Deep Learning (4): Sequence Data and Transformers

(for high school and first-year university students)

From RNNs to Attention

Planned and edited by Yoshio Okamoto / Written by Claude Opus 5.5 (Anthropic)

Ver. 0.2 October 7, 2026

Note: The following content was generated by AI. The editor has checked it, but please be aware that it may still contain errors or mistaken assumptions.

© 2026 Yoshio Okamoto. Free to use for educational purposes (please cite the source).

Introduction

Part 3 dealt with data in which the **spatial arrangement** has meaning, such as images. Part 4 deals with data in which the **temporal order** has meaning, such as text and time series (**sequence data**). In the first half we look at the classical **recurrent neural network** (RNN) and its limits, and in the second half at the **Transformer**, the foundation of today's LLMs.

When we read a long novel, we remember the characters and events that appeared earlier, and recall them as needed as we read on. "How to remember what came before, and how to recall it": this is the central problem in handling sequence data.

Part	Title
Part 1	Basics of Neural Networks
Part 2	Backpropagation and Training in Practice
Part 3	Convolutional Neural Networks
Part 4	**Sequence Data and Transformers**

How to read this booklet The main text is written so that it can be read with high-school mathematics (the dot product of vectors, exponential functions and logarithms, trigonometric functions, the summation sign $\sum$, and conditional probability). The computation of Attention is explained without matrices, using dot products of the vectors of individual tokens and weighted sums. The general notation with matrices (RNN, LSTM, Attention, multi-head attention) is collected in the **footnotes** and in **Appendix A** at the end. You can follow the main text even if you skip the appendix. All numerical values in the examples were actually computed (program in Appendix B).

Notation

- The t-th input of a sequence (at time t) is written x_t, and the hidden state h_t (the subscript t is the time). Strictly, both are vectors of several numbers, but in the main text we sometimes treat them as single numbers to explain the ideas.
- Bold $\mathbf{q}$, $\mathbf{k}$, $\mathbf{v}$ are vectors, and $\mathbf{q} \cdot \mathbf{k}$ is the dot product.
- d is the dimension (number of components) of the token vectors, d_k is the dimension of queries and keys, and n is the length of the sequence.

1 Sequence data

1.1 Characteristics of sequence data

Text, speech, changes in temperature or stock prices, seismic waveforms and so on are data whose values are **lined up in order**. They have the following characteristics.

1. **The order has meaning**: "the dog bit the man" and "the man bit the dog" consist of the same words but mean different things.
2. **The length is not fixed**: sentences have different lengths.
3. **Distant positions are related**: in "Ken gave Amy a book. He …," "He" refers to "Ken," which appeared much earlier.

The networks of Parts 1–3 had a fixed input size, so they cannot handle these properties as they are.

1.2 The task of language modeling

Below, as a typical task for sequence data, we consider **predicting the next word from the words so far** (a **language model**). It is the same task as predictive text on a smartphone suggesting "you" after "Thank." That is, we predict

$$p(x_{t+1} \mid x_1, x_2, \ldots, x_t),$$

"the probability that the next word is x_{t+1}, given $x_1, \ldots, x_t$" (a conditional probability). It is a multi-class classification of choosing the next word from all the words of the vocabulary, with as many classes as the size of the vocabulary.

> An LLM is exactly a model that has learned this task from a huge amount of text.

2 Recurrent neural networks (RNNs)

2.1 How they work

> An RNN (Recurrent Neural Network) is a network that **has a "hidden state" h_t summarizing past information, and updates it each time it reads an input**.
> Imagine reading a book with **a single small memo sheet**. Each time you read a word, you rewrite the contents of the memo as "the story so far." When predicting the next word, you rely only on this memo. This memo is the hidden state h_t.

Written for the case where the hidden state and the input are each a single number,

$$h_t = \tanh\big(w_h\, h_{t-1} + w_x\, x_t + b\big).$$

h_{t-1} is the hidden state at the previous time, w_h and w_x are weights, and b is a bias. The weighted sum of "the previous hidden state (the memo up to a moment ago)" and "the current input (the word just read)" is passed through the activation function tanh to give the new hidden state (the rewritten memo). The output $\hat{y}_t$ at each time (the probabilities of the next word) is computed from the hidden state h_t with a fully connected layer and the softmax function.[1]

Because it feeds its own output back (recurs) into the input at the next time, it is called **recurrent**. Unrolled in time, it looks like Figure 1.

[1] In practice, the hidden state and the input are vectors and the weights are matrices: $\mathbf{h}_t = \tanh(W_h\mathbf{h}_{t-1} + W_x\mathbf{x}_t + \mathbf{b})$, $\hat{\mathbf{y}}_t = \mathrm{softmax}(W_y\mathbf{h}_t + \mathbf{c})$ (Appendix A.1).

