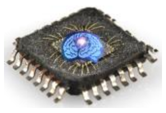
EECE-6822 Machine Learning



Intro to LLMs

Cris Ababei
Dept. of Electrical and Computer Engineering
MARQUETTE UNIVERSITY
BE THE DIFFERENCE.

PART 1
LLMs

Natural Language Processing (NLP) and/vs. Large Language Models (LLMs)

3

3

Old → New models

- Rule-based NLP
- N-grams & classical LM
- Word embeddings (Word2Vec, GloVe)
- RNN → LSTM → GRU
- Attention and Transformer revolution (2017)
- Scaling up → GPT, PaLM, LLaMA, Claude, Gemini, etc.

4

4

Natural Language Processing (NLP)

- NLP is a field of AI that focuses on the interaction between computers and humans using natural language.
- It enables computers to process and analyze human language in both text and speech, and includes a wide range of tasks beyond just generation.
- Key capabilities include understanding, interpreting, and manipulating language, which allows for applications like language translation, sentiment analysis, and chatbots.
- The "generation" aspect is called Natural Language Generation (NLG), and it is one of the many applications of NLP that allows computers to produce human-like text or speech.

Natural Language Processing (NLP)

- NLP relies on various **processes** to enable computers to produce human language:
  - Parsing
  - Semantic Analysis
  - Speech Recognition
  - Natural Language Generation
  - Sentiment Analysis - Often used in monitoring social media and managing brand reputation. It evaluates the emotional tone of texts and analyzes customer feedback and market trends.
  - Machine Translation
  - Named Entity Recognition
  - Text Classification and Categorization

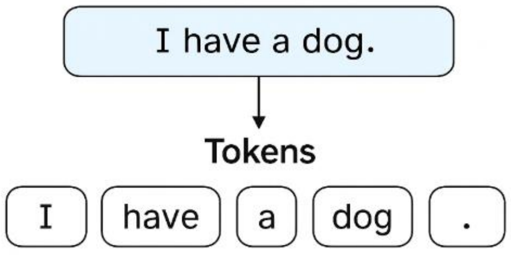
NLP concepts needed to understand LLMs

- Tokens
- Embeddings
- Language modeling
- Sequence modeling

Tokenization

Tokenization is the process of converting text into a sequence of tokens

- Tokens are individual units, such as words or subwords
- Process handles different vocabulary sizes
- Deals with rare or unseen words



Embeddings

- Each token is mapped to an integer ID using a **vocabulary**

```
["I", "have", "a", "dog", "."]
```

$\Rightarrow$

```
[103, 209, 12, 911, 5]
```

- Look up embeddings from the **embedding matrix**

$$E \in \mathbb{R}^{V \times d}$$

- V = vocabulary size
- d = embedding dimension (e.g., 256, 768, 1024, 4096)

- Assume embedding dimension $d = 4$ (very small for illustration)

Token	ID	Embedding vector
"I"	103	[0.9, 0.1, -0.2, 0.5]
"have"	209	[0.3, 0.8, 0.0, -0.1]
"a"	12	[0.2, 0.1, 0.4, 0.7]
"dog"	911	[0.7, -0.3, 0.9, 0.1]
"."	5	[0.0, 0.0, 0.0, 0.1]

$\Rightarrow$

The sentence embedding sequence becomes:

```
[
  [ 0.9,  0.1, -0.2,  0.5],   # "I"
  [ 0.3,  0.8,  0.0, -0.1],   # "have"
  [ 0.2,  0.1,  0.4,  0.7],   # "a"
  [ 0.7, -0.3,  0.9,  0.1],   # "dog"
  [ 0.0,  0.0,  0.0,  0.1],   # "."
]
```

Each token is now in **continuous space**.

Language Modeling (LM)

A **language model (LM)** assigns probabilities to text.

Language modeling is the task of estimating the probability of a natural language sequence. This is the fundamental task from which **all LLM capabilities emerge**.

Given a sentence with tokens:

$$x_1, x_2, \ldots, x_T$$

a language model estimates:

$$P(x_1, x_2, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_{<t})$$

This "predict the next token" formulation is the reason LLMs can generate coherent text, answer questions, translate, summarize, and reason.

Estimate the probability of a sentence from next-token predictions

Let's denote:

- $x_1 = $ "I"
- $x_2 = $ "have"
- $x_3 = $ "a"
- $x_4 = $ "dog"
- $x_5 = $ "."

LM computes the joint probability:

$$P(\text{"I have a dog."}) = P(x_1) \cdot P(x_2 \mid x_1) \cdot P(x_3 \mid x_1, x_2) \cdot P(x_4 \mid x_1, x_2, x_3) \cdot P(x_5 \mid x_1, x_2, x_3, x_4).$$

Concretely:

```
P("I")
× P("have" | "I")
× P("a" | "I have")
× P("dog" | "I have a")
× P("." | "I have a dog")
```

This factorization is the foundation of modern LLM training

Sequence Modeling

- **Sequence modeling** is the task of learning patterns, dependencies, and structure in ordered data — especially **language**, where word order fundamentally determines meaning.
- Sequence modeling builds **context**, not probabilities.

A sequence model takes an ordered list of inputs:

$$x_1, x_2, \ldots, x_T$$

and learns how each element depends on those before (and sometimes after) it.

In language:

- Word order → meaning
- Context → influences interpretation
- Earlier words change the distribution of later words

Sequence modeling captures:

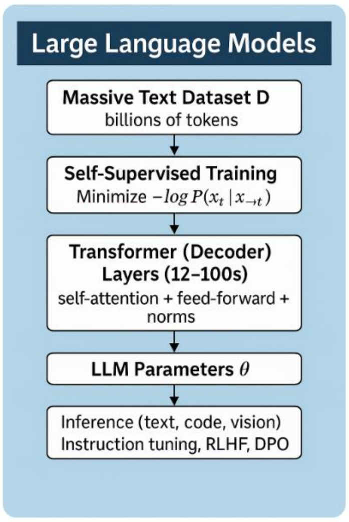
- syntax
- semantics
- long-range dependencies
- grammatical structure
- subject–verb agreement
- phrase boundaries
- semantic roles

Aspect	Sequence Modeling	Language Modeling
Purpose	Learn contextual relationships in an ordered sequence.	Predict the next token and assign probabilities to sequences.
What It Learns	Contextual embeddings h_t for each position.	Conditional distributions $P(x_t \mid x_{<t})$.
Objective	Build representations; no probability requirement.	Maximize next-token likelihood (cross-entropy).
Example ("I have a dog.")	Produces hidden states capturing structure: subject → verb → article → noun.	Computes: P("I have a dog.")
Transformer Use	Both encoder & decoder build contextual states (attention).	Decoder uses **causal masking** to prevent looking ahead.
Outputs Used For	Reasoning, understanding, semantic representation.	Text generation, autocomplete, GPT-style LLM training.
Relationship	**General framework** for processing sequences.	**Specific case** of sequence modeling with probabilistic prediction.

Large Language Models (LLMs)

What Are Large Language Models (LLMs)?

