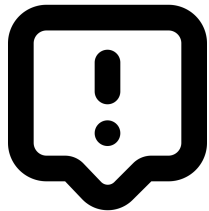


See also: https://www.computer.org/csdl/magazine/co/2025/03/10896907/24uGEvXcb1C (Can ChatGPT Learn to Count Letters?) and https://techcrunch.com/2024/08/27/why-ai-cant-spell-strawberry/

Note on tokens

Note: Language models process text as **tokens**.

For simplicity, in these slides tokens correspond to words, but in practice they are often **subwords**, depending on the tokenizer.



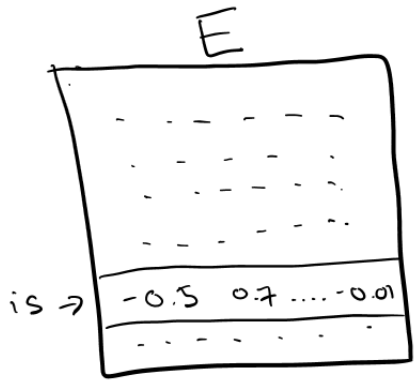
The embedding layer

Key idea is similar to word embeddings: map a token to a dense vector!

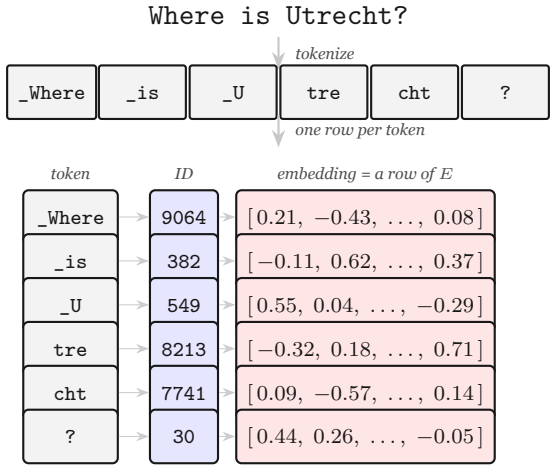
Every token in the vocabulary is represented by one row of an **embedding matrix** $E \in \mathbb{R}^{|V| \times d}$.

But here:

- E is trained with the model
- The rows are tokens (e.g., subwords)



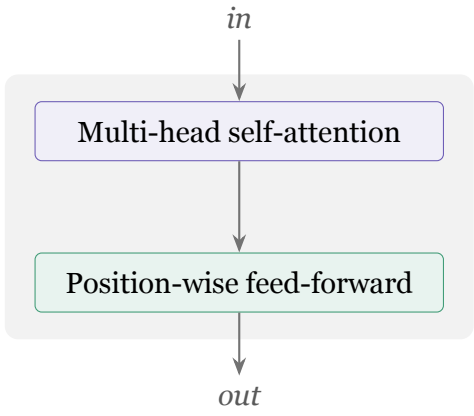
Summary: from text to vectors



Basics

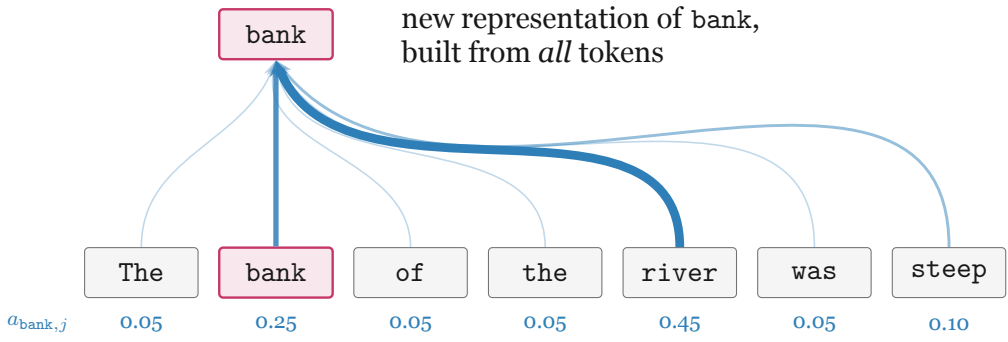
	Components	What it does
input	tokenization, embeddings	from text to vectors
Transformer block ($\times N$)	attention, positional embeddings, layer normalization, feedforward layer, residual connections	building contextual representations
Output	LM head, decoding	making word predictions

Transformer Block

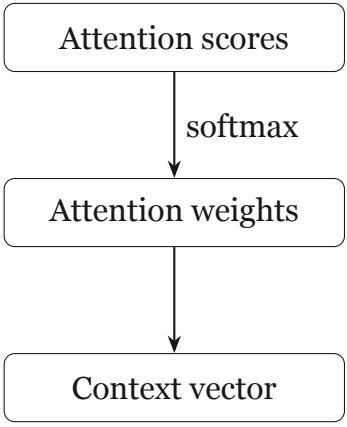


Goal:
Learn good contextual representations

Self-attention: Example



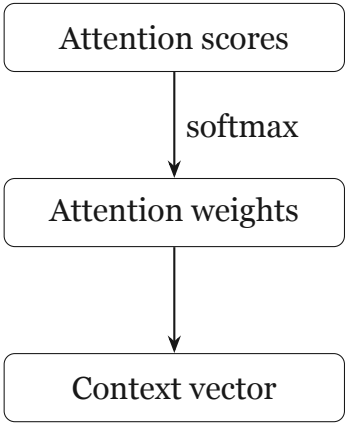
Self-attention



Allows the model to pay more attention to certain words/tokens in the sequence

Attention scores: How *relevant* each token is to the token whose representation we're computing

Self-attention



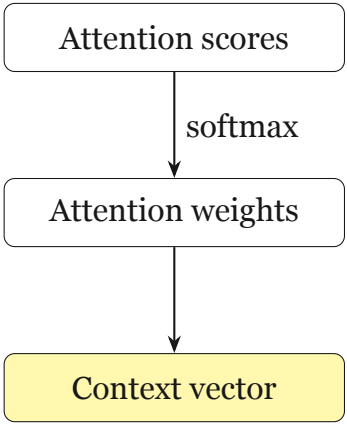
Allows the model to pay more attention to certain words/tokens in the sequence

Attention scores: How *relevant* each token is to the token whose representation we're computing

Vaswani et al., 2017:

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

Self-attention



Allows the model to pay more attention to certain words/tokens in the sequence

Attention scores: How *relevant* each token is to the token whose representation we're computing

How do we use the attention weights?

Self-attention: Computing context vectors

Key idea: The context vector $\mathbf{c}_i$ for token i is computed as a **weighted average** (with weights a_{ij}) of all token embeddings $\mathbf{x}_j$:

$$\mathbf{c}_i = \sum_j a_{ij}\, \mathbf{x}_j$$

In contrast to just averaging embeddings:

- The weighting function is *learned*
- Weights depend on both $\mathbf{x}_i$ and $\mathbf{x}_j$.

Self-attention: Computing context vectors

Key idea: The context vector $\mathbf{c}_i$ for token i is computed as a **weighted average** (with weights a_{ij}) of all token embeddings $\mathbf{x}_j$:

$$\mathbf{c}_i = \sum_j a_{ij}\, \mathbf{x}_j$$

Note: *the following is a simplified example. The token embeddings are averaged directly. In a real Transformer they are computed from Queries and Keys, and we average learned Value vectors.*

Self-attention: Computing context vectors

Context vector for token 2.

Token j	**Embedding**	**Attention weight a_{2j}**
The	`[0.1, 0.5, 0.2]`	0.3
dog	`[0.3, 0.1, 0.3]`	0.4
is	`[0.7, 0.2, 0.1]`	0.2
sleeping	`[0.2, 0.5, 0.6]`	0.1

a_{2j}: Attention from token 2 ("*dog*") to token j, i.e., how much "dog" attends to each token

Self-attention: Computing context vectors

Context vector for token 2.

Token j	**Embedding**	**Attention weight a_{2j}**
The	`[0.1, 0.5, 0.2]`	0.3
dog	`[0.3, 0.1, 0.3]`	0.4
is	`[0.7, 0.2, 0.1]`	0.2
sleeping	`[0.2, 0.5, 0.6]`	0.1

First element: 0.3 * 0.1 + 0.4 * 0.3 + 0.2 * 0.7 + 0.1 * 0.2 = 0.31
Final vector: `[0.31, 0.28, 0.26]`

Self-attention: Computing context vectors

Context vector for token 2.

Token j	Embedding	Attention weight a_{2j}
The	`[0.1, 0.5, 0.2]`	0.1
dog	`[0.3, 0.1, 0.3]`	0.7
is	`[0.7, 0.2, 0.1]`	0.1
sleeping	`[0.2, 0.5, 0.6]`	0.1

Suppose the attention weights change: more weight to the token itself. What is the new context vector?

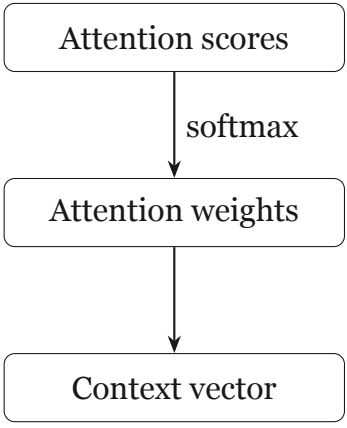
Self-attention: Computing context vectors

Context vector for token 2**.**

Token j	Embedding	Attention weight a_{2j}
The	`[0.1, 0.5, 0.2]`	0.1
dog	`[0.3, 0.1, 0.3]`	0.7
is	`[0.7, 0.2, 0.1]`	0.1
sleeping	`[0.2, 0.5, 0.6]`	0.1

First element: 0.1 * 0.1 + 0.7 * 0.3 + 0.1 * 0.7 + 0.1 * 0.2 = 0.31
Final vector: `[0.31, 0.19, 0.30]`

Self-attention

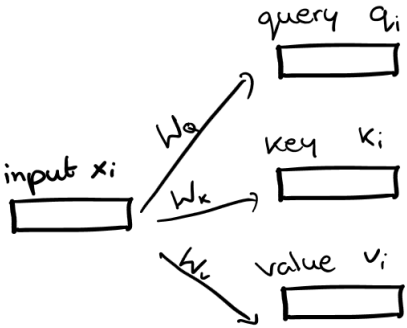


Setup (what do we need?)