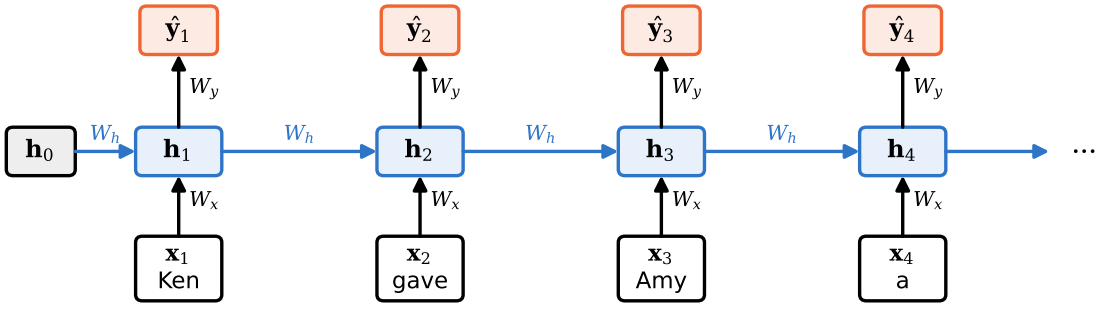


Figure 1 An RNN unrolled in time. The same weights W_x, W_h, W_y are used at every time (bold symbols are vectors and W are weight matrices). The hidden state $\mathbf{h}_t$ summarizes what has been read so far and passes it to the next time.

> What matters is that **the same weights are used at every time**. Just as the convolution of Part 3 shared weights in the spatial direction, an RNN **shares weights in the time direction**. The rule for rewriting the memo is the same whichever page you are reading. Therefore the same network can be applied to sequences of any length.

2.2 Training and vanishing gradients

RNNs are also trained by backpropagation. Regarding the unrolled Figure 1 as one deep network, the error is sent backward from the last time to the first (**Backpropagation Through Time**, BPTT).

How does the error reach the past? The route by which the hidden state h_t at time t affects the hidden state h_T at a later time T is a single road:

$$h_t \longrightarrow h_{t+1} \longrightarrow h_{t+2} \longrightarrow \cdots \longrightarrow h_T.$$

There is no shortcut connecting h_t directly to h_T; the information is always passed on one step at a time.

So for each step, we consider "if h_{s-1} is changed a little, by how many times does h_s change?" This is the one-step derivative $\dfrac{\partial h_s}{\partial h_{s-1}}$. For example, if changing h_t a little changes h_{t+1} by 0.5 times as much, and changing h_{t+1} a little changes h_{t+2} by 0.5 times as much, then a change in h_t reaches h_{t+2} at $0.5 \times 0.5 = 0.25$ times. As in the chain rule of Part 2 (differentiation of composite functions, the gear analogy), **the factor along a single road is the product of the factors of each step**. Therefore

$$\frac{\partial h_T}{\partial h_t} = \frac{\partial h_{t+1}}{\partial h_t} \times \frac{\partial h_{t+2}}{\partial h_{t+1}} \times \cdots \times \frac{\partial h_T}{\partial h_{T-1}}.$$

The number of factors multiplied is the number of steps from t to T, that is, $T - t$.

In backpropagation, the error that arrived at time T ($\partial L/\partial h_T$) multiplied by this factor becomes the error that arrives at time t ($\partial L/\partial h_t$). This is the time-direction version of "errors are sent from behind by multiplication," which we saw in Part 2.[2]

[2] University textbooks write this with the product symbol $\prod$ (pi; the multiplication version of the summation sign $\sum$) as $\dfrac{\partial \mathbf{h}_T}{\partial \mathbf{h}_t} = \displaystyle\prod_{s=t+1}^{T} \frac{\partial \mathbf{h}_s}{\partial \mathbf{h}_{s-1}}$, meaning "multiply together all the $\frac{\partial \mathbf{h}_s}{\partial \mathbf{h}_{s-1}}$ from $s = t+1$ to T" (Appendix A.2).

Computing a simple case As the simplest case, consider a hidden state that is a single number, with no activation function:

$$h_t = a\, h_{t-1} + x_t.$$

Changing h_{t-1} a little changes h_t by a times as much, so the one-step derivative is a at every step. Therefore

$$\frac{\partial h_T}{\partial h_t} = \underbrace{a \times a \times \cdots \times a}_{T-t \text{ factors}} = a^{T-t}$$

(Figure 2). The error also travels backward at this rate.

> The **one-step derivative is multiplied $T-t$ times**. If $|a| < 1$, it approaches 0 exponentially as $T-t$ grows ($0.9^{50} \approx 0.005$); if $|a| > 1$, it grows explosively ($1.1^{50} \approx 117$). This is the same structure as the vanishing and exploding gradients of Part 2, and moreover the same weight is multiplied many times.[3]
> Remember the **telephone game** of Part 2. In a line of 50 people, if the voice becomes 0.9 times as loud with each person, only 0.5% of the loudness reaches the last person.

