

Attention and Transformers

Last modified: 2026-10-18

```
# The Python chunks on this page import numpy; reticulate provisions it.
# The Julia chunks run through JuliaCall.
library(reticulate)
py_require("numpy")
```

i Source

Adapted from Logan Sizemore’s Spring 2025 lectures on attention and transformers.

The models in Neural Networks take a fixed-length feature vector and produce an output. Language does not arrive that way: a sentence is a sequence, its length varies, and what a word means depends on the words around it. This chapter builds the mechanism that handles all three — attention — and then assembles it into the transformer block, the building block of modern language models [1].

1 Language models

- A *language model* assigns probabilities to sequences of words
- In practice this is almost always posed as **next-token prediction**: given the tokens so far, give a distribution over what comes next
 - “Mary had a little _____” $\rightarrow$ **lamb**
- A *token* is the unit the model actually predicts. It is usually a subword fragment rather than a whole word, so that a fixed vocabulary can spell any input

i Watch

Sanderson explains how a model predicts the next word, from the start^a [2, Ch. 5].

^a<https://www.youtube.com/watch?v=wjZofJX0v4M>

1.1 N-gram models

- The earliest language models estimate next-token probability from counts
- An n -gram model conditions on the previous $n - 1$ tokens only:

$$P(w_t \mid w_1, \dots, w_{t-1}) \approx P(w_t \mid w_{t-n+1}, \dots, w_{t-1})$$

- For a bigram ($n = 2$) model this is a table of transition probabilities, one row per vocabulary item
- Two problems follow immediately:
 - **The table grows too large.** A vocabulary of V tokens needs V^n entries, almost all of which are never observed

- **The context is too short.** With $n = 2$, generation degenerates — “the river flows to the sea and the river flows to the sea and ...” — because the model cannot see that it has already produced this text
- Enlarging n trades the second problem for the first. Instead, we want a representation that can look arbitrarily far back without needing V^n

i Read

GeeksforGeeks explains n -gram language models^a [3].

^a<https://www.geeksforgeeks.org/nlp/n-gram-language-modelling-with-nltk/>

2 Word embeddings

- Rather than treating a token as an index into a table, represent it as a vector in $\mathbb{R}^d$:

$$\text{"lamp"} \mapsto [0.6 \quad 0.1 \quad 0.8 \quad 0.3 \quad 0.5]^\top$$

- Such a vector is a *word embedding*. The model learns it rather than a person designing it. The geometry it learns carries meaning: tokens used in similar contexts are close to one another
- That geometry makes the analogy arithmetic work: the vector offset from **Paris** to **France** points in roughly the same direction as the offset from **Rome** to **Italy**

2.1 Why a static embedding is not enough

- A fixed lookup gives each word form exactly one vector, so every sense of a homonym has to share it:
 - “the **bat** flew out of the cave” and “the **bat** was made of ash” get the same vector, somewhere between the two senses and correct for neither
- Figurative use has the same problem without the homonymy. In “she is a real **shark**”, the word names an animal but describes a person, and the animal’s vector says nothing about that
- So we want a *contextualized* embedding: a vector for this token **in this sentence**. We build it by moving the static embedding towards the other tokens present

i Watch

- The first lecture of Stanford’s Natural Language Processing with Deep Learning course (CS224N) introduces word embeddings, from 23:33^a [4], Lecture 1.
- Starmer explains word embeddings^b [5].

^a<https://www.youtube.com/watch?v=rmVRLeJRkl4&t=1413s>

^b<https://www.youtube.com/watch?v=viZrOnJclY0>

i Watch

Sanderson explains word embeddings, from 12:27^a [2, Ch. 5].

^a<https://www.youtube.com/watch?v=wjZofJX0v4M&t=747s>

i Read

GeeksforGeeks explains word embeddings^a [6].