We need **Query**, **Key** and **Value** projection matrices.

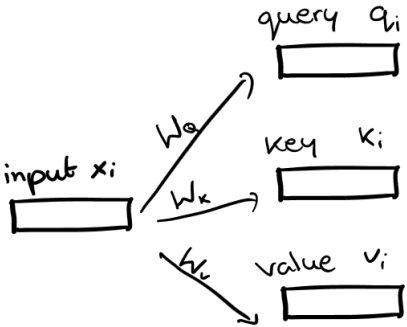
These are learned and shared across positions.

Self-attention: What do we need?



Every input vector $\mathbf{x}_i$ is mapped to a key $\mathbf{k}_i$, query $\mathbf{q}_i$, and value $\mathbf{v}_i$ vector.

Self-attention: What do we need?

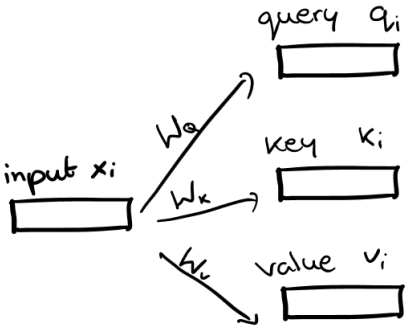


- **Query**: what the token is looking for
- **Key**: what the token represents
- **Value**: the information stored in the token

(or: query = what you're searching for, keys = index entries, values = the records returned)

Every input vector $\mathbf{x}_i$ is mapped to a key $\mathbf{k}_i$, query $\mathbf{q}_i$, and value $\mathbf{v}_i$ vector.

Self-attention: What do we need?



Each token is projected into three vectors using learned matrices:

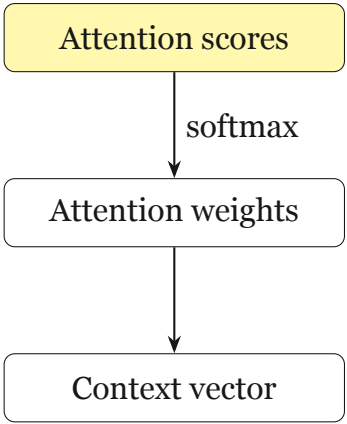
$$Q = XW_Q$$

$$K = XW_K$$

$$V = XW_V$$

Every input vector $\mathbf{x}_i$ is mapped to a key $\mathbf{k}_i$, query $\mathbf{q}_i$, and value $\mathbf{v}_i$ vector.

Computing self-attention scores



Attention scores: How *relevant* each token is to the token whose representation we're computing

Computing self-attention scores

Score between $\mathbf{x}_i$ (the *query* token) and $\mathbf{x}_j$ (a *key* token):

$$s(\mathbf{x}_i, \mathbf{x}_j) = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}}, \qquad \mathbf{q}_i = \mathbf{x}_i W_Q, \quad \mathbf{k}_j = \mathbf{x}_j W_K$$

- $\mathbf{q}_i$: query vector for token i (*what am I looking for?*)
- $\mathbf{k}_j$: key vector for token j
- d_k: the dimension of the query and key vectors

Computing self-attention scores

Score between $\mathbf{x}_i$ (the *query* token) and $\mathbf{x}_j$ (a *key* token):

$$s(\mathbf{x}_i, \mathbf{x}_j) = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}}, \qquad \mathbf{q}_i = \mathbf{x}_i W_Q, \quad \mathbf{k}_j = \mathbf{x}_j W_K$$

- $\mathbf{q}_i$: query vector for token i (*what am I looking for?*)
- $\mathbf{k}_j$: key vector for token j
- d_k: the dimension of the query and key vectors

W_Q and W_K are **learned**. The dot product measures how well token j matches what token i is looking for.

Computing self-attention scores

Score between $\mathbf{x}_i$ (the *query* token) and $\mathbf{x}_j$ (a *key* token):

$$s(\mathbf{x}_i, \mathbf{x}_j) = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}}, \qquad \mathbf{q}_i = \mathbf{x}_i W_Q, \quad \mathbf{k}_j = \mathbf{x}_j W_K$$

- $\mathbf{q}_i$: query vector for token i (*what am I looking for?*)
- $\mathbf{k}_j$: key vector for token j
- d_k: the dimension of the query and key vectors

W_Q and W_K are **learned**. The dot product measures how well token j matches what token i is looking for.

Question: Is $s(\mathbf{x}_i, \mathbf{x}_j) = s(\mathbf{x}_j, \mathbf{x}_i)$?

Computing self-attention scores

Score between $\mathbf{x}_i$ (the *query* token) and $\mathbf{x}_j$ (a *key* token):

$$s(\mathbf{x}_i, \mathbf{x}_j) = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}}, \qquad \mathbf{q}_i = \mathbf{x}_i W_Q, \quad \mathbf{k}_j = \mathbf{x}_j W_K$$

- $\mathbf{q}_i$: query vector for token i (*what am I looking for?*)
- $\mathbf{k}_j$: key vector for token j
- d_k: the dimension of the query and key vectors

W_Q and W_K are **learned**. The dot product measures how well token j matches what token i is looking for.

Question: Is $s(\mathbf{x}_i, \mathbf{x}_j) = s(\mathbf{x}_j, \mathbf{x}_i)$? *No,* $W_Q \neq W_K$

Computing self-attention scores

Score between $\mathbf{x}_i$ (the *query* token) and $\mathbf{x}_j$ (a *key* token):

$$s(\mathbf{x}_i, \mathbf{x}_j) = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}}, \qquad \mathbf{q}_i = \mathbf{x}_i W_Q, \quad \mathbf{k}_j = \mathbf{x}_j W_K$$

- $\mathbf{q}_i$: query vector for token i (*what am I looking for?*)
- $\mathbf{k}_j$: key vector for token j
- d_k: the dimension of the query and key vectors

W_Q and W_K are **learned**. The dot product measures how well token j matches what token i is looking for.

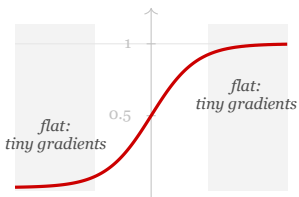
Question: Is $s(\mathbf{x}_i, \mathbf{x}_j) = s(\mathbf{x}_j, \mathbf{x}_i)$? *No,* $W_Q \neq W_K$

Why divide by $\sqrt{d_k}$?

$$s(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{q}_i \cdot \mathbf{k}_j$$

The problem 😕:

- The dot product sums over d_k dimensions, so the variance grows as the number of dimensions grows
- After the softmax, attention weights become very peaked → small gradients



(showing the sigmoid instead of the softmax for simplicity)

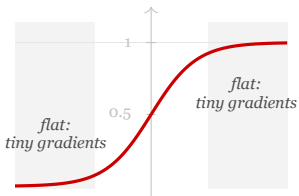
Why divide by $\sqrt{d_k}$?

$$s(\mathbf{x}_i, \mathbf{x}_j) = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}}$$

The problem 😕:

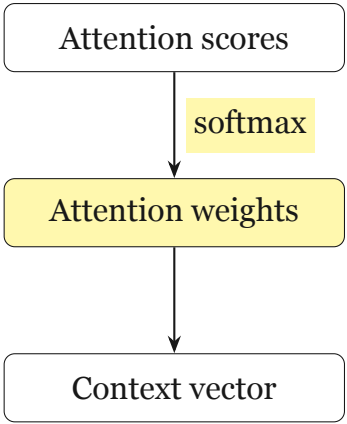
- The dot product sums over d_k dimensions, so the variance grows as the number of dimensions grows
- After the softmax, attention weights become very peaked → small gradients

Dividing by $\sqrt{d_k}$ stops the scores from growing with the dimension.



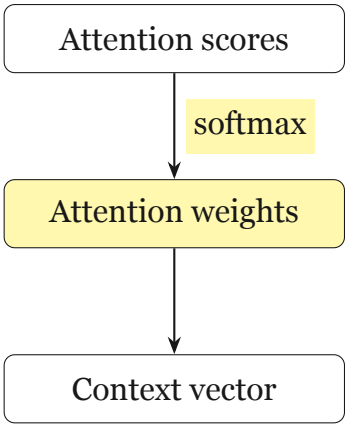
(showing the sigmoid instead of the softmax for simplicity)

How do we compute attention weights?



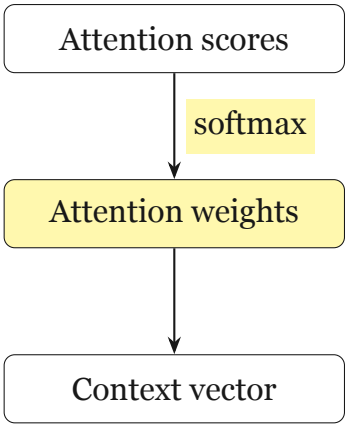
Why do we want the attention weights to sum to 1?

How do we compute attention weights?



- Forms a **probability distribution** over tokens, forcing competition for attention
- Keeps the **context vector scale** bounded, improving training stability

How do we compute attention weights?

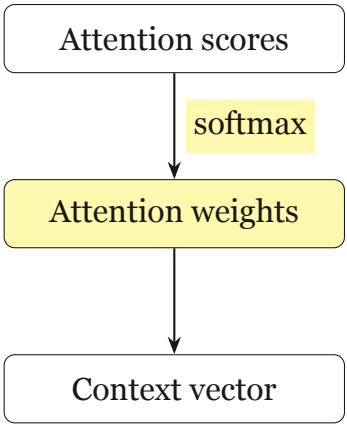


Normalize the attention scores (s_{ij}) using the softmax to obtain the attention weights (a_{ij}), ensuring that $a_{ij} \geq 0$ and $\sum_j a_{ij} = 1$

$$\boldsymbol{a}_i = \text{softmax}(\mathbf{s}_i)$$

$$a_{ij} = \big(\text{softmax}(\mathbf{s}_i)\big)_j = \frac{e^{s_{ij}}}{\sum_{j'} e^{s_{ij'}}}.$$

How do we compute attention weights?



Aside: Why softmax and not plain normalization by dividing by the sum of the values $\frac{s_{ij}}{\sum_{j'} s_{ij'}}$.