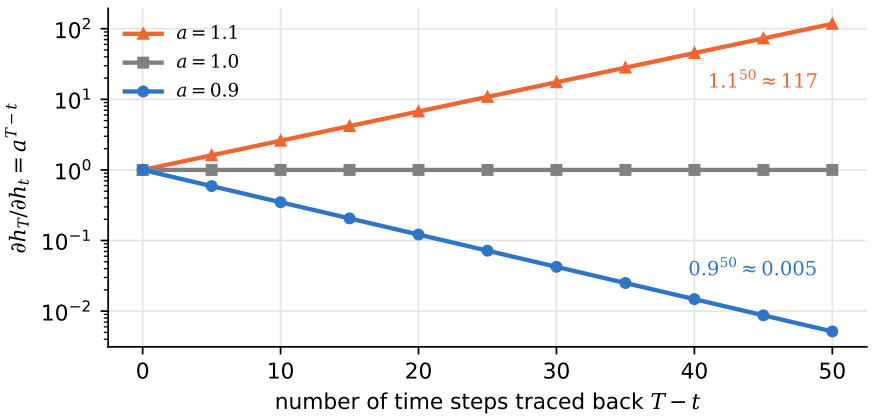


Figure 2 When the one-step derivative is a, the size a^{T-t} of the error traced back $T-t$ steps (log-scale vertical axis).

As a result, RNNs are **poor at learning relations between distant positions**. It is hard to get the error from "He" back to "Ken" 50 words earlier.

2.3 LSTM

LSTM (Long Short-Term Memory; Hochreiter and Schmidhuber, 1997) was devised to ease this problem. Besides the hidden state, an LSTM has a "memory cell" c_t, which is **updated by addition**:[4]

$$c_t = f_t \times c_{t-1} + i_t \times g_t.$$

Here f_t (the forget gate) and i_t (the input gate) are quantities between 0 and 1 that adjust, depending on the input, "how much of the previous memory to keep" and "how much of the new information g_t to write in."

[3] In the vector case, the same thing happens depending on whether the absolute values of the eigenvalues of W_h are smaller or larger than 1 (Appendix A.2). The derivative of tanh is at most 1, which tilts things further toward vanishing.

[4] In practice, c_t, f_t, i_t and g_t are vectors and $\times$ is the componentwise product (written $\odot$) (Appendix A.3).

Whereas an RNN rewrites its whole memo each time, an LSTM is a notebook that uses **an eraser** (forget gate: how much to erase) and **a pen** (input gate: how much to add) separately, so important things can be left as they are.
If $f_t \approx 1$, the memory passes unchanged to the next time, and the error also arrives without shrinking through multiplication. This is the same idea as the residual connection of Part 2 (the direct telephone line run alongside the line of people).

LSTMs were widely used in machine translation and speech recognition until the mid-2010s.

2.4 The fundamental limits of RNNs

Even with the improvement of LSTM, RNNs kept the following limits.

1. **All information passes through h_t**: however long the sentence, past information must be packed into a vector h_t of fixed size. It is like writing the synopsis of a long novel on a single memo sheet that never gets bigger.
2. **The path to distant positions is long**: for the information at position 1 to reach position t, it must go through $t-1$ updates (Figure 3(a)).
3. **No parallel computation**: computing h_t needs h_{t-1}, so the computation can only be done one time step at a time, in order. It is like a bucket brigade in a single line, where the next person cannot move until the previous one has finished. The parallel computing power of GPUs cannot be used.

3 Attention: referring directly to the positions needed

3.1 The idea

What solved the limits of RNNs at their root is **Attention** (the attention mechanism). The idea is simple: **instead of packing past information into one vector, go and get the information at the needed position directly, when it is needed** (Figure 3(b)).
Stop relying on one memo sheet, and instead **keep the book itself at hand and turn back to the page when you need it**: this is Attention. When "He" appears, you just turn back the pages and read the part about "Ken" directly.

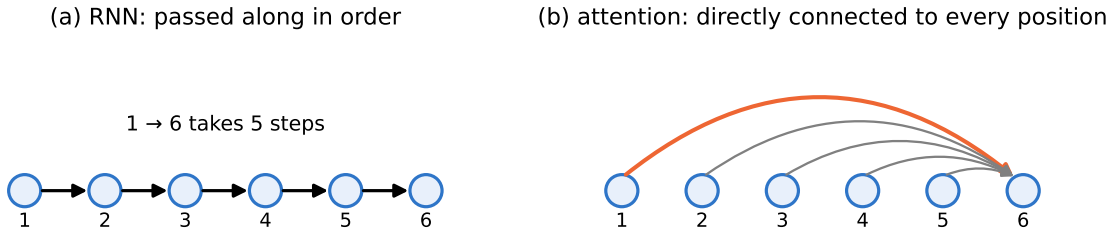


Figure 3 Paths of information. (a) In an RNN, the information at position 1 is passed on 5 times before it reaches position 6. (b) With Attention, every position can be referred to directly in one step.