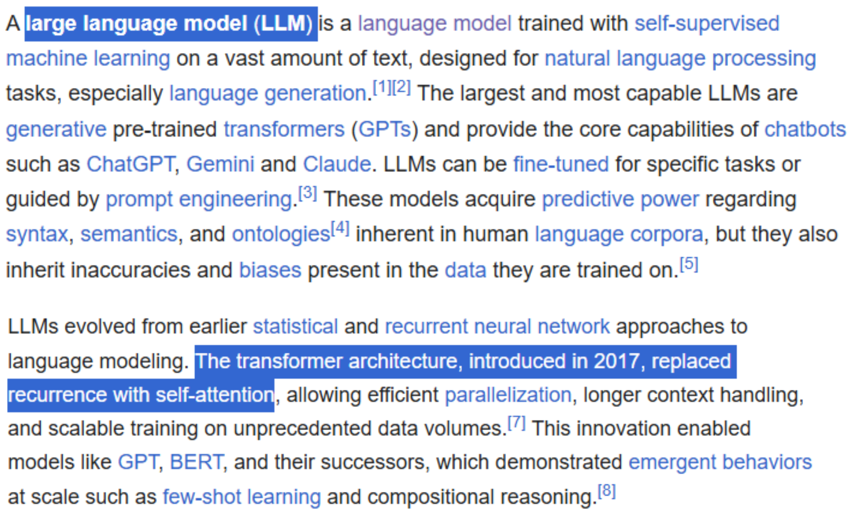
- Large Language Models **(LLMs) are neural networks** trained on massive amounts of text to predict the next **token** in a sequence.
- From this deceptively simple objective emerges powerful capabilities:
  - text generation
  - reasoning
  - summarization
  - translation
  - coding
  - dialogue / agents
  - multimodal understanding (when extended to images, audio, etc.)
- At their core, **LLMs are probabilistic sequence models** based on the **Transformer** architecture.



Large Language Models
Massive Text Dataset D
billions of tokens
Self-Supervised Training
Minimize $-\log P(x_t \mid x_{<t})$
Transformer (Decoder) Layers (12–100s)
self-attention + feed-forward + norms
LLM Parameters θ
Inference (text, code, vision)
Instruction tuning, RLHF, DPO

15

Large Language Models (LLMs)


A large language model (LLM) is a language model trained with self-supervised machine learning on a vast amount of text, designed for natural language processing tasks, especially language generation.[1][2] The largest and most capable LLMs are generative pre-trained transformers (GPTs) and provide the core capabilities of chatbots such as ChatGPT, Gemini and Claude. LLMs can be fine-tuned for specific tasks or guided by prompt engineering.[3] These models acquire predictive power regarding syntax, semantics, and ontologies[4] inherent in human language corpora, but they also inherit inaccuracies and biases present in the data they are trained on.[5]

LLMs evolved from earlier statistical and recurrent neural network approaches to language modeling. The transformer architecture, introduced in 2017, replaced recurrence with self-attention, allowing efficient parallelization, longer context handling, and scalable training on unprecedented data volumes.[7] This innovation enabled models like GPT, BERT, and their successors, which demonstrated emergent behaviors at scale such as few-shot learning and compositional reasoning.[8]

https://en.wikipedia.org/wiki/Large_language_model

16

16

Main Concepts Behind LLMs

1.Tokenization

LLMs operate on **tokens**, not characters or words.

Common tokenizers:

- Byte-Pair Encoding (BPE)
- SentencePiece
- Unigram models
- tiktoken (OpenAI)

Tokenization strikes a balance:

- small vocabulary → efficient
- subwords → handle rare words
- bytes → multilingual support

17

17

Main Concepts Behind LLMs

2.Language Modeling Objective

LLMs are trained to estimate the probability of a sequence of tokens:

$$P(x_1, x_2, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_{<t}; \theta)$$

where

- x_t = token at time t,
- $x_{<t}$ = all previous tokens,
- θ = model parameters.

Training objective (cross-entropy / negative log-likelihood):

$$\min_{\theta} -\sum_{t=1}^{T} \log P_{\theta}(x_t \mid x_{<t})$$

This is the foundation for all capabilities.

18

18

Main Concepts Behind LLMs

3.Transformer Architecture

- LLMs use the **Transformer decoder**, composed of stacked layers:
    a) Multi-Head Self-Attention
    b) Feed-Forward Networks (FFN)
    c) Positional Encodings

(a) Attention helps RNNs with accessing information

To understand the development of an attention mechanism, consider the traditional RNN model for a **seq2seq task** like language translation, which parses the entire input sequence (for instance, one or more sentences) before producing the translation, as shown in *Figure 16.1*:

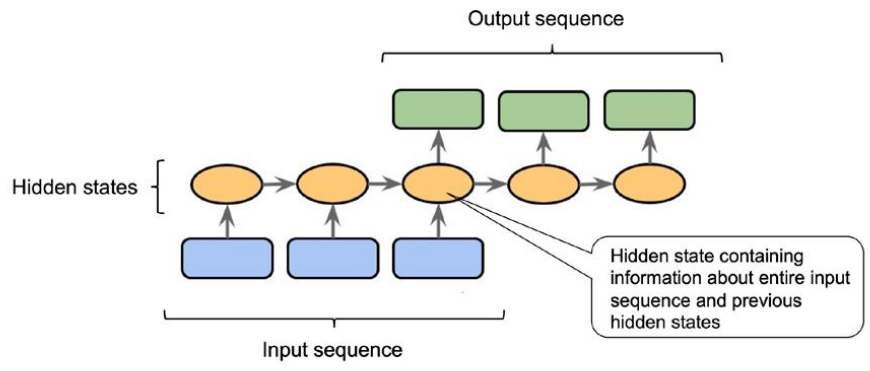

Output sequence
Hidden states
Hidden state containing information about entire input sequence and previous hidden states
Input sequence

Figure 16.1: A traditional RNN encoder-decoder architecture for a seq2seq modeling task

[***B3-Raschka**] Sebastian Raschka, Yuxi Liu, and Vahid Mirjalili, Learning with PyTorch and Scikit-Learn: Develop machine learning and deep learning models with Python, Packt Publishing, 2022.

Why is the RNN parsing the whole input sentence before producing the first output? This is motivated by the fact that translating a sentence word by word would likely result in grammatical errors, as illustrated in *Figure 16.2*:

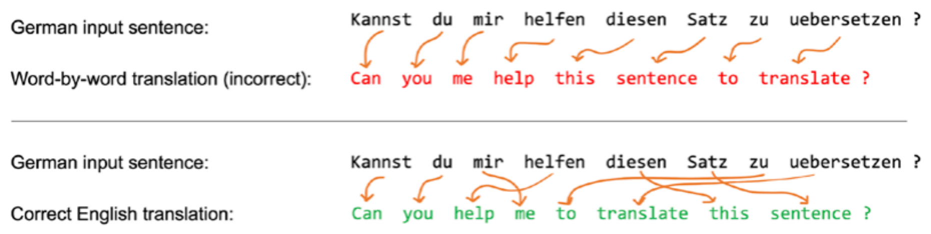

German input sentence:
Kannst du mir helfen diesen Satz zu uebersetzen ?
Word-by-word translation (incorrect):
Can you me help this sentence to translate ?
German input sentence:
Kannst du mir helfen diesen Satz zu uebersetzen ?
Correct English translation:
Can you help me to translate this sentence ?

Figure 16.2: Translating a sentence word by word can lead to grammatical errors

Attention mechanism

- Compressing all the information into one hidden unit may cause loss of information, especially for long sequences.
- It may be beneficial to have access to the whole input sequence at each time step (similar to how humans translate sentences).
- An attention mechanism:
  - Lets the RNN access all input elements at each given time step
  - Assigns different attention weights to each input element; weights designate how important/relevant a given input sequence element is at a given time step.