Problem: It does not handle arbitrary real-valued scores. E.g., `[-2, 0, 1]` would produce `[2,0,-1]`. The values sum to 1, but aren't valid probabilities.

Putting things together (more formally)

For one token i:

$$\mathbf{q}_i = \mathbf{x}_i W_Q$$

$$s_{ij} = \frac{\langle \mathbf{q}_i, \mathbf{k}_j \rangle}{\sqrt{d_k}} \quad \text{for } j = 1, \ldots, n$$

$$a_{ij} = \text{softmax}\big(s_{i1}, \ldots, s_{in}\big)_j$$

$$\mathbf{c}_i = \sum_{j=1}^{n} a_{ij}\, \mathbf{v}_j$$

Putting things together (more formally)

For one token i:

$$\mathbf{q}_i = \mathbf{x}_i W_Q$$

$$s_{ij} = \frac{\langle \mathbf{q}_i, \mathbf{k}_j \rangle}{\sqrt{d_k}} \quad \text{for } j = 1, \ldots, n$$

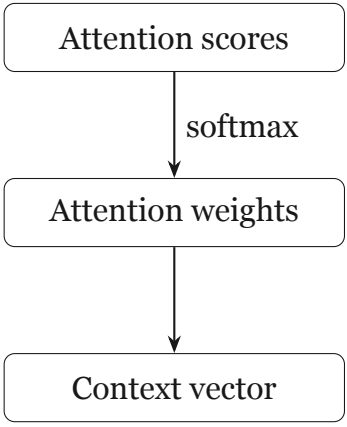
$$a_{ij} = \text{softmax}\big(s_{i1}, \ldots, s_{in}\big)_j$$

$$\mathbf{c}_i = \sum_{j=1}^{n} a_{ij}\, \mathbf{v}_j$$

All tokens at once:

$$\text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)$$

Self-attention



Allows the model to pay more attention to certain words/tokens in the sequence

Attention scores: How *relevant* each token is to the token whose representation we're computing

Vaswani et al., 2017:

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

Example: How a token uses attention

Example: *Jack said **he** was tired.*

We want to compute a representation for the token **“he”**.

- Each token is represented by a vector $\mathbf{x}_i$.
- For “he” (the 3rd token): $\mathbf{x}_3$.
- We project it to a **query vector**:

$$\mathbf{q}_3 = \mathbf{x}_3 W_Q$$

Example: How a token uses attention

Example: *Jack said **he** was tired.*

The query asks: *Which other tokens are relevant for understanding "he"?*

- Compare the query $\mathbf{q}_3$ with all **key vectors**

$$\mathbf{k}_j = \mathbf{x}_j W_K$$

- The attention **scores** $\frac{\mathbf{q}_3 \cdot \mathbf{k}_j}{\sqrt{d_k}}$ are then turned into attention **weights** (using the softmax).
- Result: the representation of "*he*" will (probably) place more weight on the **value vector** of "*Jack*".

Multi-head self-attention

Hmm… *he* may need to attend to *Jack* **and** to *tired* at the same time.

- Run h attention "heads" in parallel; each head m has its own $W_Q^{(m)}, W_K^{(m)}, W_V^{(m)}$, for $m = 1 \ldots h$
- Each head can learn a different pattern: syntax, coreference, nearby positions, etc.
- Concatenate the head outputs and project back to dimension d

Each head works in a smaller dimension: $d_k = d/h$.

Example: BERT base has 12 layers $\times$ 12 heads $=$ 144 attention patterns.

Note: More recent LLMs use **grouped query attention (GQA)**, where several query heads share one K/V head

Multi-head self-attention

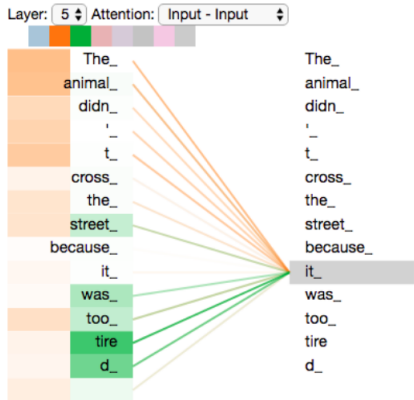


Figure: Figure from The Illustrated Transformer

Self-attention: full example

token	query	key	value
the	`[1,0,1]`	`[1,0,1]`	`[.5,.1,.0]`
dog	`[1,2,0]`	`[0,2,1]`	`[.4,.3,.2]`
is	`[0,1,1]`	`[1,1,0]`	`[.2,.6,.3]`
sleeping	`[2,1,0]`	`[2,2,1]`	`[.0,.2,.8]`

Every token is mapped to a query, key and value vector ($d_k = 3$).

Self-attention: full example

token	query	key	value
the	[1,0,1]	[1,0,1]	[.5,.1,.0]
dog	[1,2,0]	[0,2,1]	[.4,.3,.2]
is	[0,1,1]	[1,1,0]	[.2,.6,.3]
sleeping	[2,1,0]	[2,2,1]	[.0,.2,.8]

Compute the context vector for "*dog*".

Self-attention: full example

token	query	key	value	q·k	$\div\sqrt{3}$
the	`[1,0,1]`	`[1,0,1]`	`[.5,.1,.0]`	1	0.58
dog	`[1,2,0]`	`[0,2,1]`	`[.4,.3,.2]`	4	2.31
is	`[0,1,1]`	`[1,1,0]`	`[.2,.6,.3]`	3	1.73
sleeping	`[2,1,0]`	`[2,2,1]`	`[.0,.2,.8]`	6	3.46

$[1,2,0] \cdot [2,2,1] = 1{\cdot}2 + 2{\cdot}2 + 0{\cdot}1 = 6, \qquad 6/1.73 = 3.46$

Self-attention: full example

token	query	key	value	q·k	$\div\sqrt{3}$		weight
the	`[1,0,1]`	`[1,0,1]`	`[.5,.1,.0]`	1	0.58		0.04
dog	`[1,2,0]`	`[0,2,1]`	`[.4,.3,.2]`	4	2.31		0.20
is	`[0,1,1]`	`[1,1,0]`	`[.2,.6,.3]`	3	1.73		0.11
sleeping	`[2,1,0]`	`[2,2,1]`	`[.0,.2,.8]`	6	3.46		0.65

"*dog*" attends most to "*sleeping*".

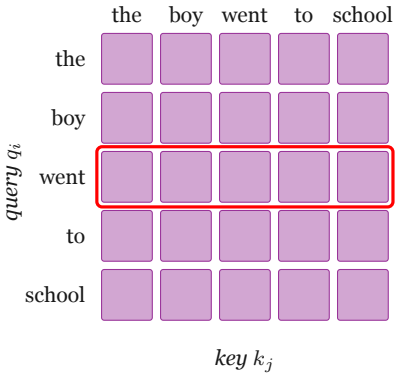
Self-attention: full example

token	query	key	value	q·k	$\div\sqrt{3}$		weight
the	`[1,0,1]`	`[1,0,1]`	`[.5,.1,.0]`	1	0.58		0.04
dog	`[1,2,0]`	`[0,2,1]`	`[.4,.3,.2]`	4	2.31		0.20
is	`[0,1,1]`	`[1,1,0]`	`[.2,.6,.3]`	3	1.73		0.11
sleeping	`[2,1,0]`	`[2,2,1]`	`[.0,.2,.8]`	6	3.46		0.65

$$0.04 \cdot \texttt{[.5,.1,.0]} + 0.20 \cdot \texttt{[.4,.3,.2]} + 0.11 \cdot \texttt{[.2,.6,.3]} + 0.65 \cdot \texttt{[.0,.2,.8]}$$

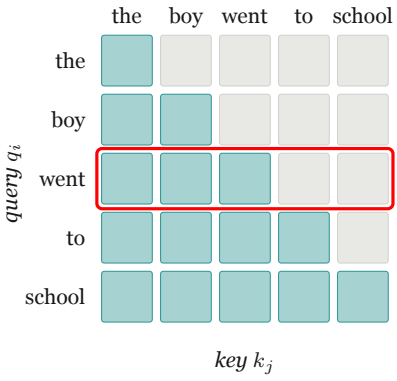
$$\mathbf{c}_{\text{dog}} = \texttt{[0.12, 0.26, 0.59]}$$

Bidirectional attention



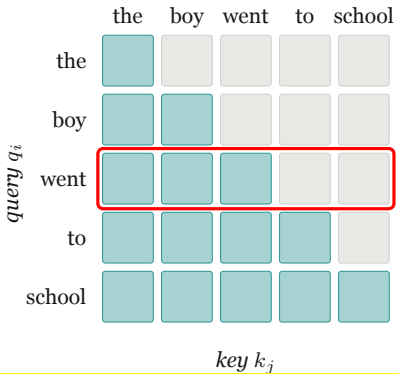
- Each **row** is a query token
- Each **column** is a key
- *went* can attend to every token, including *school*, so *school* can influence the contextual representation of went (and vice versa)

Causal attention



- Future tokens are masked out (grey)
- *went* may attend only to *the*, *boy*, *went*
- Used for models that can generate text (decoders)

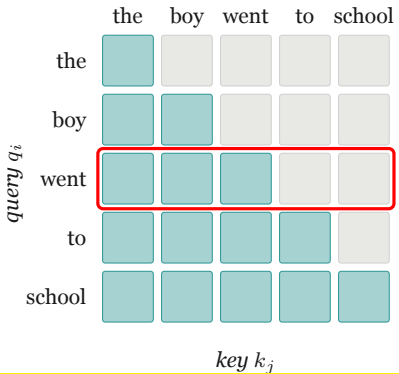
Causal attention



- Future tokens are masked out (grey)
- *went* may attend only to *the*, *boy*, *went*
- Used for models that can generate text (decoders)

Question: For 5 tokens, how many attention computations does the encoder do vs the decoder?