3.2 The computation of Attention

In Attention, three kinds of vectors are made from the vector of each token (word):[5]

[5]These are made by multiplying the token's vector by separate weight matrices W_Q, W_K and W_V (Appendix A.4).

- **query q**: "what information" the token is looking for
- **key k**: a label of "what information" the token has
- **value v**: the information the token actually passes on

> Imagine comparing your question (the query) with the titles on the spines of books in a library (the keys), and collecting the contents (the values) by reading more carefully the books that match better.[6] The point is that you do not choose just one book, but read several books "a little each," according to how well they match.

The computation by which the token at position i refers to the tokens at positions $1, \ldots, n$ has three stages.

1. **Scores**: compute the dot product of its own query $\mathbf{q}_i$ with the key $\mathbf{k}_j$ of each position, and divide by $\sqrt{d_k}$:
$$s_j = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}} \qquad (j = 1, \ldots, n).$$
The larger the dot product, the "better the match."[7]
2. **Attention weights**: turn the scores into probabilities with the softmax function:
$$a_j = \frac{e^{s_j}}{e^{s_1} + e^{s_2} + \cdots + e^{s_n}}.$$
The weights are all positive and add up to 1. These are called **attention weights**: the allocation of "which books to read, in what proportions."
3. **Output**: mix the values of the positions with the attention weights:
$$\mathbf{o}_i = a_1\mathbf{v}_1 + a_2\mathbf{v}_2 + \cdots + a_n\mathbf{v}_n.$$

> That is, the output at each position is **the values of all positions mixed with the weights of "how much to refer to each"**. Since the tokens within one sentence refer to one another, this is called **self-attention**.

This computation, done for all positions together, can be written with matrices as the single formula $\mathrm{softmax}\big(QK^{\mathsf{T}}/\sqrt{d_k}\big)V$ (Appendix A.4).

3.3 Example

We take four words from "Ken gave Amy a book. He …," set the dimension of queries and keys to $d_k = 2$, and give them the following vectors (program in Appendix B).

Word	Query **q**	Key **k**	Value **v**
Ken	(0, 1)	(2, 1)	(1, 0, 0)
Amy	(1, 0)	(1, −1)	(0, 1, 0)
book	(1, 1)	(−1, 0)	(0, 0, 1)
he	(2, 2)	(0, 1)	(0, 0, 0)

[6] Intuitive explanations of queries, keys and values are also found in Section 3.3 of the separate document "Reading microgpt" and in Section 3 of "Introduction to How LLMs Work."

[7] Dividing by $\sqrt{d_k}$ prevents the dot products from becoming too large when the dimension d_k is large, which would make the result of softmax extreme (one almost 1, the rest almost 0). For this reason, this computation is called Scaled Dot-Product Attention.

The dot products of the query $(2, 2)$ of "he" with each key are

$$\text{Ken} : 2\cdot 2+2\cdot 1 = 6, \quad \text{Amy} : 2\cdot 1+2\cdot(-1) = 0, \quad \text{book} : 2\cdot(-1)+2\cdot 0 = -2, \quad \text{he} : 2\cdot 0+2\cdot 1 = 2.$$

Dividing by $\sqrt{2}$ and taking the softmax function, the attention weights are

$$(0.929,\ 0.013,\ 0.003,\ 0.055),$$

so "he" refers strongly to "Ken." The output of "he," mixing the values, is

$$0.929\,(1, 0, 0) + 0.013\,(0, 1, 0) + 0.003\,(0, 0, 1) + 0.055\,(0, 0, 0) = (0.929,\ 0.013,\ 0.003),$$

which is almost exactly the value of "Ken." The token "he" opened the page of "Ken" and brought its contents back almost unchanged. The attention weights for all the words are shown in Figure 4(a).