The original attention mechanism for RNNs

Given an input sequence $x = (x^{(1)}, x^{(2)}, \dots, x^{(T)})$, the attention mechanism assigns a weight to each element $x^{(i)}$ (or, to be more specific, its hidden representation) and helps the model identify which part of the input it should focus on. For example, suppose our input is a sentence, and a word with a

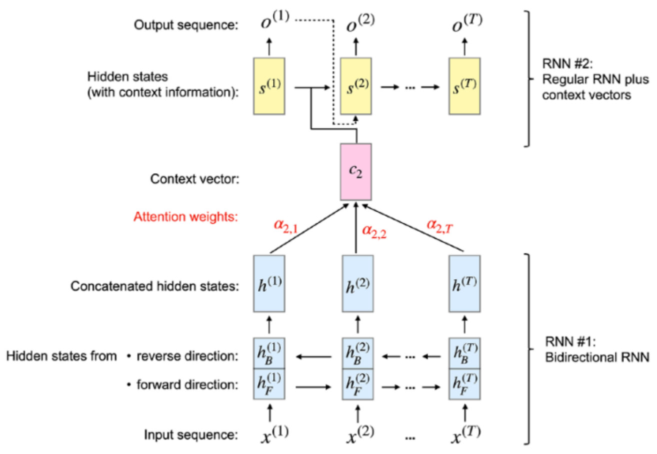


Figure 16.3: RNN with attention mechanism

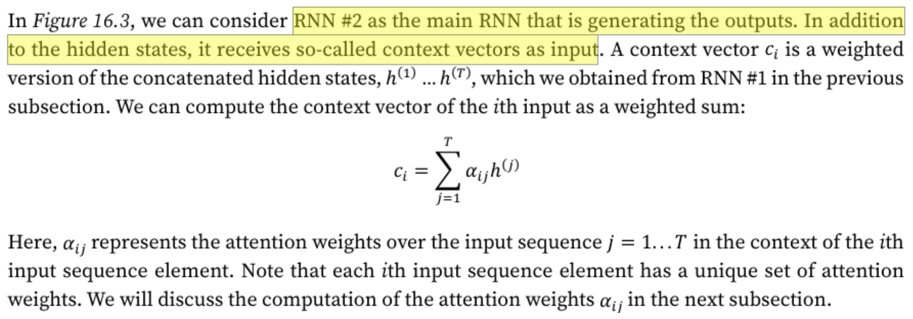
RNN#1: Processing the inputs using a bidirectional RNN

- The first RNN (RNN #1) of the attention-based RNN is a bidirectional RNN that generates context vectors, c_i

The bidirectional RNN #1 processes the input sequence x in the regular forward direction $(1 \dots T)$ as well as backward $(T \dots 1)$. Parsing a sequence in the backward direction has the same effect as reversing the original input sequence—think of reading a sentence in reverse order. The rationale behind this is to capture additional information since current inputs may have a dependence on sequence elements that came either before or after it in a sentence, or both.

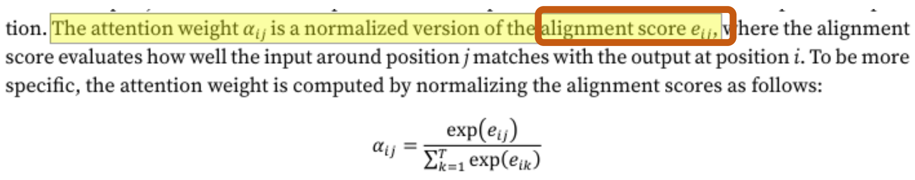
Consequently, from reading the input sequence twice (that is, forward and backward), we have two hidden states for each input sequence element. For instance, for the second input sequence element $x^{(2)}$, we obtain the hidden state $h_F^{(2)}$ from the forward pass and the hidden state $h_B^{(2)}$ from the backward pass. These two hidden states are then concatenated to form the hidden state $h^{(2)}$. For example, if both $h_F^{(2)}$ and $h_B^{(2)}$ are 128-dimensional vectors, the concatenated hidden state $h^{(2)}$ will consist of 256 elements. We can consider this concatenated hidden state as the "annotation" of the source word since it contains the information of the jth word in both directions.

RNN#2: Generating outputs from context vectors


In *Figure 16.3*, we can consider RNN #2 as the main RNN that is generating the outputs. In addition to the hidden states, it receives so-called context vectors as input. A context vector c_i is a weighted version of the concatenated hidden states, $h^{(1)} \dots h^{(T)}$, which we obtained from RNN #1 in the previous subsection. We can compute the context vector of the ith input as a weighted sum:

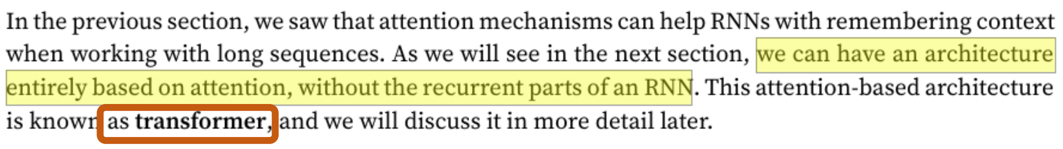
$$c_i = \sum_{j=1}^{T} \alpha_{ij} h^{(j)}$$

Here, α_{ij} represents the attention weights over the input sequence $j = 1 \dots T$ in the context of the ith input sequence element. Note that each ith input sequence element has a unique set of attention weights. We will discuss the computation of the attention weights α_{ij} in the next subsection.


tion. The attention weight α_{ij} is a normalized version of the alignment score e_{ij}, where the alignment score evaluates how well the input around position j matches with the output at position i. To be more specific, the attention weight is computed by normalizing the alignment scores as follows:

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T} \exp(e_{ik})}$$

Introducing a basic self-attention mechanism


In the previous section, we saw that attention mechanisms can help RNNs with remembering context when working with long sequences. As we will see in the next section, we can have an architecture entirely based on attention, without the recurrent parts of an RNN. This attention-based architecture is known as **transformer**, and we will discuss it in more detail later.

What is Self-Attention?

Self-attention is a mechanism that lets each token in a sequence **look at (attend to)** all other tokens (or all *previous* tokens in decoder-only models) and compute **context-aware representations**.

For an input sequence (tokens embedded as vectors):

$$X = [x_1, x_2, \ldots, x_T]$$

self-attention produces new vectors:

$$Z = [z_1, z_2, \ldots, z_T]$$

where each z_t is a *weighted sum* of value vectors across the sequence.

27

27

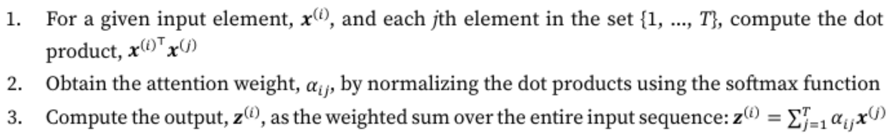
Basic self-attention mechanism

- Goal of self-attention is to model the dependencies of the current input element to all other input elements
- Self-attention mechanisms are composed of three stages
  1. Derive importance weights based on the similarity between the current element and all other elements in the sequence
  2. Normalize the weights, which usually involves the use of the already familiar softmax function
  3. Use these weights in combination with the corresponding sequence elements to compute the attention value

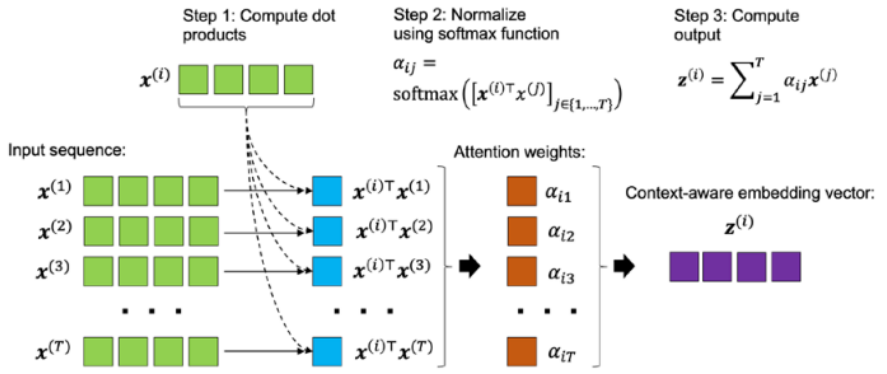
28

28

Recap and summarize the three main steps behind the basic self-attention operation:



1. For a given input element, $x^{(i)}$, and each jth element in the set {1, ..., T}, compute the dot product, $x^{(i)\top}x^{(j)}$
2. Obtain the attention weight, α_{ij}, by normalizing the dot products using the softmax function
3. Compute the output, $z^{(i)}$, as the weighted sum over the entire input sequence: $z^{(i)} = \sum_{j=1}^{T}\alpha_{ij}x^{(j)}$

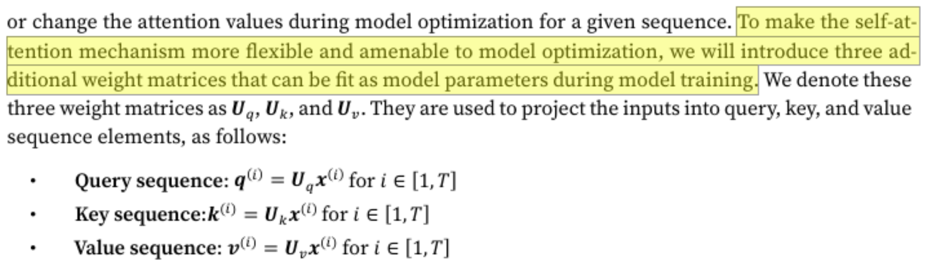


29

29

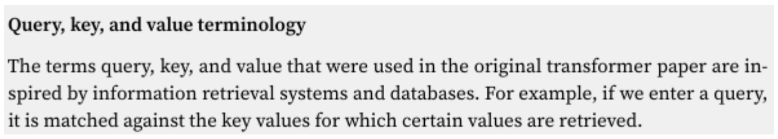
Parameterizing the self-attention mechanism: scaled dot-product attention

- More advanced self-attention mechanism called scaled dot-product attention that is used in the transformer architecture



or change the attention values during model optimization for a given sequence. To make the self-attention mechanism more flexible and amenable to model optimization, we will introduce three additional weight matrices that can be fit as model parameters during model training. We denote these three weight matrices as $\boldsymbol{U}_q$, $\boldsymbol{U}_k$, and $\boldsymbol{U}_v$. They are used to project the inputs into query, key, and value sequence elements, as follows:

- **Query sequence:** $\boldsymbol{q}^{(i)} = \boldsymbol{U}_q\boldsymbol{x}^{(i)}$ for $i \in [1, T]$
- **Key sequence:** $\boldsymbol{k}^{(i)} = \boldsymbol{U}_k\boldsymbol{x}^{(i)}$ for $i \in [1, T]$
- **Value sequence:** $\boldsymbol{v}^{(i)} = \boldsymbol{U}_v\boldsymbol{x}^{(i)}$ for $i \in [1, T]$



Query, key, and value terminology

The terms query, key, and value that were used in the original transformer paper are inspired by information retrieval systems and databases. For example, if we enter a query, it is matched against the key values for which certain values are retrieved.

30

30

The Q / K / V Projections

Each token embedding x_t is linearly projected into:

$$q_t = U_Q x_t, \quad k_t = U_K x_t, \quad v_t = U_V x_t$$

Stacked across the sequence:

$$Q = XU_Q, \quad K = XU_K, \quad V = XU_V$$

Where:

- $U_Q, U_K, U_V \in \mathbb{R}^{d \times d_k}$
- $Q, K, V \in \mathbb{R}^{T \times d_k}$

Interpretation:

Matrix	Meaning
U_Q	shapes **queries** (what the token is looking for)
U_K	shapes **keys** (what the token contains)
U_V	shapes **values** (information to aggregate)

Attention Score Computation

Attention scores measure similarity between queries and keys:

$$\text{Scores} = \frac{QK^\top}{\sqrt{d_k}}$$

This yields a $T \times T$ matrix:

- Row i: how much token i attends to all other tokens
- Column j: influence of token j

Softmax → Attention Weights

Apply row-wise softmax:

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

Each row of A sums to 1.

These are the **attention weights**.

- Large weight → strong attention
- Small weight → weak attention

33

Weighted Sum of Value Vectors

Compute the final attention outputs:

$$Z = AV$$

This yields contextualized embeddings.

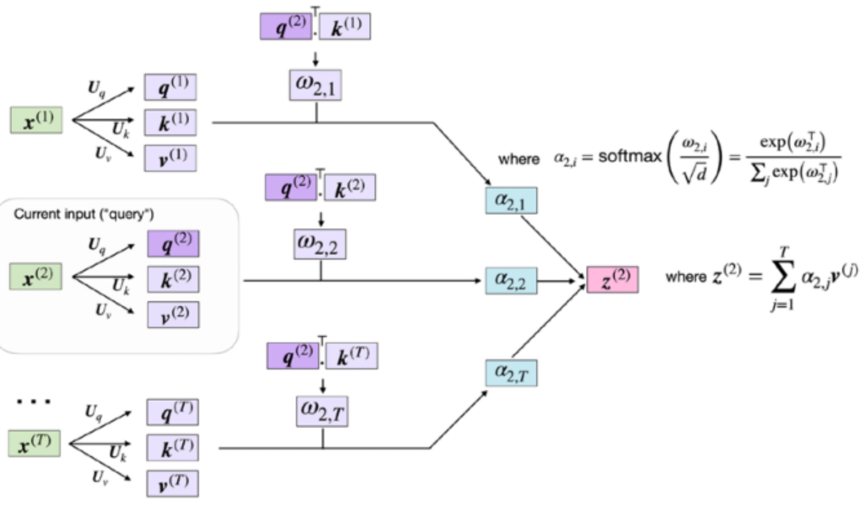
Each z_t becomes:

$$z_t = \sum_{j=1}^{T} A_{t,j} \cdot v_j$$

Meaning token t absorbs information from the entire sequence.

34

Figure 16.5 illustrates how these individual components are used to compute the context-aware embedding vector corresponding to the second input element:



$q^{(2)} \cdot^{\top} k^{(1)}$
$\omega_{2,1}$
U_q
U_k
U_v
$x^{(1)}$
$q^{(1)}$
$k^{(1)}$
$v^{(1)}$
where $\alpha_{2,i} = \text{softmax}\left(\frac{\omega_{2,i}}{\sqrt{d}}\right) = \frac{\exp(\omega_{2,i}^{\top})}{\sum_j \exp(\omega_{2,j}^{\top})}$
$q^{(2)} \cdot^{\top} k^{(2)}$
$\alpha_{2,1}$
Current input ("query")
$\omega_{2,2}$
$x^{(2)}$
$q^{(2)}$
$k^{(2)}$
$v^{(2)}$
$\alpha_{2,2}$
$z^{(2)}$
where $z^{(2)} = \sum_{j=1}^{T} \alpha_{2,j} v^{(j)}$
$q^{(2)} \cdot^{\top} k^{(T)}$
$\alpha_{2,T}$
$\omega_{2,T}$
...
$x^{(T)}$
$q^{(T)}$
$k^{(T)}$
$v^{(T)}$

35

35

Why Self-Attention?