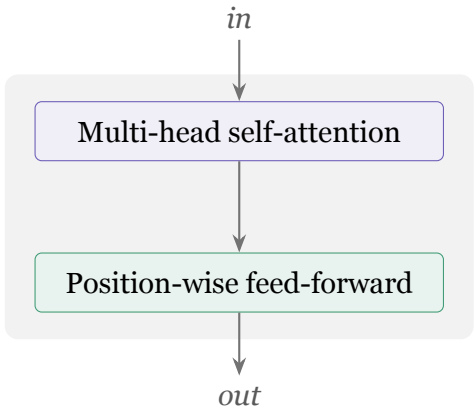
Causal attention



- Future tokens are masked out (grey)
- *went* may attend only to *the*, *boy*, *went*
- Used for models that can generate text (decoders)

Question: For 5 tokens, how many attention computations does the encoder do vs the decoder? *25 vs. 15*

Transformer Block



Goal:
Learn good contextual representations

Feedforward Layer

After attention mixes information across tokens, the feedforward network (FFN) processes each token *independently*.

$$\text{FFN}(\mathbf{x}_1),\ \text{FFN}(\mathbf{x}_2),\ \ldots,\ \text{FFN}(\mathbf{x}_n)$$

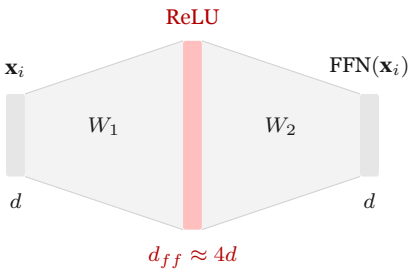
For each token representation $\mathbf{x}_i$:

$$\text{FFN}(\mathbf{x}_i) = \text{ReLU}(\mathbf{x}_i W_1 + \mathbf{b}_1)W_2 + \mathbf{b}_2$$

The parameters are the same for each token position i, but different across layers.

Feedforward Layer

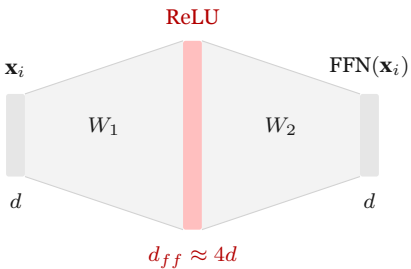
$$\text{FFN}(\mathbf{x}_i) = \text{ReLU}(\mathbf{x}_i W_1 + \mathbf{b}_1)W_2 + \mathbf{b}_2$$



- Input and output both have dimension d
- d is fixed for the *whole* model, so it's easier to stack Transformer blocks.

Feedforward Layer

$$\text{FFN}(\mathbf{x}_i) = \text{ReLU}(\mathbf{x}_i W_1 + \mathbf{b}_1)W_2 + \mathbf{b}_2$$



- Modern LLMs replace the ReLU with a gated unit: SwiGLU.
- The input is projected twice in parallel: *what* to pass on, and *how much* of it gets through.

Positional embeddings

Self-attention (recap)

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

Positional embeddings

Self-attention (recap)

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

Question: Take the formula above. Consider the following two sentences: *the dog bites the man* and *the man bites the dog*. Would the representation of *dog* be the same or different in these two cases?

Positional embeddings

Self-attention (recap)

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

No! Because:

- The computation depends only on pairwise similarities ($QK^\top$)
- Does not depend on the absolute position or order of tokens

See also Positional embeddings in transformers EXPLAINED (video) and Position Information in Transformers: An Overview, 2022.

Positional embeddings

Self-attention (recap)

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

No! Because:

- The computation depends only on pairwise similarities ($QK^\top$)
- [illegible]

We need to explicitly encode the order of tokens.

See also Positional embeddings in transformers EXPLAINED (video) and Position Information in Transformers: An Overview, 2022.

Positional Embeddings

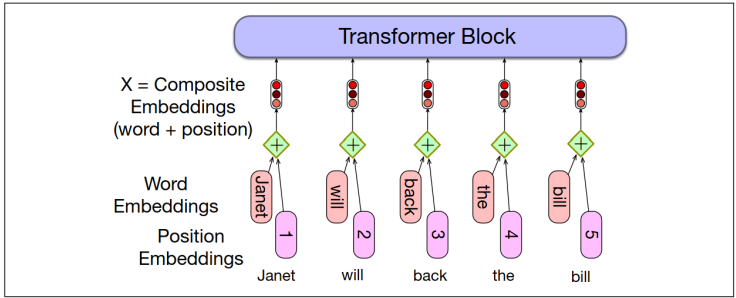


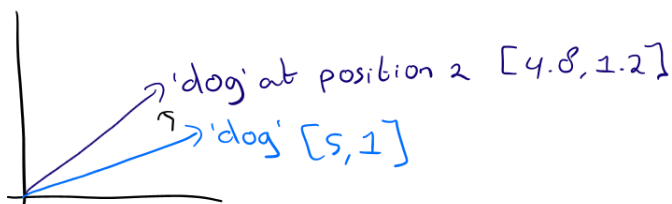
Figure 8.14 A simple way to model position: add an embedding of the absolute position to the token embedding to produce a new embedding of the same dimensionality.

Figure: Figure from Speech and Language Processing, Jurafsky and Martin, Chapter 8

Positional embeddings

Types

- *Absolute* vs. *relative* positional information
- *Learned* vs. *fixed*
- *Added* vs. *concatenated*
- Etc.



Rotary Position Embeddings (RoPE)

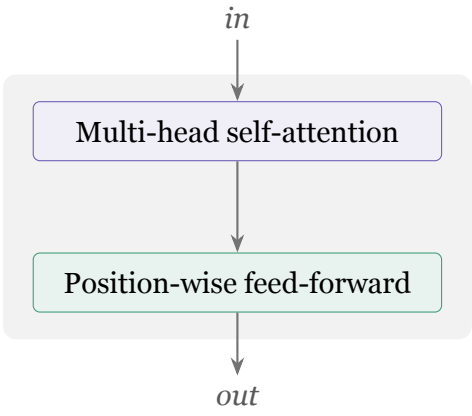
RoPE is a positional encoding method used by many modern LLMs (e.g., Open AI's gpt-oss and Llama).

- Encodes position by **rotating query and key vectors**; the amount of rotation depends on the token's position.
- Enables the attention mechanism to capture the **relative distance between tokens**.
- Unlike standard positional embeddings, RoPE does **not add a positional vector** to the token embedding.

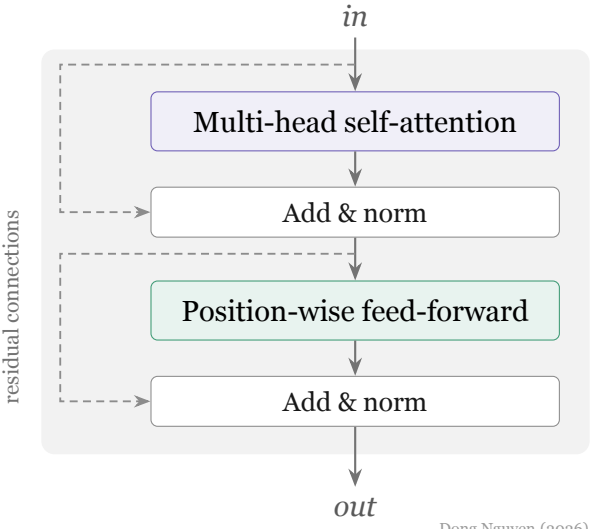
Key idea: Position is incorporated directly into the attention mechanism.

RoFormer: Enhanced Transformer with Rotary Position Embedding, Su et al., 2021 [link]

Transformer Block



Transformer Block



Layer Normalization (LayerNorm)

Goal: Keep token representations at a stable scale during training.

Suppose the token representations ($X \in \mathbb{R}^{n \times d}$) are as follows, where rows are tokens:

$$X = \begin{bmatrix} \leftarrow \mathbf{x}_1 \rightarrow \\ \leftarrow \mathbf{x}_2 \rightarrow \\ \vdots \\ \leftarrow \mathbf{x}_n \rightarrow \end{bmatrix}$$

Layer Normalization (LayerNorm)

Goal: Keep token representations at a stable scale during training.

Suppose the token representations ($X \in \mathbb{R}^{n \times d}$) are as follows, where rows are tokens:

$$X = \begin{bmatrix} \leftarrow \mathbf{x}_1 \rightarrow \\ \leftarrow \mathbf{x}_2 \rightarrow \\ \vdots \\ \leftarrow \mathbf{x}_n \rightarrow \end{bmatrix} \longrightarrow \mathrm{LN}(X) = \begin{bmatrix} \mathrm{LN}(\mathbf{x}_1) \\ \mathrm{LN}(\mathbf{x}_2) \\ \vdots \\ \mathrm{LN}(\mathbf{x}_n) \end{bmatrix}$$

LayerNorm normalizes the d dimensions of each token's vector **independently** (i.e., not across tokens).

Layer Normalization (LayerNorm)

Representation for token i:

$$\mathbf{x}_i = [x_{i1}, x_{i2}, \ldots, x_{id}].$$

Compute the following:

$$\mu_i = \frac{1}{d}\sum_{j=1}^{d} x_{ij}, \qquad \sigma_i^2 = \frac{1}{d}\sum_{j=1}^{d}(x_{ij} - \mu_i)^2.$$

Layer Normalization (LayerNorm)

First normalize the token representation (ϵ is for numerical stability):

$$\hat{\mathbf{x}}_i = \frac{\mathbf{x}_i - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}.$$

After normalization:

$$\text{mean}(\hat{\mathbf{x}}_i) \approx 0, \qquad \text{variance}(\hat{\mathbf{x}}_i) \approx 1.$$

Then apply learned scale ($\boldsymbol{\gamma}$) and offset ($\boldsymbol{\beta}$) parameters.

$$\text{LN}(\mathbf{x}_i) = \boldsymbol{\gamma} \odot \hat{\mathbf{x}}_i + \boldsymbol{\beta}$$

The same $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ are used for all token positions within a LayerNorm layer.

Layer Normalization (LayerNorm)

First normalize the token representation (ϵ is for numerical stability):

$$\hat{\mathbf{x}}_i = \frac{\mathbf{x}_i - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}.$$

After normalization:

$$\text{mean}(\hat{\mathbf{x}}_i) \approx 0, \qquad \text{variance}($$

Then apply learned scale ($\boldsymbol{\gamma}$) and offset ($\boldsymbol{\beta}$) parameters.

$$\text{LN}(\mathbf{x}_i) = \boldsymbol{\gamma} \odot \hat{\mathbf{x}}_i + \boldsymbol{\beta}$$

The same $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ are used for all token positions within a LayerNorm layer.