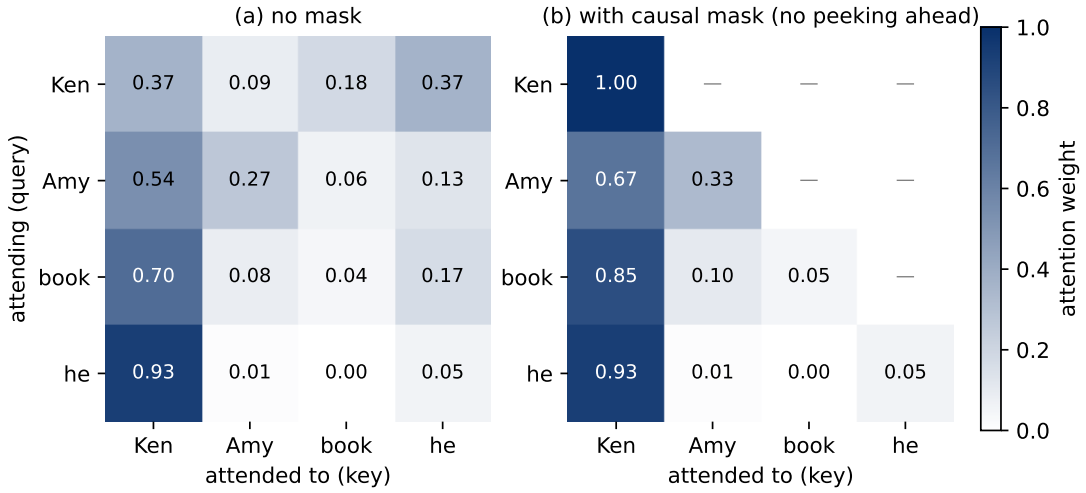


Figure 4 Attention weights in the example. Rows are the attending side (query) and columns the attended side (key); each row sums to 1. (b) is with the causal mask (Section 3.4) applied.

> In real models, these query, key and value vectors are not given by people; they are made automatically according to the context, by learning the weights that produce them.

3.4 The causal mask

In a language model, we "predict the next word," so the computation at position i **must not look at words after it (in the future)**. It is like trying to guess the culprit while reading a mystery novel: you must not look at the last page first. So the scores s_j of future positions $j > i$ are replaced by $-\infty$ (minus infinity) before taking the softmax function.[8]

Since $e^{-\infty} = 0$, the attention weights to future positions become 0 (Figure 4(b)). This is called the **causal mask**. It is a blindfold over the future pages. The last word, "he," comes after all the words anyway, so its result does not change with or without the mask.

3.5 No information about order: positional encoding

Self-attention has the following property.

[8] In matrix terms, this corresponds to adding to the matrix of scores a matrix M that is "0 if $j \le i$, $-\infty$ if $j > i$" (Appendix A.5).

If the order of the input tokens is rearranged, the outputs are merely rearranged in the same way; the output value for each token does not change.

This is because dot products and weighted sums are all computations that do not depend on order (the program in Appendix B checks that the output of "he" is the same when the input is reversed). It is like putting word cards in a bag and shaking it: you can tell which words were there, but not the order they were in. As it stands, "the dog bit the man" and "the man bit the dog" cannot be distinguished.

So a **vector representing the position** (a **positional encoding**) is added to each token's vector before input. It is like writing "1st," "2nd" and so on on each word card. The original Transformer used, for components $2i$ and $2i+1$ of the vector at position pos,

$$PE_{\text{pos},\,2i} = \sin\Big(\frac{\text{pos}}{10000^{2i/d}}\Big), \qquad PE_{\text{pos},\,2i+1} = \cos\Big(\frac{\text{pos}}{10000^{2i/d}}\Big)$$

(Figure 5). $10000^{2i/d}$ becomes larger as the component number i increases, so each component is a sine wave with a different period: components with small numbers change quickly, and those with large numbers change slowly.

Just as the combination of the second, minute and hour hands of a clock expresses the time, the combination of these components expresses the position.

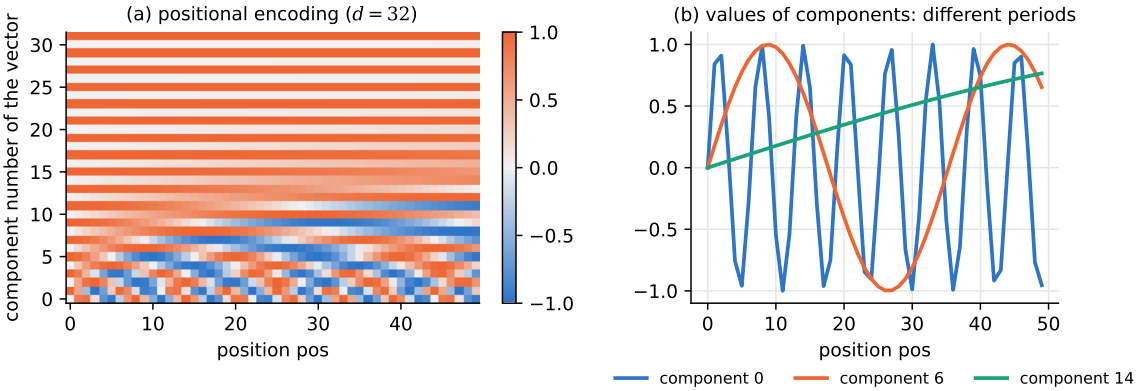


Figure 5 Positional encoding. (a) The vector for each position (horizontal axis), with 32 components arranged vertically. (b) How the values of some components change.

Recent models often use other methods, such as rotating the query and key vectors according to position (RoPE). In any case, they share the point that **Attention itself does not know the order, so information about order must be supplied from outside**.

4 The Transformer

4.1 Multi-head attention

In practice, h sets of Attention are computed in parallel (**multi-head attention**). Each set (**head**) makes queries, keys and values with its own weights and computes Attention separately. Finally, the outputs of the h heads are joined together and transformed once more to mix them (formulas in Appendix A.6).