- It replaces recurrence (RNNs) and convolutions (CNNs) by allowing:
  - **Long-range dependencies** (subject ↔ verb even far away)
  - **Parallelization** over all tokens
  - **Flexible context modeling**
  - **Better scaling with depth and width**
- In LLMs, self-attention is the primary mechanism that gives tokens awareness of each other.

36

36

What Self-Attention Learns

- Different heads learn:
  - grammatical structure
  - semantic roles
  - long-range dependencies
  - coreference (e.g., “the dog” $\leftrightarrow$ “it”)
  - reasoning dependencies
  - token alignment (translation, summarization, coding)
- These abilities emerge through the Q/K/V interactions defined by U_Q, U_K, U_V

37

37

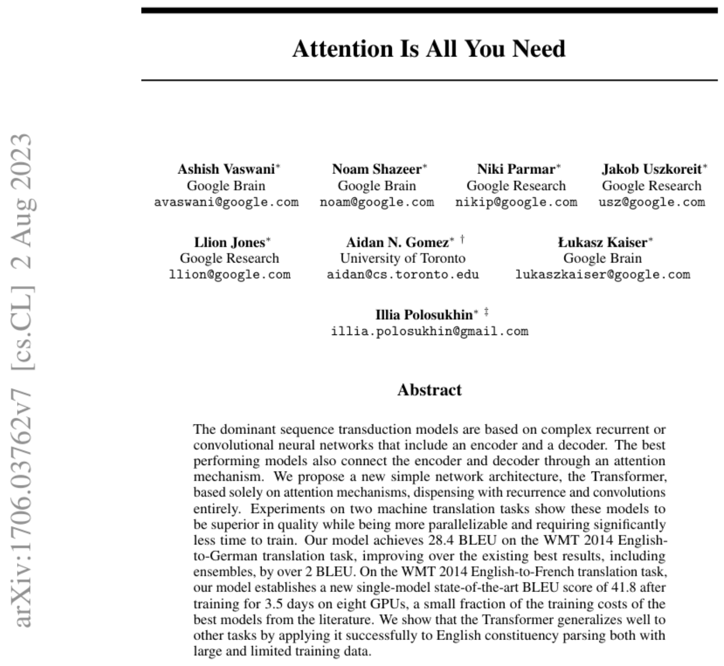
Attention is all we need: introducing the **original transformer architecture**

- A few years after experimenting with attention mechanisms for RNNs, researchers found that an attention-based language model was even more powerful when the recurrent layers were deleted
- This led to the development of the transformer architecture
- Thanks to the **self-attention mechanism**, a transformer model can capture long-range dependencies among the elements in an input sequence—in an NLP context; for example, this helps the model better “understand” the meaning of an input sentence.
- Although **this transformer architecture was originally designed for language translation**, it can be generalized to other tasks such as English constituency parsing, text generation, and text classification.

38

38

Transformer - Original Paper



Attention Is All You Need

Ashish Vaswani[*] Google Brain avaswani@google.com

Noam Shazeer[*] Google Brain noam@google.com

Niki Parmar[*] Google Research nikip@google.com

Jakob Uszkoreit[*] Google Research usz@google.com

Llion Jones[*] Google Research llion@google.com

Aidan N. Gomez[*] [†] University of Toronto aidan@cs.toronto.edu

Łukasz Kaiser[*] Google Brain lukaszkaiser@google.com

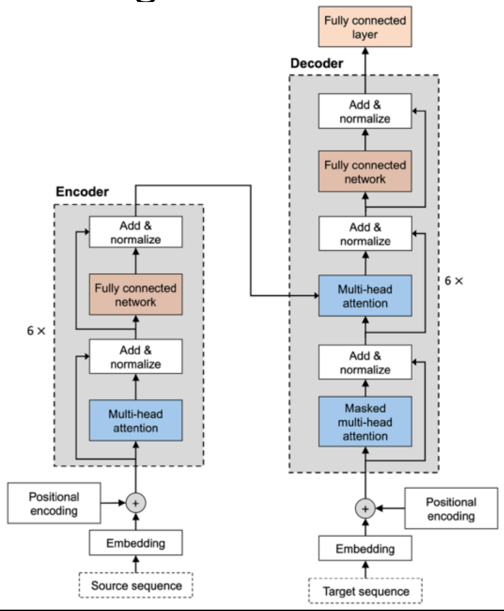
Illia Polosukhin[*] [‡] illia.polosukhin@gmail.com

Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.8 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature. We show that the Transformer generalizes well to other tasks by applying it successfully to English constituency parsing both with large and limited training data.

39

The original transformer architecture



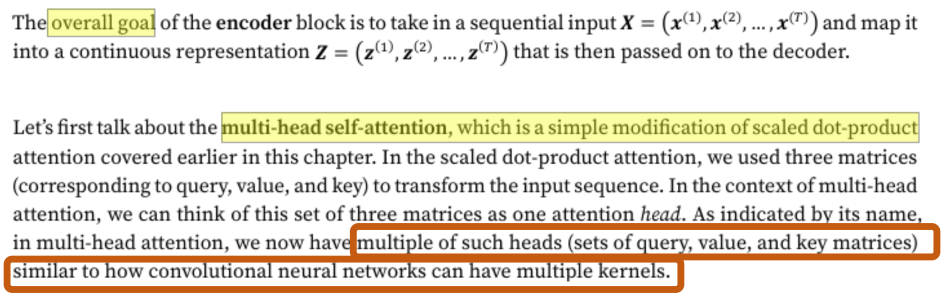
Two main blocks:

1. **The encoder** - receives the original sequential input and encodes the embeddings using a multi-head self-attention module
2. **The decoder** - takes in the processed input and outputs the resulting sequence (for instance, the translated sentence) using a masked form of self-attention

40

40

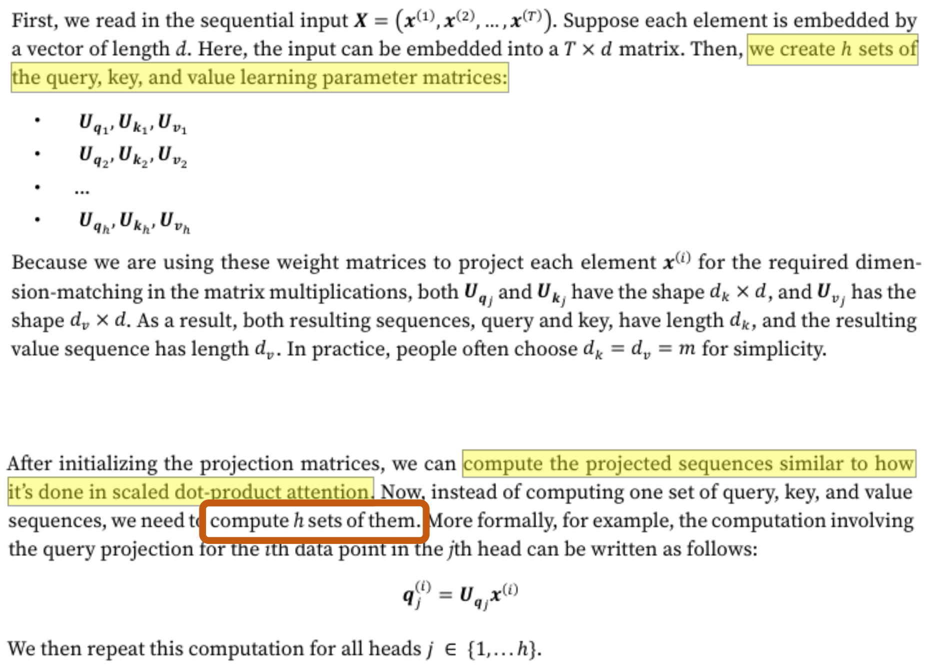
Encoding context embeddings via multi-head attention



The overall goal of the **encoder** block is to take in a sequential input $\boldsymbol{X} = \left(\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(T)}\right)$ and map it into a continuous representation $\boldsymbol{Z} = \left(\boldsymbol{z}^{(1)}, \boldsymbol{z}^{(2)}, \ldots, \boldsymbol{z}^{(T)}\right)$ that is then passed on to the decoder.