Note: most recent LLMs use **RMSNorm**: similar idea but without the mean centering and without $\boldsymbol{\beta}$.

LayerNorm Example

LayerNorm is applied **per row (per token)**:

$$X = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \end{bmatrix} \qquad (2 \text{ tokens, hidden size } d = 3).$$

For token 1 (ϵ = 0 for simplicity):

$$\mu_1 = \frac{1+2+3}{3} = 2$$

$$\sigma_1^2 = \frac{(1-2)^2 + (2-2)^2 + (3-2)^2}{3} = \frac{2}{3}.$$

$$\hat{\mathbf{x}}_1 = \frac{[1,2,3] - 2}{\sqrt{2/3}} \approx [-1.225, 0, 1.225].$$

LayerNorm Example

LayerNorm is applied **per row (per token)**:

$$X = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \end{bmatrix} \qquad (2 \text{ tokens, hidden size } d = 3).$$

For token 1 (ϵ = 0 for simplicity):

$$\hat{\mathbf{x}}_1 = \frac{[1, 2, 3] - 2}{\sqrt{2/3}} \approx [-1.225, 0, 1.225].$$

Question: What about token 2?

LayerNorm Example

LayerNorm is applied **per row (per token)**:

$$X = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \end{bmatrix} \qquad (2 \text{ tokens, hidden size } d = 3).$$

For token 1 (ϵ = 0 for simplicity):

$$\hat{\mathbf{x}}_1 = \frac{[1, 2, 3] - 2}{\sqrt{2/3}} \approx [-1.225, 0, 1.225].$$

Same!

$$\hat{\mathbf{x}}_2 = \frac{[2, 4, 6] - 4}{\sqrt{8/3}} \approx [-1.225, 0, 1.225].$$

Residual connections

Suppose a Transformer layer computes some transformation $F(\mathbf{x})$.

Without a residual connection:

$$\mathbf{x}_{\text{new}} = F(\mathbf{x})$$

Residual connections

Suppose a Transformer layer computes some transformation $F(\mathbf{x})$.

Without a residual connection:

$$\mathbf{x}_{\text{new}} = F(\mathbf{x})$$

With a **residual connection**:

$$\mathbf{x}_{\text{new}} = \mathbf{x} + F(\mathbf{x})$$

- The layer does not have to replace the existing representation.
- Instead, it computes an **update** to the representation.
- Information can pass directly through many layers.

Residual connections

Each sublayer in the Transformer has a **residual connection**.

$$x_{\text{new}} = \text{LayerNorm}\left(x + \text{Sublayer}(x)\right)$$

- x: representation entering the sublayer
- Sublayer(x): e.g., attention or feed-forward network
- The original representation x is **added back**

Attention Is All You Need, Vaswani et al., 2017 [link]

Residual stream

The attention and FFN sublayers each "*read*" their input from the **residual stream** and then "*write*" their result to the residual stream (Elhage et al.)

Attention is considered the **token-mixing** component of the Transformers architecture, as it mixes information from neighboring token streams into the current stream.

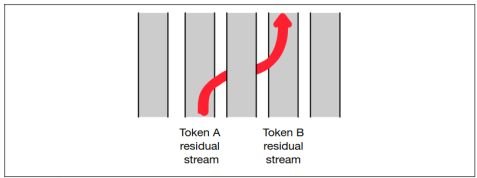


Figure 8.8 An attention head can move information from token A's residual stream into token B's residual stream.

Figure: Figure from Speech and Language Processing, Jurafsky and Martin, Chapter 8

The original Transformer vs. now

	Vaswani et al. (2017)	**Modern LLMs**
Normalization	LayerNorm, *after*	RMSNorm, *before*
Feed-forward	ReLU	SwiGLU
Positions	absolute (added at the input)	RoPE (inside attention)
Attention	multi-head	grouped-query

See also The Big LLM Architecture Comparison by Sebastian Raschka

Basics

	Components	What it does
Input	tokenization, embeddings	from text to vectors
Transformer block ($\times N$)	attention, positional embeddings, layer normalization, feedforward layer, residual connections	building contextual representations
Output	LM head, decoding	making word predictions

LM head

(recap) A **language model** can estimate the probability of the **next word** given previous words:

$$p(\texttt{great} \mid \texttt{this is})$$

LM head

Language modeling (LM) head: Takes the output of the final transformer layer at token i and predicts the next token at position $i + 1$.

Weight tying: many models reuse the *input* embedding matrix ($E^\top$)

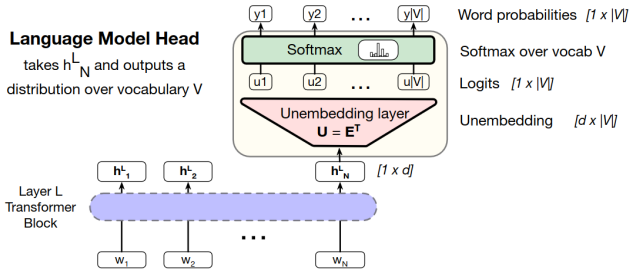


Figure: Figure 7.15 from Speech and Language Processing, Jurafsky and Martin, Chapter 7

Compared with an n-gram model

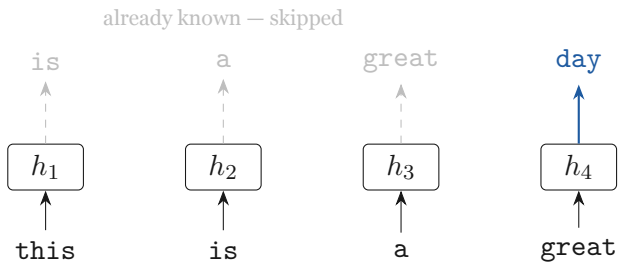
An n-gram model **counts**: how often does `great` follow `this is`?

- Every context has its own entry with its frequency in a table.
- Even though `this is` and e.g., `that was` are similar, they are completely distinct in the table.

Now:

- Similar contexts get similar vectors, so they get similar predictions.
- Prediction is based on the *whole context window* (e.g., thousands of tokens), not just last $n - 1$ words.

Inference: only run the head on the very last token



Aside: KV cache

Note: the keys and values of previous tokens *never change*, due to the causal attention.

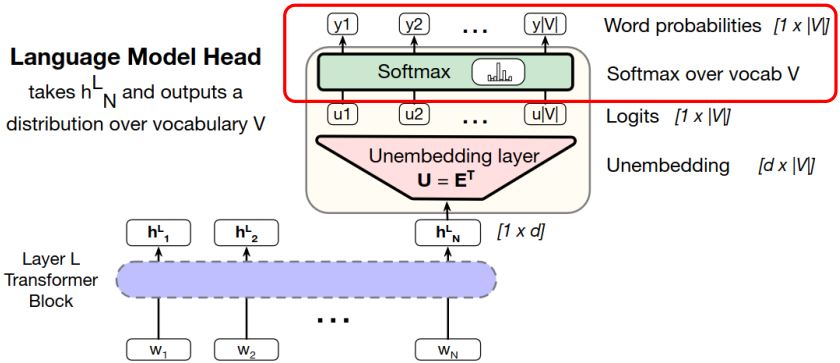
- So we can **cache** k_j, v_j for every previous token
- Each new token computes only its own q, k, v and attends to the cache

Much faster! 😄
But... uses extra memory to store past information; cache grows as context grows 😕

See also: KV Caching Explained: Optimizing Transformer Inference Efficiency.

Decoding

The head outputs a distribution, not a token.
Decoding = How do we choose the next token?



Greedy decoding

At each time step *t*, choose the word (or in practice: token) with the highest probability given the previous words ($w_{<t}$).

$$\arg\max_{w \in V} P(w \mid w_{<t})$$

Why is this not often used?

Top-*k* Sampling

Steps:

- For each word w, compute its probability given the context ($p(w_t|\mathbf{w}_{<t})$)
- Sort all words. Only keep the *k* most probable words.
- Renormalize the remaining *k* words to a probability distribution.
- Sample a word from the remaining *k* words according to its probability.

Note: when *k=1* this is the same as greedy decoding.
In the Transformers library, k is by default 50. [link]

Top-k sampling

Suppose $k = 3$

Token	Prob.
cat	0.40
dog	0.25
then	0.15
water	0.10
book	0.06
ran	0.04

Top-k sampling

Suppose $k = 3$

Token	Prob.
cat	0.40
dog	0.25
then	0.15
water	0.10
book	0.06
ran	0.04

Take the words and renormalize (so the probabilities sum up to 1).

Token	Prob.
cat	0.40/0.8 = 0.50
dog	0.25/0.8 = 0.31
then	0.15/0.8 = 0.19

Top-p Sampling

Similar to top-k sampling, but select the smallest set of tokens (V_{sel}) such that:

$$\sum_{w \in V_{sel}} P(w \mid \mathbf{w}_{<t}) \geq p$$

$p = 0.6$

Token	Prob.
cat	0.40
dog	0.25
then	0.15
water	0.10
book	0.06
ran	0.04

Top-p Sampling

Similar to top-k sampling, but select the smallest set of tokens (V_{sel}) such that:

$$\sum_{w \in V_{sel}} P(w \mid \mathbf{w}_{<t}) \geq p$$

Like before, renormalize and then sample.

cat: $0.40/0.65 = 0.62$
dog: $0.25/0.65 = 0.38$

$p = 0.6$

Token	Prob.
cat	0.40
dog	0.25
then	0.15
water	0.10
book	0.06
ran	0.04

Temperature

To control the randomness (–> creativity, diversity) of the output, divide the logits z_i by a temperature T (where $T > 0$).

Softmax with Temperature T:

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

Temperature

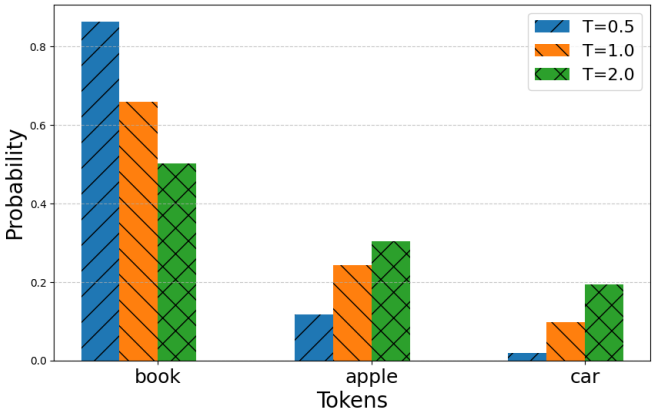
To control the randomness (–> creativity, diversity) of the output, divide the logits z_i by a temperature T (where $T > 0$).