The dimension of each head is usually $d_k = d/h$. It is like several people reading the same sentence from different points of view, and each head is expected to capture a different kind of

relation (grammatical relations, the relation between a pronoun and what it refers to, and so on). If a language teacher, a history teacher and a science teacher read the same passage, each pays attention to different things; the readings of all of them are brought together.

4.2 The Transformer block

The Transformer (Vaswani et al., 2017) consists of N layers of the following **block** stacked together (Figure 6).

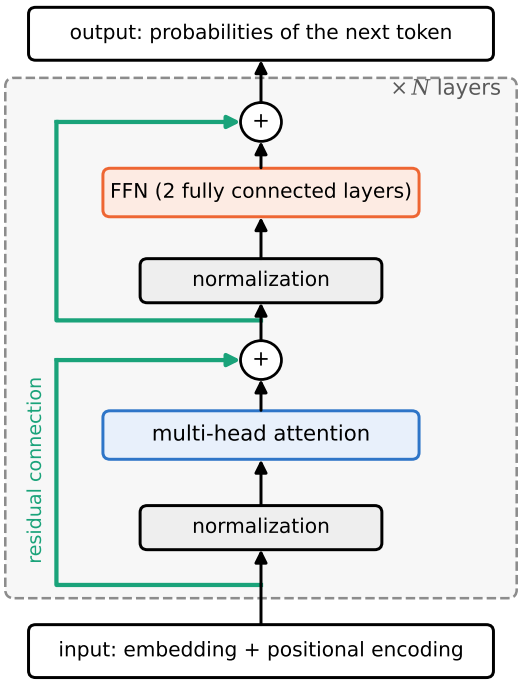


Figure 6 A Transformer block (the structure on the side that generates output, used in LLMs). The green lines are residual connections.

In one block, the following two-stage computation is applied to the sequence of token vectors X:

$$X \leftarrow X + \mathrm{MultiHead}\big(\mathrm{Norm}(X)\big)$$
$$X \leftarrow X + \mathrm{FFN}\big(\mathrm{Norm}(X)\big)$$

- **Multi-head attention** (MultiHead): the part where tokens exchange information with one another.
- **FFN** (Feed-Forward Network): a two-layer fully connected network that transforms each token's vector separately (exactly the multilayer network of Part 1). The dimension of its hidden layer is usually about $4d$.
- **Residual connections** ($X+\cdots$) and **normalization** (Norm): the techniques seen in Part 2 for making deep networks trainable.

That is, the Transformer is a network that **alternates "exchange of information between tokens (Attention)" and "transformation of each token (FFN)" dozens of times**. A fully connected layer and the softmax function applied to the output of the last layer give the probabilities of the next token.
In terms of a class, it is like alternating, dozens of times, **time for group discussion**

(Attention: listening to each other's opinions and taking them in) and **time for each person to organize their thoughts** (FFN: sorting out what they heard in their own mind).

4.3 The number of parameters

The parameters of one block are roughly as follows (biases and normalization are small and omitted). A weight matrix with d rows and d columns consists of d^2 numbers.

Part	Weights	Parameters
Attention	four matrices, for queries, keys, values and the final mixing (each $d \times d$)	$4d^2$
FFN	two matrices, layer 1 ($d \to 4d$) and layer 2 ($4d \to d$) (each $4d^2$)	$8d^2$
Total		$12d^2$

For example, a model with $d = 768$ and 12 layers has $12 \times 12 \times 768^2 \approx 85$ million parameters in its blocks and about 39 million in the word embeddings (vocabulary of 50,257 words $\times$ 768), about 124 million in total. This is the scale of the smallest GPT-2 model. Llama 3 8B has $d = 4096$ and 32 layers; its structure is somewhat different, so it does not follow this formula exactly, but by the same reasoning it comes to about 8 billion.

4.4 Training and generation

In **training**, as in Part 2, the cross-entropy of next-token prediction,

$$L = -\sum_t \log p(x_{t+1} \mid x_1, \ldots, x_t),$$

is minimized by backpropagation. It is the sum, over all positions, of minus the logarithm of "the probability given to the correct next token" at each position.

The correct answer (the next token) comes from the text itself, so no one needs to attach labels by hand (**self-supervised learning**; "Supervised and Unsupervised Learning," Section 9.2). Fill-in-the-blank questions, made by hiding the words one at a time, can be made automatically from any text.

Thanks to the causal mask, **the predictions for all positions of a sentence can be made together in one computation**. The prediction at position t only looks up to position t, so computing them at the same time is not "cheating." All the fill-in-the-blank questions can be answered at once without seeing the answers. Not having to compute one time step at a time in order, as an RNN does, and being able to train on large amounts of data in parallel on GPUs, is a major reason why Transformers could be scaled up.

In **generation**, the model repeatedly predicts one token, adds it to the end, and uses that as input again.[9] It is the same as building a sentence by tapping the suggested word on your phone's predictive text again and again.