Let's first talk about the **multi-head self-attention**, which is a simple modification of scaled dot-product attention covered earlier in this chapter. In the scaled dot-product attention, we used three matrices (corresponding to query, value, and key) to transform the input sequence. In the context of multi-head attention, we can think of this set of three matrices as one attention *head*. As indicated by its name, in multi-head attention, we now have multiple of such heads (sets of query, value, and key matrices) similar to how convolutional neural networks can have multiple kernels.

See S.Raschka's book for a nice explanation of the concept of multi-head self-attention with h heads.



First, we read in the sequential input $\boldsymbol{X} = \left(\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(T)}\right)$. Suppose each element is embedded by a vector of length d. Here, the input can be embedded into a $T \times d$ matrix. Then, we create h sets of the query, key, and value learning parameter matrices:

- $\boldsymbol{U}_{q_1}, \boldsymbol{U}_{k_1}, \boldsymbol{U}_{v_1}$
- $\boldsymbol{U}_{q_2}, \boldsymbol{U}_{k_2}, \boldsymbol{U}_{v_2}$
- ...
- $\boldsymbol{U}_{q_h}, \boldsymbol{U}_{k_h}, \boldsymbol{U}_{v_h}$

Because we are using these weight matrices to project each element $\boldsymbol{x}^{(i)}$ for the required dimension-matching in the matrix multiplications, both $\boldsymbol{U}_{q_j}$ and $\boldsymbol{U}_{k_j}$ have the shape $d_k \times d$, and $\boldsymbol{U}_{v_j}$ has the shape $d_v \times d$. As a result, both resulting sequences, query and key, have length d_k, and the resulting value sequence has length d_v. In practice, people often choose $d_k = d_v = m$ for simplicity.

After initializing the projection matrices, we can compute the projected sequences similar to how it's done in scaled dot-product attention. Now, instead of computing one set of query, key, and value sequences, we need to compute h sets of them. More formally, for example, the computation involving the query projection for the ith data point in the jth head can be written as follows:

$$\boldsymbol{q}_j^{(i)} = \boldsymbol{U}_{q_j} \boldsymbol{x}^{(i)}$$

We then repeat this computation for all heads $j \in \{1, \ldots h\}$.

We follow the steps of the single head attention calculation to calculate the context vectors as described in the *Parameterizing the self-attention mechanism: scaled dot-product attention* section. We will skip the intermediate steps for brevity and assume that we have computed the context vectors for the second input element as the query and the eight different attention heads, which we represent as

Then, we concatenate these vectors into one long vector of length $d_v \times h$ and use a linear projection (via a fully connected layer) to map it back to a vector of length d_v. This process is illustrated in *Figure 16.7*:

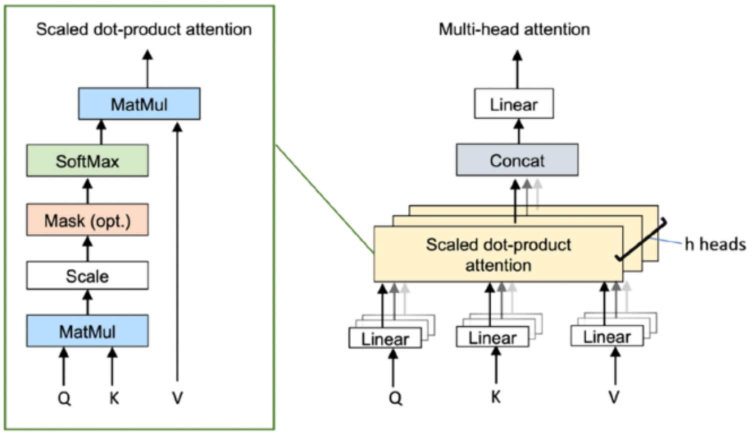


Figure 16.7: Concatenating the scaled dot-product attention vectors into one vector and passing it through a linear projection

43

Learning a language model: decoder and **masked** multi-head attention

Similar to the encoder, the **decoder** also contains several repeated layers. Besides the two sublayers that we have already introduced in the previous encoder section (the multi-head self-attention layer and fully connected layer), each repeated layer also contains a masked multi-head attention sublayer.

Masked attention is a variation of the original attention mechanism, where masked attention only passes a limited input sequence into the model by "masking" out a certain number of words. For example, if we are building a language translation model with a labeled dataset, at sequence position i during the training procedure, we only feed in the correct output words from positions 1,...,i-1. All other words (for instance, those that come after the current position) are hidden from the model to prevent the model from "cheating." This is also consistent with the nature of text generation: although the true translated words are known during training, we know nothing about the ground truth in practice. Thus, we can only feed the model the solutions to what it has already generated, at position i.

44

Figure 16.8 illustrates how the layers are arranged in the decoder block:

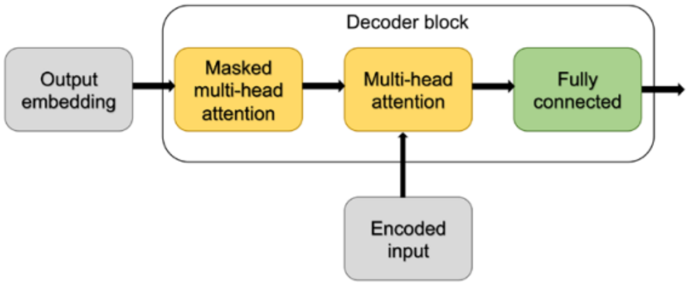


Figure 16.8: Layer arrangement in the decoder part

First, the previous output words (output embeddings) are passed into the masked multi-head attention layer. Then, the second layer receives both the encoded inputs from the encoder block and the output of the masked multi-head attention layer into a multi-head attention layer. Finally, we pass the multi-head attention outputs into a fully connected layer that generates the overall model output: a probability vector corresponding to the output words.

See S.Raschka's book for more details.

45

45

Building large-scale language models by leveraging unlabeled data

- Given that large amounts of text (books, websites, and social media posts) are generated every day, an interesting question is **how we can use such unlabeled data for improving the model training**
- Self-supervised learning: we can generate "labels" from supervised learning from plain text itself
- Self-supervised learning - also referred to as unsupervised pretraining - is essential for the success of modern transformer-based models.
- The "unsupervised" in unsupervised pre-training supposedly refers to the fact that we use unlabeled data; however, since we use the structure of the data to generate labels (for example, the next-word prediction task), it is still a supervised learning process.

46

46

To elaborate a bit further on how unsupervised pre-training and next-word prediction works, if we have a sentence containing *n* words, the pre-training procedure can be decomposed into the following three steps:

1. At time *step 1*, feed in the ground-truth words 1, ..., *i*-1.
2. Ask the model to predict the word at position *i* and compare it with the ground-truth word *i*.
3. Update the model and time step, *i*:= *i*+1. Go back to step 1 and repeat until all words are processed.

We should note that in the next iteration, we always feed the model the ground-truth (correct) words instead of what the model has generated in the previous round.