Softmax with Temperature T:

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

- Common values for temperature: 0.7, 1.
- Technically temperature of 0 is not possible. APIs that support a temperature of 0 then revert back to choosing the most likely word (greedy decoding).

Temperature



As the temperature approaches 0, the probability of the most likely word approaches 1.

Question about decoding

You want an LLM to (a) extract numbers from invoices, and (b) brainstorm songs for a social event.

Which temperature would you pick for each: $T = 0.1$ or $T = 1.2$?

Question about decoding

You want an LLM to (a) extract numbers from invoices, and (b) brainstorm songs for a social event.

Which temperature would you pick for each: $T = 0.1$ or $T = 1.2$?

- (a) Extraction: low temperature (we want the most likely, reproducible answer)
- (b) Brainstorming: higher temperature (more creativity)

Agenda

Goal: Introduce you to the core components of Large Language Models

Roadmap:

- Historical context
- Core components of Transformer models
- Transformer models
- Training LLMs: pretraining and instruction tuning

Transformer models

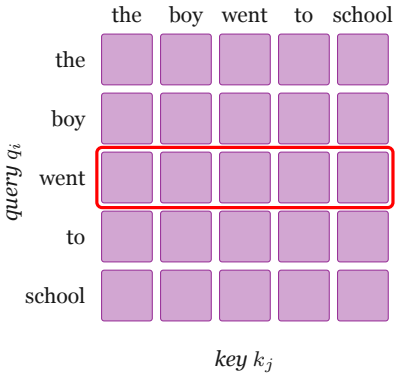
Encoders vs. Decoders

BERT is called an *encoder*. GPT is called a *decoder*. The original 2017 Transformer is an *encoder-decoder*. 🤔

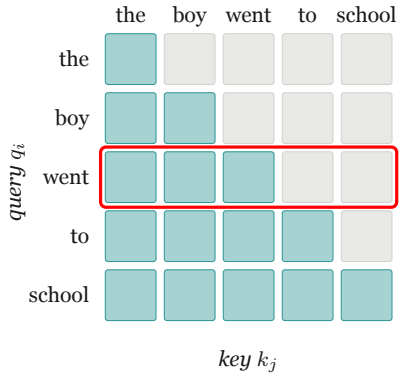
They are actually very similar! The key difference is what tokens can be looked at ("**attended** to").

Recap: attention

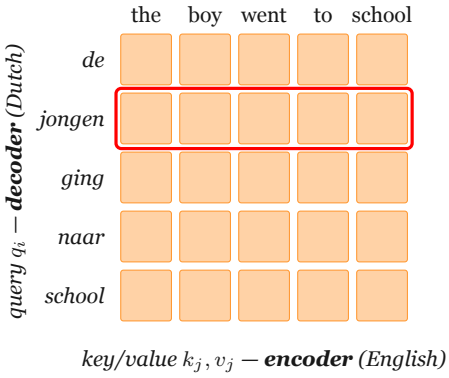
Encoder: bidirectional attention



Decoder: causal attention

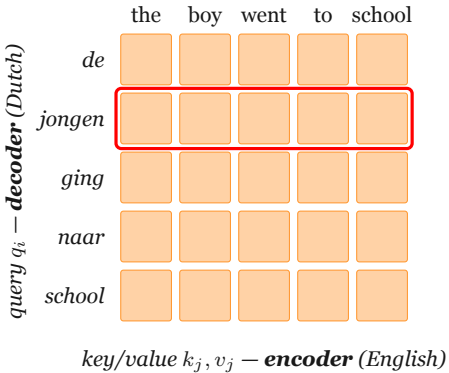


Cross-attention



- Example: translating English → Dutch
- Queries come from the **decoder**, keys and values from the **encoder**
- *jongen* may attend to the *whole* English sentence

Cross-attention



- Example: translating English → Dutch
- Queries come from the **decoder**, keys and values from the **encoder**
- *jongen* may attend to the *whole* English sentence

Note: *The grid doesn't need to be a square: the two sentences can have different lengths; even different modalities are possible!*

Where do Q, K, V come from?

	Q **from**	K, V **from**	**Mask**
Encoder self-attention	encoder	encoder	none
Decoder self-attention	decoder	decoder	causal
Cross-attention	decoder	encoder	none

Encoder-only models

Examples: BERT, RoBERTa, DeBERTa, ModernBERT, etc.

- Gives strong representations of the *given* input 😄
- However, cannot be used for text generation 😕

Contextualized word representations

Yesterday: an embedding for **each word (type)**.

*Today, I went to the **bank** to deposit a check.*

bank	0.52	0.48	-0.01	···	0.28

*The hut is located near the **bank** of the river.*

bank	-0.27	0.28	-0.07	···	0.82

Contextualized word representations

Key idea: Have embeddings for each **word token**

Previously (e.g., word2vec):

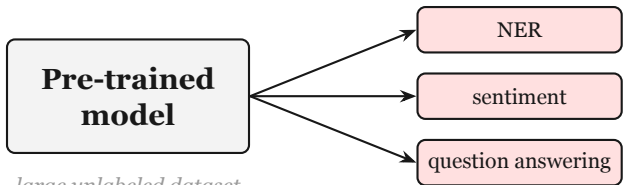
- One embedding for each word **type**
- A table where each word is mapped to a vector.

Now:

- One embedding for each work **token**
- Embeddings for a token are created based on the context
- There is *no single* embedding for a word anymore.

Pre-train and fine-tune paradigm

- **Pre-train** (once) on large text datasets to learn general language representations, linguistic patterns, and world knowledge.
- **Fine-tune** the pre-trained model on task-specific labeled data to adapt it to a specific downstream task.



Universal Language Model Fine-tuning for Text Classification, Howard and Ruder, 2018 [link]
Deep Contextualized Word Representations, Peters et al., 2018 [link]

Masked language modeling (MLM)

An encoder sees the whole sequence (bidirectional attention), so next-token prediction would be trivial.

Instead: hide (using `[MASK]`) some tokens and predict them by looking from **both sides**.

`the boy [MASK] to [MASK]`

Masked language modeling (MLM)

An encoder sees the whole sequence (bidirectional attention), so next-token prediction would be trivial.

Instead: hide (using `[MASK]`) some tokens and predict them by looking from **both sides**.

`the boy [MASK] to [MASK]`

- Select **x%** (e.g., 15) of the tokens; the loss is only computed on these tokens.
- At each one: predict the *original* token, as a softmax over the whole vocabulary. Then calculate cross-entropy loss.

BERT

BERT was proposed by Devlin et al. (NAACL 2019)

Two pre-training tasks:

- Masked language modeling: sample 15% of the tokens. 80% are masked, 10 % replaced with a random token, 10% kept the same.
- Next sentence prediction

Then fine-tune for your specific task!

Training BERT

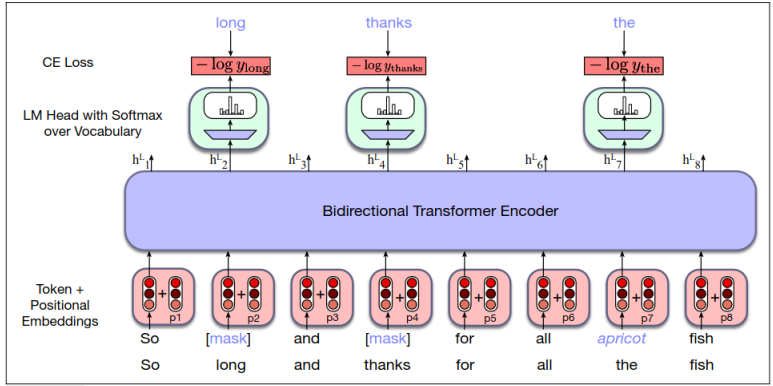


Figure 9.4 Masked language model training. In this example, three of the input tokens are selected, two of which are masked and the third is replaced with an unrelated word. The probabilities assigned by the model to these three items are used as the training loss. The other 5 tokens don't play a role in training loss.

Figure: Figure from Speech and Language Processing, Jurafsky and Martin, Chapter 9

Training BERT

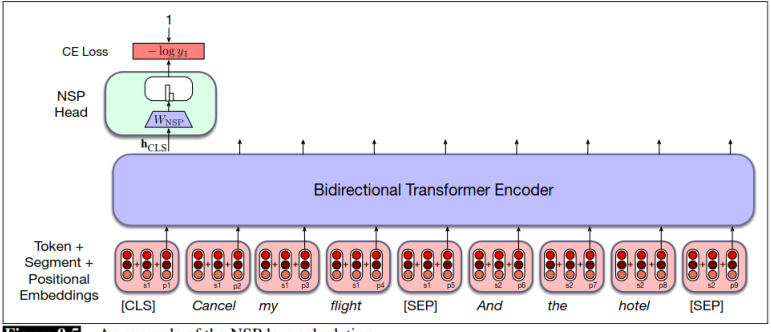


Figure 9.5 An example of the NSP loss calculation.

Figure: Figure from Speech and Language Processing, Jurafsky and Martin, Chapter 9

Sequence classification with BERT

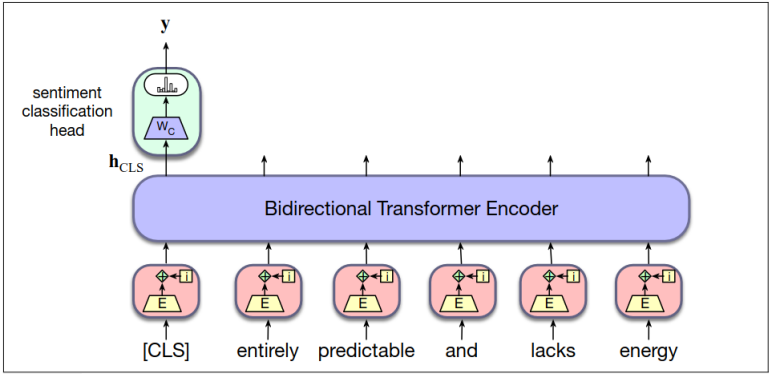


Figure 9.6 Sequence classification with a bidirectional transformer encoder. The output vector for the [CLS] token serves as input to a simple classifier.