[9] How generation works is also explained with an actual program in Section 7 of the separate document "Reading microgpt."

4.5 Comparison with RNNs

	RNN	Transformer
How past information is held	Summarized in a hidden state h_t of fixed size (one memo sheet)	The vectors of all positions are kept as they are (the book itself)
Path to distant positions	Long, in proportion to the distance	One step (direct reference)
Parallel computation in training	Not possible (in time order)	Possible (all positions at once)
Computation for sequence length n	Proportional to n	Proportional to n^2 (dot products of all pairs)
Information about order	Built into the structure	Supplied from outside by positional encoding

The weakness of the Transformer is that, since it computes attention for all pairs, **the computation is proportional to the square of the sequence length**. Just as n people each shaking hands once with everyone else makes about $n^2/2$ handshakes, doubling the number of tokens makes the computation about four times larger. Ways of handling long documents are still being actively researched.

5 Summary

1. In sequence data, the order has meaning, the length is not fixed, and distant positions are related.
2. An RNN passes a hidden state along in time and shares weights in the time direction. But its gradients vanish or explode exponentially, like a^{T-t}, so it is hard to learn distant relations, and it cannot compute in parallel.
3. Attention decides "where to look" by the dot products of queries and keys, and takes out the information at the needed positions directly. All positions can be computed together.
4. In language models, the causal mask hides the future, and since Attention does not know the order, positional encoding supplies the positions.
5. The Transformer stacks dozens of layers of "information exchange between tokens by Attention" and "transformation of each token by FFN," with residual connections and normalization in between.

Summary of the whole series

Part	What we learned	Connection with earlier parts
Part 1	Stacking neurons with nonlinear activation functions in between lets hidden layers transform the input into representations that are easy to handle	Logistic regression = one neuron
Part 2	Backpropagation finds the gradients of all parameters efficiently, and gradient descent learns from them	The same "minimization of the loss" as the least-squares method and gradient descent
Part 3	Weight sharing in the spatial direction (convolution) learns local features of images hierarchically	The image version of Part 1's "transformation of representations"
Part 4	From weight sharing in the time direction (RNN) to direct reference (Attention). The Transformer became the foundation of LLMs	Vanishing gradients and residual connections of Part 2; weight sharing of Part 3

From the least-squares method to LLMs, what stays the same is the framework of **"choose a model, choose a loss function, and minimize the loss using gradients."** What has changed is the structure of the model (what is connected and how) and its scale.

Appendix A Writing with matrices and vectors

This appendix uses partial derivatives and matrices (first-year university linear algebra).

Notation Vectors are written in bold lowercase ($\mathbf{x}_t$, $\mathbf{h}_t$) and matrices in uppercase (W). The superscript T is the **transpose** (swapping rows and columns), not a power. $\mathbf{q}^{\mathsf{T}}\mathbf{k}$ is the dot product. $\odot$ is the componentwise product. tanh is the hyperbolic tangent function, and softmax is the softmax function applied to each row. From A.4 on, we use the $n \times d$ matrix X whose **rows** are the vectors of the n tokens, and multiply weight matrices from the right, as in $Q = XW_Q$.

A.1 RNN

$$\mathbf{h}_t = \tanh\bigl(W_h\mathbf{h}_{t-1} + W_x\mathbf{x}_t + \mathbf{b}\bigr), \qquad \hat{\mathbf{y}}_t = \mathrm{softmax}\bigl(W_y\mathbf{h}_t + \mathbf{c}\bigr).$$

W_h, W_x and W_y are common to all times.

A.2 Backpropagation through time and vanishing gradients

As explained in Section 2.2 of the main text, by the time the error at time T reaches time t, the one-step derivatives have been multiplied $T - t$ times:

$$\frac{\partial \mathbf{h}_T}{\partial \mathbf{h}_t} = \prod_{s=t+1}^{T} \frac{\partial \mathbf{h}_s}{\partial \mathbf{h}_{s-1}}.$$

$\prod$ is the product symbol. When the hidden state is a vector, $\partial\mathbf{h}_s/\partial\mathbf{h}_{s-1}$ is the matrix (the Jacobian matrix) listing "by how many times each component of $\mathbf{h}_s$ changes when each component of $\mathbf{h}_{s-1}$ is changed a little," and the multiplication is a matrix product. Ignoring the activation function, $\partial\mathbf{h}_s/\partial\mathbf{h}_{s-1} = W_h$, so the same matrix W_h is multiplied $T - t$ times. Decomposing along the eigenvectors of W_h, the component in each direction is multiplied by the eigenvalue λ to the power $T - t$, so directions with $|\lambda| < 1$ vanish and those with $|\lambda| > 1$ explode. The scalar case of Section 2.2 (a^{T-t}) is the case in which W_h is 1×1.