A complete training procedure of a transformer-based model consists of two parts: (1) pre-training on a large, unlabeled dataset and (2) training (that is, fine-tuning) the model for specific downstream tasks using a labeled dataset. In the first step, the pre-trained model is not designed for any specific task but rather trained as a "general" language model. Afterward, via the second step, it can be generalized to any customized task via regular supervised learning on a labeled dataset.

With the representations that can be obtained from the pre-trained model, there are mainly two strategies for transferring and adopting a model to a specific task: (1) a **feature-based approach** and (2) a **fine-tuning approach**. (Here, we can think of these representations as the hidden layer activations of the last layers of a model.)

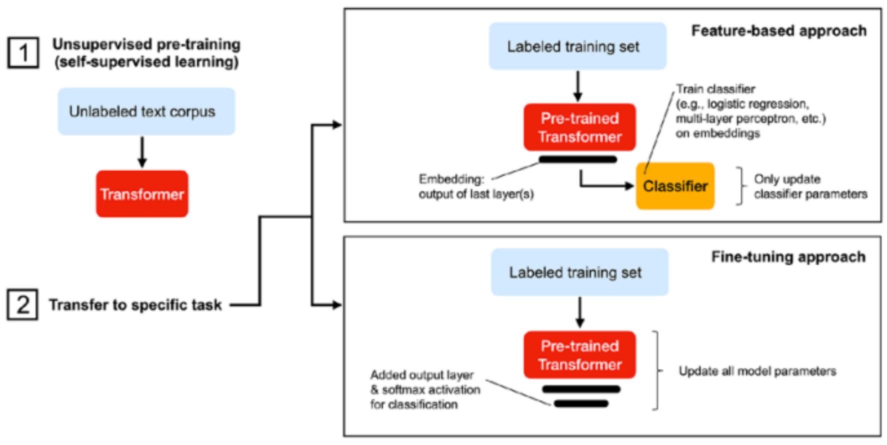

1
Unsupervised pre-training (self-supervised learning)
Unlabeled text corpus
Transformer
2
Transfer to specific task
Feature-based approach
Labeled training set
Pre-trained Transformer
Train classifier (e.g., logistic regression, multi-layer perceptron, etc.) on embeddings
Embedding: output of last layer(s)
Classifier
Only update classifier parameters
Fine-tuning approach
Labeled training set
Pre-trained Transformer
Update all model parameters
Added output layer & softmax activation for classification

Figure 16.10: The two main ways to adopt a pre-trained transformer for downstream tasks

Summary: (a)Multi-Head Self-Attention

Each token attends to all previous tokens:

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

where Q, K, V are learned linear projections of hidden states.

This allows the model to capture:

- long-range dependencies
- contextual relationships
- compositional structure

49

49

(b)Feed-Forward Networks (FFN)

Each position passes independently through a 2-layer MLP:

$$\mathrm{FFN}(x)=W_2\,\sigma(W_1x+b_1)+b_2$$

Where:

- Hidden dimension (e.g., 4× the embedding size)
- Activation is typically GELU or ReLU

FFN provides:

- Nonlinearity
- Dimensional expansion/compression
- Per-token transformation independent of other tokens

50

50

(c)Why Do We Need Positional Encodings?

Self-attention treats input tokens as an **unordered set**.

The attention mechanism:

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

does **not depend on token positions**—only their embeddings.

→ Therefore, without extra information, a Transformer cannot know:

- the order of words
- which token came first
- whether "dog bites man" or "man bites dog"

Solution: Add **positional encodings** to token embeddings.

51

51

What Are Positional Encodings?

For an input sequence of T tokens with embeddings:

$$X = [x_1, x_2, \ldots, x_T]$$

we define positional vectors:

$$\tilde{x}_t = x_t + p_t$$

where:

- p_t is the positional encoding for position t
- $\tilde{x}_t$ is the combined embedding used by the Transformer

Matrix form:

$$\tilde{X} = X + P$$

where $P \in \mathbb{R}^{T \times d}$ contains all positional encodings.

52

52

Types of Positional Encodings

A.**Sinusoidal Positional Encodings (Original Transformer)**

Defined as:

$$p_t^{(2k)} = \sin\left(\frac{t}{10000^{2k/d}}\right)$$

$$p_t^{(2k+1)} = \cos\left(\frac{t}{10000^{2k/d}}\right)$$

Properties:

- Allows extrapolation to longer sequences
- Provides unique signatures for each position
- Encodes relative positions implicitly
  (because of sine/cosine phase relationships)

53

53

Types of Positional Encodings

B.**Learned Positional Embeddings (GPT, BERT)**

Instead of sinusoids:

$$P = \begin{bmatrix} p_1^\top \\ p_2^\top \\ \vdots \\ p_T^\top \end{bmatrix}$$

is a **learnable matrix**.

Advantages:

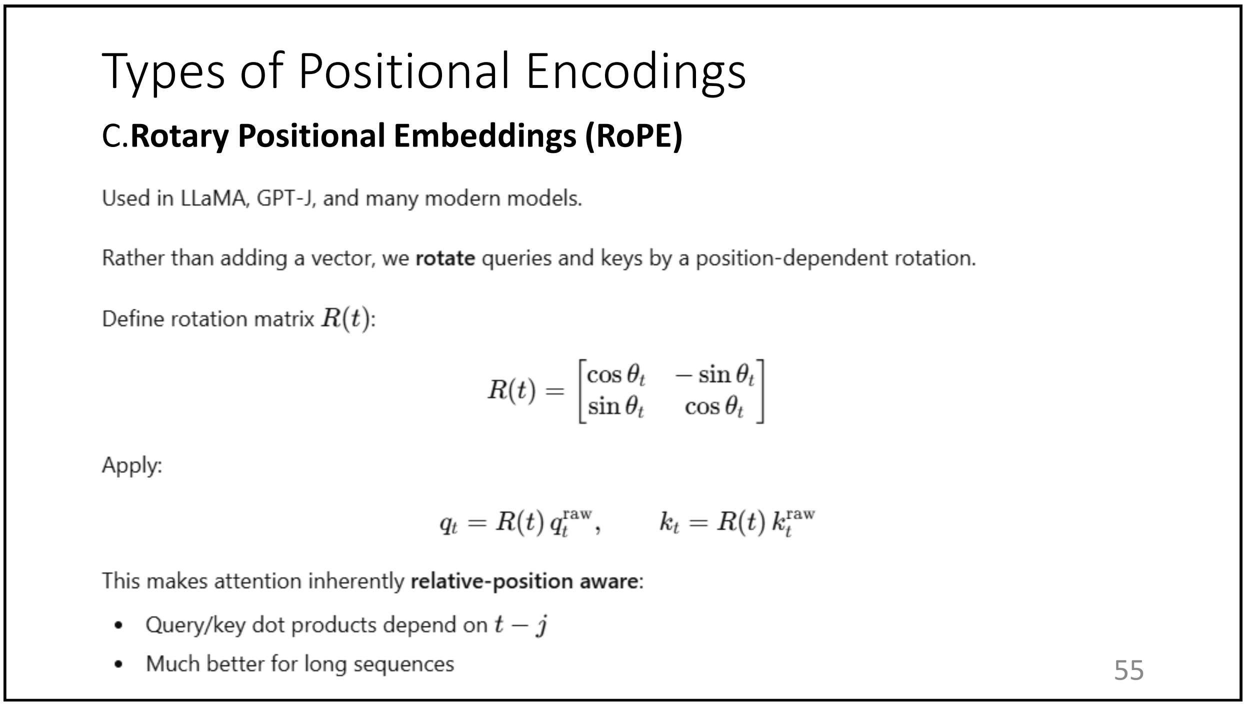
- Can learn task-specific position structure
- Simple and effective
- Used in many large LLMs

Limit:

- Does not extrapolate to sequences longer than training length

54

54



Types of Positional Encodings

C.**Rotary Positional Embeddings (RoPE)**

Used in LLaMA, GPT-J, and many modern models.