^a<https://www.geeksforgeeks.org/nlp/word-embeddings-in-nlp/>

3 Contextualizing by proximity

- The simplest way to build such a vector is to average each token's embedding with its neighbours in a fixed window:

$$c_t = \frac{1}{2k+1} \sum_{j=t-k}^{t+k} x_j$$

- Neighbors often carry the needed information. In “the pitcher threw a fastball”, the word immediately after `pitcher` tells us it is a person rather than a jug
- But the window weights tokens by *position*, when we want to weight them by *relevance*. In “the crane by the lake”, the token that disambiguates `crane` is `lake`, three positions away, while the two tokens in between contribute nothing
- Attention replaces the fixed positional weights with weights the model computes from the content of the tokens themselves [7]

4 Attention

Take a sentence of T tokens with embeddings $x_1, \dots, x_T$, each in $\mathbb{R}^d$, and build the contextualized embedding for position i .

4.1 Step 1: score every pair

- We need a number saying how relevant token j is to token i . The inner product gives this number:

$$s_{ij} = x_i \cdot x_j \tag{1}$$

- The inner product is positive when x_j has a positive projection onto x_i , zero when the two are orthogonal, and negative when the projection is negative — so a large s_{ij} means the two tokens point the same way in embedding space
- In “the crane by the lake”, $s_{\text{crane}, \text{lake}}$ should come out largest

i Watch

Starmer explains cosine similarity^a [8], the inner product with the lengths divided out.

^a<https://www.youtube.com/watch?v=e9U0QAFbLI>

4.2 Step 2: turn the scores into weights

- We want weights that are non-negative and sum to one, so that the result is a weighted *average* rather than an arbitrary combination. The softmax gives such weights:

$$\alpha_{ij} = \frac{\exp\{s_{ij}\}}{\sum_{k=1}^T \exp\{s_{ik}\}}$$

- The row α_i is the *attention distribution* for token i : how much of each other token it reads

i Read

GeeksforGeeks explains the softmax function^a [9].

^a<https://www.geeksforgeeks.org/deep-learning/the-role-of-softmax-in-neural-networks-detailed-explanation-and-applications/>

4.3 Step 3: take the weighted sum

$$c_i = \sum_{j=1}^T \alpha_{ij} x_j$$

- Attention pulls each token towards the tokens it scored highly against, and away from the ones it did not. **crane** moves towards **lake**

4.4 Queries, keys and values

- The three roles a token plays in those steps have names:
 - The token doing the asking is the **query**
 - The tokens being compared against are the **keys**
 - The vectors that get averaged are the **values**
- Written with all T tokens stacked into the rows of Q , K and V , the whole computation is one expression:

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

- The $\sqrt{d_k}$ in the denominator does not appear in the score of Equation 1, and it matters. With d_k large, the entries of $QK^\top$ have standard deviation growing like $\sqrt{d_k}$. The softmax of a vector with a few very large entries is nearly one-hot, and its gradient is then nearly zero. Dividing by $\sqrt{d_k}$ keeps the scores in a range where the softmax still has a usable gradient

4.5 Where are the trainable parameters?

- Steps 1 to 3 have none. Every quantity is a fixed function of the input embeddings, so there is nothing for gradient descent to learn
- We fix this by giving the query, key and value roles their own linear layers:

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

- Now W^Q, W^K, W^V are the parameters. The model learns *what to compare* (the query and key projections) separately from *what to pass along* (the value projection). That is why we use three matrices rather than one
- Together, the projections and the three steps form one **self-attention block**. The block is “self” attention because the queries, keys and values all come from the same sequence

i Watch

- Sanderson draws the attention pattern, from 4:29^a [2, Ch. 6].
- Starmer explains attention^b [10].

^a<https://www.youtube.com/watch?v=eMlx5ffNoYc&t=269s>

^b<https://www.youtube.com/watch?v=PSs6nxngL6k>

i Watch

- The CS224N lecture on self-attention and transformers explains self-attention, from 16:22^a [4], Lecture 9.
- Starmer works through the matrix math of attention^b [11].

^a<https://www.youtube.com/watch?v=ptuGllU5SQQ&t=982s>

^b<https://www.youtube.com/watch?v=KphmOJnLAdI>

i Read

GeeksforGeeks explains self-attention^a [12].