A.3 LSTM

The update of the LSTM memory cell is

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{g}_t.$$

The forget gate $\mathbf{f}_t$ and input gate $\mathbf{i}_t$ are computed with the sigmoid function from the previous hidden state and the current input, and each component takes a value between 0 and 1. $\partial \mathbf{c}_t / \partial \mathbf{c}_{t-1}$ is the matrix with $\mathbf{f}_t$ on its diagonal, so if $\mathbf{f}_t \approx \mathbf{1}$, the error reaches the past with almost no decay.

A.4 Scaled Dot-Product Attention

From X $(n \times d)$, whose rows are the vectors of the n tokens, we make

$$Q = XW_Q, \qquad K = XW_K, \qquad V = XW_V$$

(W_Q, W_K are $d \times d_k$ and W_V is $d \times d_v$), and compute

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

- QK^T is an $n \times n$ matrix whose (i, j) entry is the dot product of "the query of position i" and "the key of position j," that is, the score s_j of Section 3.2 of the main text (multiplied by $\sqrt{d_k}$).
- Applying softmax to each row makes each row a set of attention weights summing to 1.
- Multiplying that by V makes each row "the values of all positions mixed with the attention weights."

A.5 The causal mask

We add

$$M_{ij} = \begin{cases} 0 & (j \le i) \\ -\infty & (j > i) \end{cases}$$

to the matrix of scores before taking softmax:

$$\text{softmax}\left(\frac{QK^\mathsf{T}}{\sqrt{d_k}} + M\right) V.$$

A.6 Multi-head attention and FFN

Each head m has its own weights $W_Q^{(m)}$, $W_K^{(m)}$, $W_V^{(m)}$, and

$$\begin{aligned} \text{head}_m &= \text{Attention}\big(XW_Q^{(m)},\ XW_K^{(m)},\ XW_V^{(m)}\big), \\ \text{MultiHead}(X) &= \big[\,\text{head}_1, \ldots, \text{head}_h\,\big]\, W_O. \end{aligned}$$

[ ] means joining the outputs of the heads side by side. The FFN computes, for each token's vector $\mathbf{x}$,

$$\text{FFN}(\mathbf{x}) = W_2\, f\big(W_1\mathbf{x} + \mathbf{b}_1\big) + \mathbf{b}_2$$

(W_1 is $4d \times d$, W_2 is $d \times 4d$, and f is the activation function).

Appendix B Python code (standard library only)

This program computes the examples of Sections 3.3–3.5. It uses no external libraries, only the standard features of Python 3.

```
# Computing self-attention (Scaled Dot-Product Attention) (standard library only)
import math

tokens = ["Ken", "Amy", "book", "he"]
Q = [[0, 1], [1, 0], [1, 1], [2, 2]]          # query of each token (d_k = 2)
K = [[2, 1], [1, -1], [-1, 0], [0, 1]]        # keys
V = [[1, 0, 0], [0, 1, 0], [0, 0, 1], [0, 0, 0]]  # values

def softmax(s):
    m = max(s)
    e = [math.exp(v - m) for v in s]
    return [v / sum(e) for v in e]

def attention(Q, K, V, causal=False):
    dk = len(Q[0])
    A, out = [], []
    for i, q in enumerate(Q):
        s = [sum(a * b for a, b in zip(q, k)) / math.sqrt(dk) for k in K]
        if causal:                            # do not look at the future (j > i)
            s = [v if j <= i else -math.inf for j, v in enumerate(s)]
        a = softmax(s)
        A.append(a)
        out.append([sum(a[j] * V[j][c] for j in range(len(V))) for c in range(len(V[0]))])
    return A, out

if __name__ == "__main__":
    for causal in (False, True):
        A, out = attention(Q, K, V, causal)
        print("with causal mask" if causal else "without causal mask")
        print("      " + "".join(f"{t:>8s}" for t in tokens))
        for t, row in zip(tokens, A):
            print(f"{t:>4s}  " + "".join(f"{v:8.3f}" for v in row))
        print('output of "he" =', [round(v, 3) for v in out[3]])

    # Reversing the order of the inputs only reverses the outputs (no information about order)
    perm = [3, 2, 1, 0]
    _, out1 = attention(Q, K, V)
    _, out2 = attention([Q[p] for p in perm], [K[p] for p in perm], [V[p] for p in perm])
    print('output of "he" with the input in reverse order =', [round(v, 3) for v in out2[0]])
```

Output:

```
without causal mask
           Ken     Amy    book      he
 Ken     0.365   0.089   0.180   0.365
 Amy     0.539   0.266   0.065   0.131
book     0.703   0.084   0.042   0.171
  he     0.929   0.013   0.003   0.055
output of "he" = [0.929, 0.013, 0.003]
with causal mask
           Ken     Amy    book      he
 Ken     1.000   0.000   0.000   0.000
 Amy     0.670   0.330   0.000   0.000
book     0.848   0.102   0.050   0.000
```

```
  he     0.929   0.013   0.003   0.055
output of "he" = [0.929, 0.013, 0.003]
output of "he" with the input in reverse order = [0.929, 0.013, 0.003]
```