Figure: Figure from Speech and Language Processing, Jurafsky and Martin, Chapter 9

Sequence classification with BERT

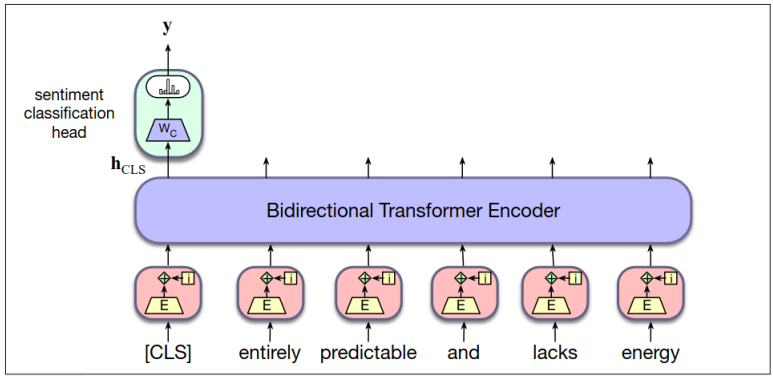


Figure 9.6 Sequence classification with a bidirectional transformer encoder. The output vector for the [CLS] token serves as input to a simple classifier.

Question: Why is BERT not suitable for generating text?

Decoder-only models

Examples: GPT, Llama, Mistral, Qwen.

- **Objective:** next-token prediction
- Also known as *autoregressive* language models or *causal* language models
- Can generate text, easy to scale, strong capabilities 😄
- Only sees left context; so representations are usually weaker than an encoder's 😕

GPT model family

- GPT was originally introduced in the paper "*Improving Language Understanding by Generative Pre-Training*" by Radford et al. (2018), which introduced generative pre-training; 12 Transformer layers
- GPT-2 was introduced in by Radford et al. (2019); 48 Transformer layers and 1.5B parameters.
- GPT-3 was introduced in Brown et al. (2020); 96 Transformer layers and 175B parameters.
- **GPT architecture:** only the decoder part of the original Transformer architecture.

Improving Language Understanding by Generative Pre-Training, Radford et al., 2018 [link]
Language Models are Unsupervised Multitask Learners, Radford et al., 2019 [link]
Language Models are Few-Shot Learners, Brown et al., 2020 [link]

Encoder-decoder models

- Encoder: bidirectional attention over the input sequence
- Cross attention: decoder queries attend to all encoder output states
- Decoder: causal attention over previously generated tokens
- Especially good with complex relationships between sequences, e.g., machine translation, or different modalities (speech $\rightarrow$ text).

Transformers

Introduced by Vaswani et al., "*Attention Is All You Need*" (2017) for the task of machine translation (English -> German and French).

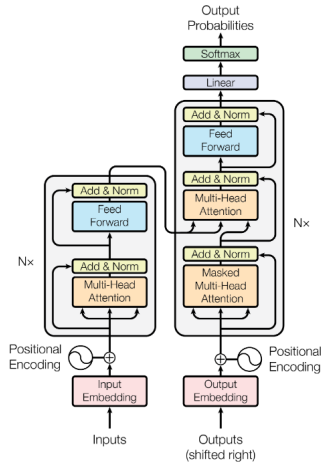


Figure 1: The Transformer - model architecture.

T5: Text-to-Text Transfer Transformer

Key idea: T5 casts *every NLP task* as **text-to-text**: both the input and output are sequences of tokens.

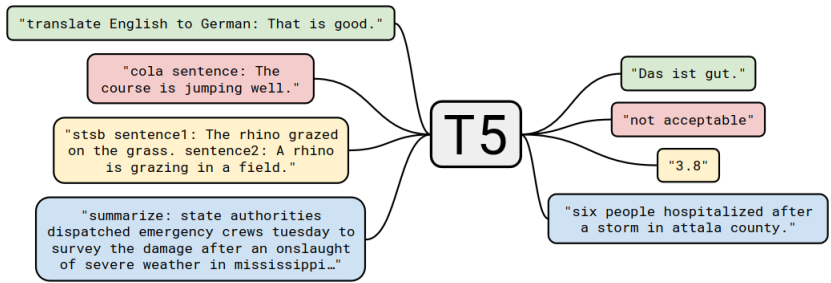


Figure: Figure from Raffel et al. 2020, link

T5: Text-to-Text Transfer Transformer

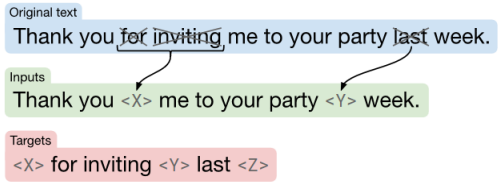


Figure: Figure 2 from Raffel et al. 2020, link

Note: How does this compare to BERT? Both corrupt input text. BERT predicts the masked tokens (i.e., classification). T5 generates a full sequence with missing spans.

Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer , Raffel et al., 2020 [link]

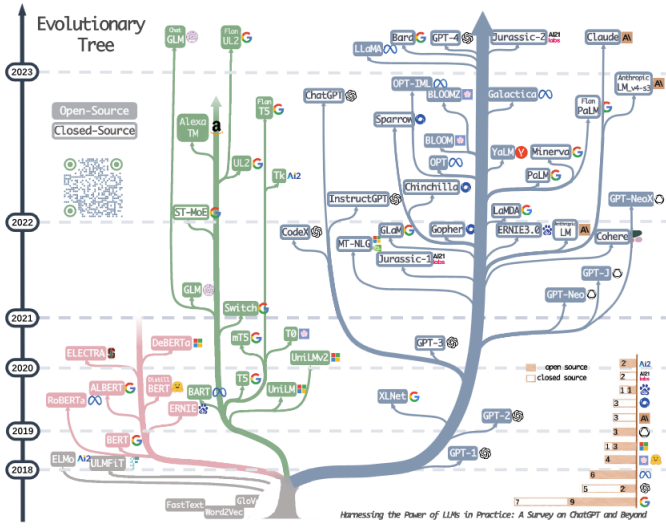


Figure: Figure 1 from Harnessing the Power of LLMs in Practice: A Survey on ChatGPT and Beyond, Yang et al., 2024; Transformer-based models are shown in non-grey colors: decoder-only models in the blue branch, encoder-only models in the pink branch, and encoder-decoder models in the green branch.

Recap

	Encoder	Decoder	Encoder-decoder
Attention	bidirectional	causal	bidirectional (encoder) and causal (decoder)
Cross-attention	no	no	yes
Text generation	no	yes	yes
Typical training objective	MLM	next token	denoising
Example model	BERT	GPT	T5, Whisper

Number of Transformer layers

Model	Architecture	Layers	Parameters
BERT-base (2018)	Encoder-only	12	110M
BERT-large (2018)	Encoder-only	24	340M
T5 (2020)	Encoder-decoder	12 + 12	220M
GPT-3 (2020)	Decoder-only	96	175B
Llama 3 70B (2024)	Decoder-only	80	70B

Note: Besides the number of layers, parameter count also depends on hidden size, feed-forward size, vocabulary size, etc.

Recent Large Language Models

Model	Architecture	# layers
Gemma 3 12B (2025)	decoder-only	48
Qwen2.5 72B (2024)	decoder-only	80
Qwen3 8B (2025)	decoder-only	36
SmolLM3 3B (2025)	decoder-only	36

Unfortunately, for closed LLMs, we don't know the exact architecture 😕

Example: Llama 3 8B

	8B	70B	405B
Layers	32	80	126
Model Dimension	4,096	8192	16,384
FFN Dimension	14,336	28,672	53,248
Attention Heads	32	64	128
Key/Value Heads	8	8	8
Peak Learning Rate	3×10^{-4}	1.5×10^{-4}	8×10^{-5}
Activation Function	SwiGLU		
Vocabulary Size	128,000		
Positional Embeddings	RoPE ($\theta = 500,000$)		

Table 3 Overview of the key hyperparameters of Llama 3. We display settings for 8B, 70B, and 405B language models.

Figure: Figure from https://ai.meta.com/research/publications/the-llama-3-herd-of-models

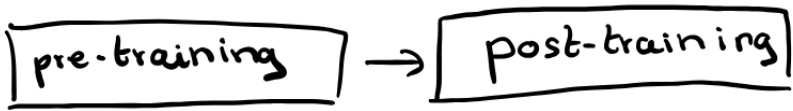
Agenda

Goal: Introduce you to the core components of Large Language Models

Roadmap:

- Historical context
- Core components of Transformer models
- Transformer models
- Training LLMs: pretraining and instruction tuning

How are LLMs trained?



- Pre-training
  - start with a randomly initialized model
  - lots of (web) data, self-supervised
- Post-training
  - start with the pre-trained model
  - different types of post-training: instruction tuning, preference alignment
  - smaller, more curated datasets

Pre-training

Pre-training: The task

You know what I am going to …

Pre-training: The task

You know what I am going to …

say **tell** **do** **green** **to**

Pre-training: The task

You know what I am going to …

say **tell** **do** **green** **to**

Pre-training data can also include factual and mathematical knowledge, usually with more skewed next-token probabilities.

*5+5=**10*** *The capital of France is **Paris***

Pre-training: The task

The task! Next word prediction

Needed: A text corpus.

Steps:

1. At each time step *t*, the model needs to predict the next word.
2. Compare the predicted word with the correct word and update the model.

Self-supervised learning: the labels are automatically generated from the input data.