^a<https://www.geeksforgeeks.org/nlp/self-attention-in-nlp/>

5 Multi-head attention

- One block learns one notion of relevance. But **crane** relates to **lake** for a different reason than a verb relates to its subject, and a single attention distribution has to average those notions together
- So we run h blocks in parallel, each with its own W_m^Q, W_m^K, W_m^V , and concatenate their outputs:

$$\text{MultiHead}(X) = [\text{head}_1 \quad \dots \quad \text{head}_h] W^O$$

- Each *head* learns a different kind of relationship. In a trained model the heads are interpretable often enough to be worth looking at: some track syntactic dependencies, some track coreference, some attend almost entirely to the previous token
- In practice we give each head d/h dimensions rather than d . Then multi-head attention costs about the same as single-head attention on the full width

i Watch

- Sanderson explains multiple heads, from 19:19^a [2, Ch. 6].
- The CS224N lecture explains multi-headed attention, from 42:57^b [4], Lecture 9.

^a<https://www.youtube.com/watch?v=eMlx5ffNoYc&t=1159s>

^b<https://www.youtube.com/watch?v=ptuGIIU5SQQ&t=2577s>

i Read

GeeksforGeeks explains multi-head attention^a [13].

^a<https://www.geeksforgeeks.org/nlp/multi-head-attention-mechanism/>

6 The transformer block

- Multi-head attention mixes information *across* positions. It does no per-position processing at all, so we pair it with a feedforward network (FFN) that processes each position separately:

$$\text{FFN}(z) = \max(0, zW_1 + b_1)W_2 + b_2$$

- We wrap both sublayers in a residual connection and a layer normalization — the “Add & Norm” of the original diagram:

$$z = \text{LayerNorm}(x + \text{MultiHead}(x))$$

$$y = \text{LayerNorm}(z + \text{FFN}(z))$$

- The residual connections make depth trainable, for the same reason as in Backpropagation: they give the gradient a path to the input that does not pass through the sublayer’s weights
- That whole assembly is one **transformer block**, the T in Generative Pre-trained Transformer (GPT)

6.1 Position

- Attention is permutation-equivariant: permute the rows of X and every c_i is permuted with them, unchanged. The mechanism has no notion of order
- That is a serious problem for language, where “the owl saw the cat” and “the cat saw the owl” use the same tokens. The standard fix adds a *positional encoding* to each embedding before the first block. Then a token’s vector carries where it sits as well as what it is

i Watch

- The CS224N lecture on self-attention builds the transformer, from 36:18^a [4], Lecture 9.
- Starmer explains the transformer^b [14].

^a<https://www.youtube.com/watch?v=ptuGllU5SQQ&t=2178s>

^b<https://www.youtube.com/watch?v=zxQyTK8quyY>

i Watch

The first lecture of Stanford’s Transformers and Large Language Models course (CME 295) goes from tokenization and embeddings through attention to the full transformer^a [15], Lecture 1.

^a<https://www.youtube.com/watch?v=114i2Kz-LZA>

i Read

GeeksforGeeks explains:

- the transformer^a [16]
- layer normalization^b [17]
- positional encoding^c [18]

^a<https://www.geeksforgeeks.org/machine-learning/getting-started-with-transformers/>

^b<https://www.geeksforgeeks.org/deep-learning/what-is-layer-normalization/>

^c<https://www.geeksforgeeks.org/nlp/positional-encoding-in-transformers/>

i Read

The University of Washington’s Machine Learning course (CSE 446) slides cover attention^a and transformers^b [19], Lectures 22 and 23.

^a<https://courses.cs.washington.edu/courses/cse446/26sp/schedule/lectures/22-attention.pdf>

^b<https://courses.cs.washington.edu/courses/cse446/26sp/schedule/lectures/23-transformers.pdf>

7 From blocks to large language models

- Transformer blocks take a sequence of vectors and return a sequence of vectors of the same shape, so they stack:

$$X \rightarrow \text{block} \rightarrow \text{block} \rightarrow \dots \rightarrow \text{block} \rightarrow H$$

- A stack of them is the main body of a large language model. To predict the next token, we take the final-layer vector at the last position and pass it through one more linear layer with a softmax, producing a distribution over the whole vocabulary
- Training is next-token prediction on very large quantities of text, most of it collected from the web. The models are large: GPT-3 has 175 billion trainable parameters [20]. Parameter counts for later commercial models are mostly undisclosed, so read any figure you meet for them as an unverified estimate, and check where it came from before repeating it

- The model learns more than language from that corpus. It also absorbs the information in the text, the copyrighted material in the text, and the biases in the text. We take up this point in Ethics and Fairness

i Watch

Starmer explains decoder-only transformers^a [21], the kind behind ChatGPT.