Rather than adding a vector, we **rotate** queries and keys by a position-dependent rotation.

Define rotation matrix $R(t)$:

$$R(t) = \begin{bmatrix} \cos\theta_t & -\sin\theta_t \\ \sin\theta_t & \cos\theta_t \end{bmatrix}$$

Apply:

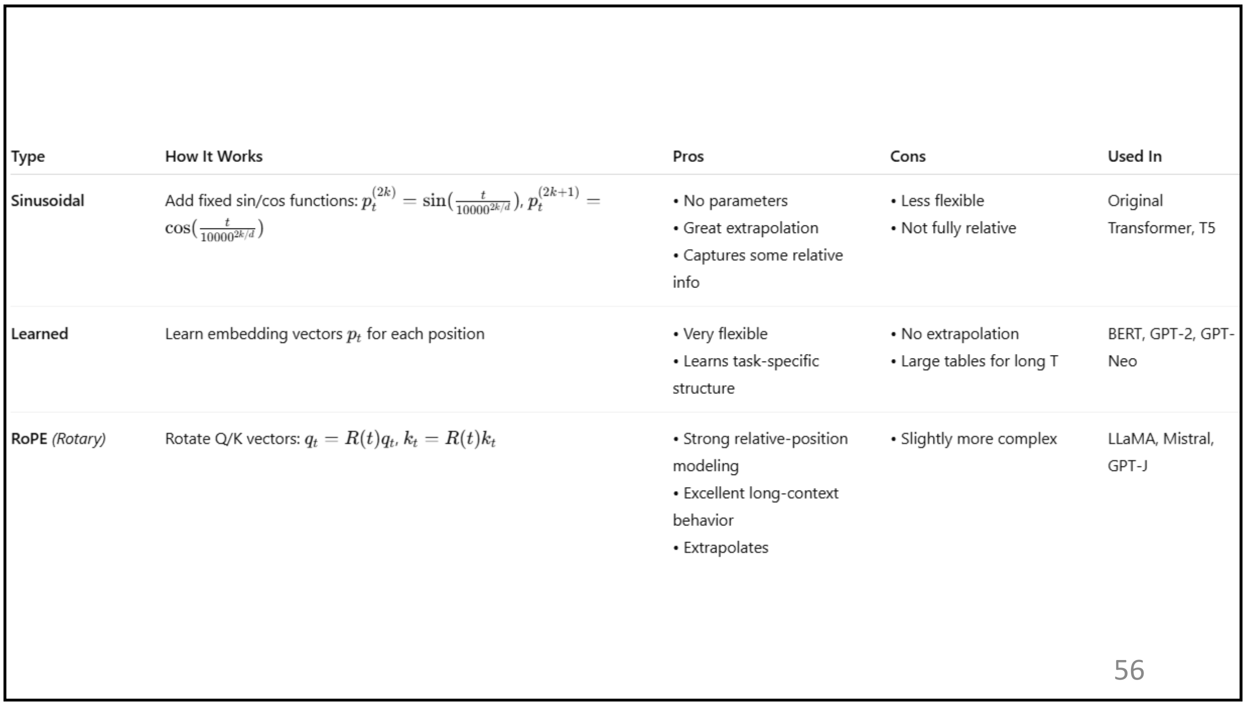
$$q_t = R(t)\, q_t^{\text{raw}}, \qquad k_t = R(t)\, k_t^{\text{raw}}$$

This makes attention inherently **relative-position aware**:

- Query/key dot products depend on $t - j$
- Much better for long sequences

55

55



Type	How It Works	Pros	Cons	Used In
Sinusoidal	Add fixed sin/cos functions: $p_t^{(2k)} = \sin(\frac{t}{10000^{2k/d}})$, $p_t^{(2k+1)} = \cos(\frac{t}{10000^{2k/d}})$	• No parameters • Great extrapolation • Captures some relative info	• Less flexible • Not fully relative	Original Transformer, T5
Learned	Learn embedding vectors p_t for each position	• Very flexible • Learns task-specific structure	• No extrapolation • Large tables for long T	BERT, GPT-2, GPT-Neo
RoPE *(Rotary)*	Rotate Q/K vectors: $q_t = R(t)q_t$, $k_t = R(t)k_t$	• Strong relative-position modeling • Excellent long-context behavior • Extrapolates	• Slightly more complex	LLaMA, Mistral, GPT-J

56

56

Reinforcement Learning (RL) and Large Language Models (LLMs)

57

57

Reinforcement Learning (RL)

- RL is a post-training method for **fine-tuning LLMs** to align them with human preferences, improve reasoning, and generate more helpful responses.
- The most common technique is Reinforcement Learning from Human Feedback (RLHF), where humans provide preference data to train a reward model, which is then used to update the LLM using RL.
- Other methods like Reinforcement Learning from AI Feedback (RLAIF) and Reinforcement Learning from Self-Feedback (RLSF) use AI or the model's own confidence to create a reward signal, reducing the need for human labels.
- **Videos:**
  - Video explains the basics of RL from Human Feedback (RLHF) for large language models: https://www.youtube.com/watch?v=qPN_XZcJf_s&t=203s
  - Lecture on reinforcement learning (RL) fine-tuning of large language models (LLMs): https://www.youtube.com/watch?v=NTSYgbwBVaY&t=2s

58

58

More videos

- Jay Alammar - "Hands-On Large Language Models: Language Understanding and Generation"
  - https://www.youtube.com/watch?v=RVxl9u7rt9w
- Natasha Jacques - What Makes ChatGPT Chat? Modern AI for the layperson
  - https://www.youtube.com/watch?v=KvTGUI4Tznw

59

59

PART 2
Code Time!

60

60

Code Time

- See demonstration and discussion in class.
- See also links in the “Code Examples” for this lecture assignment.

61

61

Conclusion

Takeaways

62

62

Main Takeaways

- LLMs are Transformer-based sequence models trained with next-token prediction.
- They rely on attention, tokenization, and massive data + compute.
- Scaling produces emergent reasoning and generalization abilities.
- After pretraining, instruction tuning and feedback alignment make them useful and safe.
- LLMs operate entirely in-context, making them flexible problem solvers.

63

63

References and Credits

Many of the teaching materials for this course have been adapted from various sources. We are very grateful and thank the following professors, researchers, and practitioners for sharing their teaching materials (in no particular order):

- Yaser S. Abu-Mostafa, Malik Magdon-Ismail and Hsuan-Tien Lin. https://amlbook.com/slides.html
- Ethem Alpaydin. https://www.cmpe.boun.edu.tr/~ethem/i2ml3e/
- Natasha Jaques. https://courses.cs.washington.edu/courses/cse446/25sp/
- Lyle Ungar. https://alliance.seas.upenn.edu/~cis520/dynamic/2022/wiki/index.php?n=Lectures.Lectures
- Aurelien Geron. https://github.com/ageron/handson-ml3
- Sebastian Raschka. https://github.com/rasbt/machine-learning-book
- Trevor Hastie. https://www.statlearning.com/resources-python
- Andrew Ng. https://www.youtube.com/playlist?list=PLoROMvodv4rMiGQp3WXShtMGgzqpfVfbU
- Richard Povineli. https://www.richard.povinelli.org/teaching
- … and many others.

64

64