Cross-entropy (CE) loss

- $p(w)$: true distribution, defined as

$$p(w) = \begin{cases} 1 & \text{if } w = w_{true} \\ 0 & \text{otherwise} \end{cases}$$

- $\hat{p}(w)$: model's predicted probability for word w

Then calculate the cross-entropy (CE) loss as follows:

$$\text{CE} = -\sum_{w \in V} p(w) \log(\hat{p}(w))$$

Cross-entropy (CE) loss

- $p(w)$: true distribution, defined as

$$p(w) = \begin{cases} 1 & \text{if } w = w_{true} \\ 0 & \text{otherwise} \end{cases}$$

- $\hat{p}(w)$: model's predicted probability for word w

Then calculate the cross-entropy (CE) loss as follows:

$$\text{CE} = -\sum_{w \in V} p(w) \log(\hat{p}(w))$$

Which simplifies to:

$$\text{CE} = -\log(\hat{p}(w_{\text{true}}))$$

Cross-entropy (CE) loss

$$\text{CE} = -\log(\hat{p}(w_{\text{true}}))$$

Suppose the correct word (w_{true}) is *cat*, and the model assigns this word a 0.6 probability.

$$\text{CE} = -\log_2(0.6) = 0.73$$

Versus assigning it a 0.05 probability:

$$\text{CE} = -\log_2(0.05) = 4.32$$

Pre-training data

What data?

- The Web!
- Books
- Etc. Etc.
- Increasingly: *Synthetic data*

Rarely used as is: data deduplication, filtering of noisy texts (e.g., boilerplate texts), offensive texts. Some of these steps can be very subjective and introduce biases!

GPT3's training data

Dataset	Quantity (tokens)	Weight in training mix	Epochs elapsed when training for 300B tokens
Common Crawl (filtered)	410 billion	60%	0.44
WebText2	19 billion	22%	2.9
Books1	12 billion	8%	1.9
Books2	55 billion	8%	0.43
Wikipedia	3 billion	3%	3.4

Table 2.2: Datasets used to train GPT-3. "Weight in training mix" refers to the fraction of examples during training that are drawn from a given dataset, which we intentionally do not make proportional to the size of the dataset. As a result, when we train for 300 billion tokens, some datasets are seen up to 3.4 times during training while other datasets are seen less than once.

Figure: https://arxiv.org/pdf/2005.14165

Pre-training data: Common Crawl

- One of the main sources for training language models: **web crawl data**
- Over 250 billion pages spanning 18 years. 3–5 billion new pages added each month.
- Many different subsets have been derived from Common Crawl. One example: The Colossal Clean Crawled Corpus (C4) which was used to train T5.
- https://commoncrawl.org/

Pre-training data: Take a look yourself!

- Go to https://wimbd.apps.allenai.org/ *Internet Domain Explorer*.
- Go to https://huggingface.co/datasets/allenai/c4/viewer/en to see texts from the C4 corpus.

Pre-training data: Take a look yourself!

- Go to https://wimbd.apps.allenai.org/ *Internet Domain Explorer*.
- Go to https://huggingface.co/datasets/allenai/c4/viewer/en to see texts from the C4 corpus.

Would you be comfortable if your own website was included?
What about your social media posts?

Data mixtures

Pre-training data is now usually a *mixture* of different types of data (e.g., web text, code, books, synthetic data).

- Mixtures are tuned empirically (small runs, then scale up)
- Code and math data improve *reasoning*, even on non-coding tasks

Data mixtures

Pre-training data is now usually a *mixture* of different types of data (e.g., web text, code, books, synthetic data).

- Mixtures are tuned empirically (small runs, then scale up)
- Code and math data improve *reasoning*, even on non-coding tasks

Mid-training: between pre- and post training

- Towards the end of training, shift from raw web data to smaller, *higher quality* data (e.g., text books, curated web data, sometimes also instruction data). *Save the best data for last.*
- Smaller learning rates.

In-context learning

Large-pretrained models (e.g. GPT3) turned out to be capable of *in-context* learning. Condition the model on a natural language instruction and performs the task by predicting what comes next.

Language Models are Few-Shot Learners, Brown et al. 2020 [url]

Few-shot prompting

Prompt:

What is the sentiment of the following text?
Answer with: positive or negative.

Examples:

Text: “One of the best movies I’ve ever seen”
Sentiment: positive

Text: “Can’t believe I wasted money on this”
Sentiment: negative

Text: “This movie was awful. I was so bored and left after one hour.”
Sentiment:

Why is pre-training alone not sufficient?

Hint: Models are pre-trained based on word prediction tasks

Translate to Dutch:

This movie was great!

How would a model based on just pre-training respond?

Why is pre-training alone not sufficient?

Hint: Models are pre-trained based on word prediction tasks

Translate to Dutch:

This movie was great!

How would a model based on just pre-training respond?

It **continues** the text, for example:

- "*Deze film was geweldig!*" (web data also contains some instruction texts!)
- ...followed by more translation exercises
- "*I liked it a lot.*" (more review-like text)

Why is pre-training alone not sufficient?

Hint: Models are pre-trained based on word prediction tasks

Translate to Dutch:

This movie was great!

How would a model based on just pre-training respond?

- Not good yet at responding to instructions 😕
- Doesn't "understand" formatting, stopping etc. yet 😕
- Models can generate harmful output! 😕

Why is pre-training alone not sufficient?

Hint: Models are pre-trained based on word prediction tasks

Translate to Dutch:

This movie was great!

How would a model based on just pre-training respond?

next word prediction

vs.

being able to follow instructions,
be helpful, do not cause harm

Instruction Tuning

Instruction tuning

Usually implemented as **supervised fine tuning** (SFT)
Default practice for final models. Often (but not always) indicated by names like `-Instruct` or `-Chat`.

Goal: Make LLMs better at following instructions.

Setup: train on many tasks, test on *seen* and *unseen* tasks

Finetuned Language Models Are Zero-Shot Learners, Wei et al. 2022 [url]
Multitask Prompted Training Enables Zero-Shot Task Generalization, Sanh et al. 2022 [url]

Instruction tuning

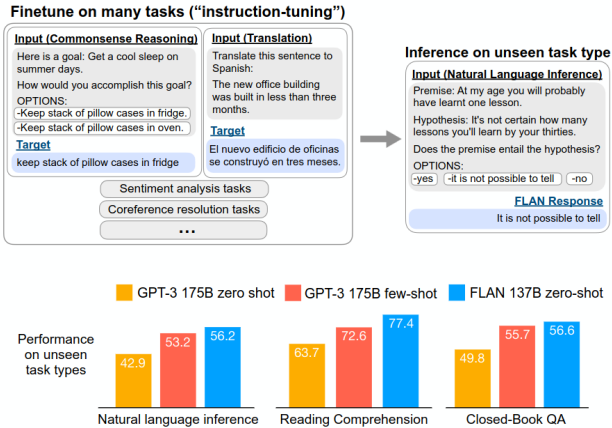


Figure: Figure from Wei et al. 2022

Instruction tuning data

Further train the model on instruction data!

`(instruction/prompt, response)`

Instruction

Given a set of numbers, find the maximum value.
Set: 10, 3, 25, 6, 16

Response

The maximum value is 25.

Instruction tuning data

Further train the model on instruction data!

`(instruction/prompt, response)`

Instruction

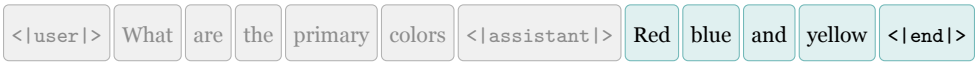
What makes “your” different from “you’re”?

Response

“Your” is a possessive pronoun, while “you’re” is a contraction for “you are.” For example: *your name*, *your keys*; *you’re welcome*, *you’re invited*.

Just like before: next word prediction

Loss on the **completion only**



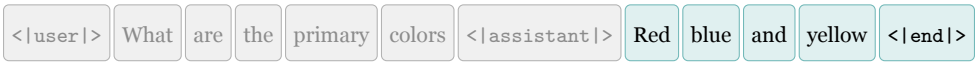
contributes to the loss

masked out

Like before, we calculate cross entropy loss.

Just like before: next word prediction

Loss on the **completion only**



Loss on the **full sequence**

contributes to the loss

masked out

Like before, we calculate cross entropy loss.

What do models learn from instruction tuning?

Knowledge and capabilities come from **pretraining**; post-training (incl. instruction tuning) **surfaces** this knowledge and teaches the **format/style**.

Debated/Still not fully understood.

LIMA: Less Is More for Alignment, Zhou et al. NeurIPS 2023 [url]
Operationalising the Superficial Alignment Hypothesis via Task Complexity, Vergara-Browne et al. ICML 2026 [url]
Do Models Really Learn to Follow Instructions? An Empirical Study of Instruction Tuning, Kung and Peng, ACL 2023 [url]

Datasets

- **Human-generated**
  - Freshly created (e.g., crowd workers, volunteers, etc.)
  - Repurposing existing datasets (e.g., classification datasets) by converting them into an instruction tuning format
- **Synthethic data via distillation**
  - Prompt + responses generated by a larger, stronger model to “teach” a smaller model.

Note: instruction tuning datasets are much smaller than pre-training datasets.

Instruction Tuning for Large Language Models: A Survey, Zhang et al., ACM Comput. Surv 2026 [url]

Datasets

Take a look yourself!
Aya dataset:
https://huggingface.co/datasets/CohereLabs/aya_dataset/viewer

Tulu 3 SFT Mixture: https://huggingface.co/datasets/allenai/tulu-3-sft-mixture/viewer

Instruction

Write an email declining a meeting invitation. Be polite.

Response A

Thank you for the invitation, but unfortunately I won't be able to make it.

Response B

Thank you for the invitation. Unfortunately, I won't be able to attend due to a scheduling conflict. I appreciate the invite and hope we can meet another time.

Which response would you prefer? A or B?

Limitations of instruction tuning

- Data are prompts and “correct responses”. However, sometimes there are multiple “valid” responses but one is better than the other.
- Standard instruction tuning trains on good responses, not bad ones. So the model does not explicitly learn to distinguish between them. It *doesn’t explicitly learn what not to do*.
- Difficult to optimize for *vague/more subjective* goals like helpfulness.

Final words

The field has moved from n-gram models to large neural networks that can learn rich representations.

But we're not there yet! Stay tuned, e.g., learn about **post-training**, **evaluation** and many other topics later at the summer school.

Resources

Resources

- Attention in transformers, step-by-step by 3Blue1Brown (YouTube)
- Speech and Language Processing book by Jurafsky and Martin (free)
- The Illustrated Transformer by Jay Alammar