^a<https://www.youtube.com/watch?v=bQ5BoolX9Ag>

i Read

GeeksforGeeks introduces large language models^a [22].

^a<https://www.geeksforgeeks.org/artificial-intelligence/large-language-model-llm/>

8 Transformers beyond language

- Nothing in the attention mechanism is specific to words. It needs a set of vectors and a reason to compare them, which many kinds of data supply:
 - **Images**, cut into patches, each patch embedded as a token
 - **Audio**, as a sequence of spectrogram frames
 - **Proteins**, as a sequence of amino-acid residues
- As a result, one architecture now does work that used to need a different model family per modality

i Read

GeeksforGeeks explains the vision transformer^a [23], which embeds image patches as tokens.

^a<https://www.geeksforgeeks.org/deep-learning/vision-transformer-vit-architecture/>

9 Attention in code

The code in this section computes scaled dot-product attention for a four-token sentence with a three-dimensional embedding, and prints the attention distribution for the first token. We chose the embeddings so that the first and last tokens point in a similar direction (the `crane/lake` case). The weight the first token puts on the last should then be the largest of its weights on the other three tokens.

9.1 Python

```
import numpy as np

def softmax_rows(scores):
    shifted = scores - scores.max(axis=1, keepdims=True)
    weights = np.exp(shifted)
    return weights / weights.sum(axis=1, keepdims=True)

def attention(queries, keys, values):
    scores = queries @ keys.T / np.sqrt(keys.shape[1])
    weights = softmax_rows(scores)
    return weights @ values, weights
```

```

# "the crane by the lake": token 0 and token 3 point the same way.
# Hand-picked 3-dimensional embeddings keep the arithmetic checkable by eye;
# a trained model learns embeddings with hundreds of dimensions.
x = np.array([[0.8, 0.1, 0.4],      # crane
              [0.0, 1.0, 0.1],      # by
              [0.1, 0.9, 0.0],      # the
              [0.7, 0.2, 0.5]])      # lake

context, weights = attention(x, x, x)
print(np.round(weights[0], 3))

```

```
[0.298 0.203 0.206 0.293]
```

```
print(np.round(context[0], 3))
```

```
[0.464 0.476 0.286]
```

9.2 R

```

softmax_rows <- function(scores) {
  shifted <- sweep(scores, 1, apply(scores, 1, max))
  weights <- exp(shifted)
  sweep(weights, 1, rowSums(weights), FUN = "/")
}

attention <- function(queries, keys, values) {
  scores <- queries %*% t(keys) / sqrt(ncol(keys))
  weights <- softmax_rows(scores)
  list(context = weights %*% values, weights = weights)
}

# "the crane by the lake": token 1 and token 4 point the same way.
# Hand-picked 3-dimensional embeddings keep the arithmetic checkable by eye;
# a trained model learns embeddings with hundreds of dimensions.
x <- rbind(
  c(0.8, 0.1, 0.4),      # crane
  c(0.0, 1.0, 0.1),      # by
  c(0.1, 0.9, 0.0),      # the
  c(0.7, 0.2, 0.5)       # lake
)

result <- attention(x, x, x)
print(round(result$weights[1, ], 3))

```

```
[1] 0.298 0.203 0.206 0.293
```

```
print(round(result$context[1, ], 3))
```

```
[1] 0.464 0.476 0.286
```

9.3 Julia

```

function softmax_rows(scores)
  shifted = scores .- maximum(scores, dims = 2)
  weights = exp.(shifted)
  weights ./ sum(weights, dims = 2)
end;

```

```
function attention(queries, keys, values)
    scores = queries * keys' / sqrt(size(keys, 2))
    weights = softmax_rows(scores)
    (context = weights * values, weights = weights)
end;

# "the crane by the lake": token 1 and token 4 point the same way.
# Hand-picked 3-dimensional embeddings keep the arithmetic checkable by eye;
# a trained model learns embeddings with hundreds of dimensions.
x = [0.8 0.1 0.4;      # crane
     0.0 1.0 0.1;      # by
     0.1 0.9 0.0;      # the
     0.7 0.2 0.5       # lake
    ];

result = attention(x, x, x);
println(round.(result.weights[1, :], digits = 3))
```

```
[0.298, 0.203, 0.206, 0.293]
```

```
println(round.(result.context[1, :], digits = 3))
```

```
[0.464, 0.476, 0.286]
```

The function `softmax_rows` subtracts each row's maximum before exponentiating. That subtraction changes nothing mathematically, because the constant cancels between numerator and denominator. Without it, a score of a few hundred overflows to `Inf` and the whole row becomes `NaN`. Every softmax used in practice does this.

10 Further reading

- Vaswani et al. [24] — the paper the transformer block comes from
- Lin et al. [25] — a survey of the variants that followed, free on arXiv:2106.04554¹
- Amidi and Amidi [15] — Stanford's course on transformers and large language models, whose first lecture, Transformers² (1 h 44 min), goes from tokenization and embeddings through attention to the full transformer; its syllabus³ lists what each later lecture covers
- Starmer [26], on word embeddings, attention and transformers, illustrated, with PyTorch tutorials

Watch

Andrew Ng's lectures for DeepLearning.AI introduce attention for sequence models: the intuition [27]; and the model itself [28].

References

- [1] Wikipedia contributors, "Transformer (deep learning architecture) — Wikipedia, the free encyclopedia." 2026. Available: [https://en.wikipedia.org/wiki/Transformer_\(deep_learning_architecture\)](https://en.wikipedia.org/wiki/Transformer_(deep_learning_architecture))
- [2] G. Sanderson, "Neural networks." 3Blue1Brown, video series on YouTube; chapters 5 and 6 added in 2024, 2017. Available: https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi

¹<https://arxiv.org/abs/2106.04554>

²<https://www.youtube.com/watch?v=114i2Kz-LZA>

³<https://cme295.stanford.edu/syllabus/>

- [3] GeeksforGeeks, “N-gram language modelling with NLTK.” GeeksforGeeks, tutorial article, 2026. Available: <https://www.geeksforgeeks.org/nlp/n-gram-language-modelling-with-nltk/>
- [4] C. Manning *et al.*, “CS224N: Natural language processing with deep learning, winter 2021.” Stanford Online, lecture videos on YouTube, 2021. Available: <https://www.youtube.com/playlist?list=PLoROMvovd4rOSH4v6133s9LFPRHjEmbmJ>
- [5] J. Starmer, “Word embedding and Word2Vec, clearly explained.” StatQuest, video on YouTube, 2023. Available: <https://www.youtube.com/watch?v=viZrOnJclY0>
- [6] GeeksforGeeks, “Word embeddings in NLP.” GeeksforGeeks, tutorial article, 2026. Available: <https://www.geeksforgeeks.org/nlp/word-embeddings-in-nlp/>
- [7] Wikipedia contributors, “Attention (machine learning) — Wikipedia, the free encyclopedia.” 2026. Available: [https://en.wikipedia.org/wiki/Attention_\(machine_learning\)](https://en.wikipedia.org/wiki/Attention_(machine_learning))
- [8] J. Starmer, “Cosine similarity, clearly explained.” StatQuest, video on YouTube, 2023. Available: <https://www.youtube.com/watch?v=e9U0QAFbLI>
- [9] GeeksforGeeks, “Softmax activation function in neural networks.” GeeksforGeeks, tutorial article, 2025. Available: <https://www.geeksforgeeks.org/deep-learning/the-role-of-softmax-in-neural-networks-detailed-explanation-and-applications/>
- [10] J. Starmer, “Attention for neural networks, clearly explained.” StatQuest, video on YouTube, 2023. Available: <https://www.youtube.com/watch?v=PSs6nxngL6k>
- [11] J. Starmer, “The matrix math behind transformer neural networks, one step at a time.” StatQuest, video on YouTube, 2024. Available: <https://www.youtube.com/watch?v=KphmOJnLAdI>
- [12] GeeksforGeeks, “Self - attention in NLP.” GeeksforGeeks, tutorial article, 2026. Available: <https://www.geeksforgeeks.org/nlp/self-attention-in-nlp/>
- [13] GeeksforGeeks, “Multi-head attention mechanism.” GeeksforGeeks, tutorial article, 2026. Available: <https://www.geeksforgeeks.org/nlp/multi-head-attention-mechanism/>
- [14] J. Starmer, “Transformer neural networks, ChatGPT’s foundation, clearly explained.” StatQuest, video on YouTube, 2023. Available: <https://www.youtube.com/watch?v=zxQyTK8quyY>
- [15] A. Amidi and S. Amidi, “CME 295: Transformers and large language models, autumn 2026.” Stanford University, course webpage with lecture slides and recorded lectures, 2026. Available: <https://cme295.stanford.edu/>
- [16] GeeksforGeeks, “Transformers in machine learning.” GeeksforGeeks, tutorial article, 2026. Available: <https://www.geeksforgeeks.org/machine-learning/getting-started-with-transformers/>
- [17] GeeksforGeeks, “What is layer normalization?” GeeksforGeeks, tutorial article, 2026. Available: <https://www.geeksforgeeks.org/deep-learning/what-is-layer-normalization/>
- [18] GeeksforGeeks, “Positional encoding in transformers.” GeeksforGeeks, tutorial article, 2026. Available: <https://www.geeksforgeeks.org/nlp/positional-encoding-in-transformers/>
- [19] N. Jaques, “CSE 446/546: Machine learning, lecture slides.” University of Washington, 2026. Accessed: Oct. 12, 2026. [Online]. Available: <https://courses.cs.washington.edu/courses/cse446/26sp/schedule/lectures/>
- [20] T. B. Brown *et al.*, “Language models are few-shot learners,” in *Advances in neural information processing systems*, 2020, pp. 1877–1901. Available: <https://arxiv.org/abs/2005.14165>
- [21] J. Starmer, “Decoder-only transformers, ChatGPTs specific transformer, clearly explained.” StatQuest, video on YouTube, 2023. Available: <https://www.youtube.com/watch?v=bQ5BoolX9Ag>
- [22] GeeksforGeeks, “Introduction to large language model (LLM).” GeeksforGeeks, tutorial article, 2026. Available: <https://www.geeksforgeeks.org/artificial-intelligence/large-language-model-llm/>
- [23] GeeksforGeeks, “Vision transformer (ViT) architecture.” GeeksforGeeks, tutorial article, 2025. Available: <https://www.geeksforgeeks.org/deep-learning/vision-transformer-vit-architecture/>
- [24] A. Vaswani *et al.*, “Attention is all you need,” in *Advances in neural information processing systems*, 2017. Available: <https://arxiv.org/abs/1706.03762>
- [25] T. Lin, Y. Wang, X. Liu, and X. Qiu, “A survey of transformers,” *AI Open*, vol. 3, pp. 111–132, 2022, doi: 10.1016/j.aiopen.2022.10.001⁴.
- [26] J. Starmer, *The StatQuest illustrated guide to neural networks and AI: With hands-on examples in PyTorch*. Independently published, 2025. Available: <https://statquest.org/statquest-store/>

⁴<https://doi.org/10.1016/j.aiopen.2022.10.001>

- [27] A. Ng, “Attention model intuition.” DeepLearning.AI, Deep Learning Specialization, Course 5; video on YouTube, 2018. Available: <https://www.youtube.com/watch?v=SysgYptB198>
- [28] A. Ng, “Attention model.” DeepLearning.AI, Deep Learning Specialization, Course 5; video on YouTube, 2018. Available: <https://www.youtube.com/watch?v=quoGRI-1l0A>