# Course Materials for GEN-AI

*Northeastern University*

These materials have been prepared and sourced for the course **GEN-AI** at Northeastern University. Every effort has been made to provide proper citations and credit for all referenced works.

If you believe any material has been inadequately cited or requires correction, please contact me at:

**Instructor: Ramin Mohammadi**
r.mohammadi@northeastern.edu

*Thank you for your understanding and collaboration.*

---

Content sourced from multiple references - see Reference section for details

# Transformers

## Introduction

Transformers, introduced in the seminal 2017 paper "Attention is All You Need" by Vaswani et al., have revolutionized the field of natural language processing (NLP). This document explores the key concepts behind transformers and their attention mechanisms, tracing the evolution from traditional sequence-to-sequence models to the more advanced transformer architecture.

## Transformer Architecture

Transformers build upon the attention mechanism, introducing several key innovations that have made them the state-of-the-art in many NLP tasks. To motivate the transformer, consider the following passage:

> The restaurant refused to serve me a ham sandwich because it only cooks vegetarian food. In the end, they just gave me two slices of bread. Their ambiance was just as good

as the food and service.

1. **Input Size**: Encoded inputs can be large; even a short passage with 37 words and embeddings of length 1024 results in $37 \times 1024 = 37,888$ dimensions, making fully connected networks impractical for longer texts.
2. **Variable Lengths**: NLP inputs vary in length, so parameter sharing across word positions—similar to convolutional networks—is essential.
3. **Ambiguity**: Words need connections across the text to resolve ambiguities (e.g., "it" referring to "restaurant"), and these connections should adjust based on context, extending over large spans.

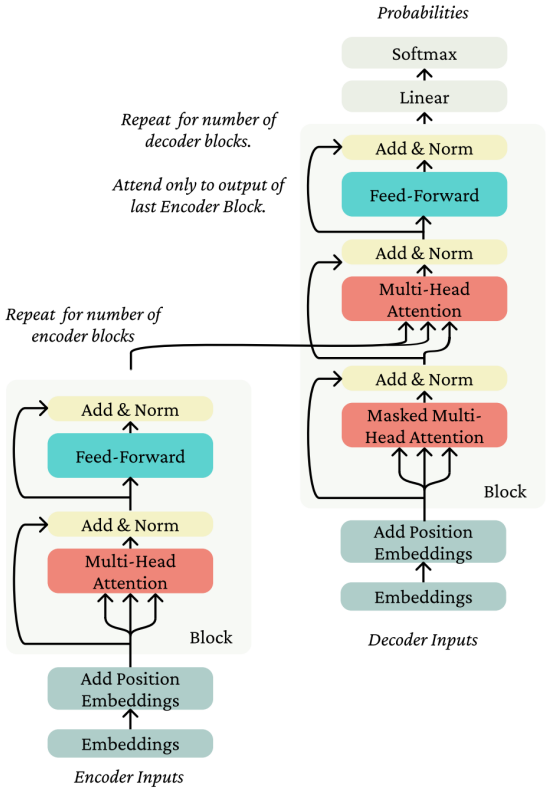


Figure 1: Transformer Architecture

# Applications of Transformer Architectures

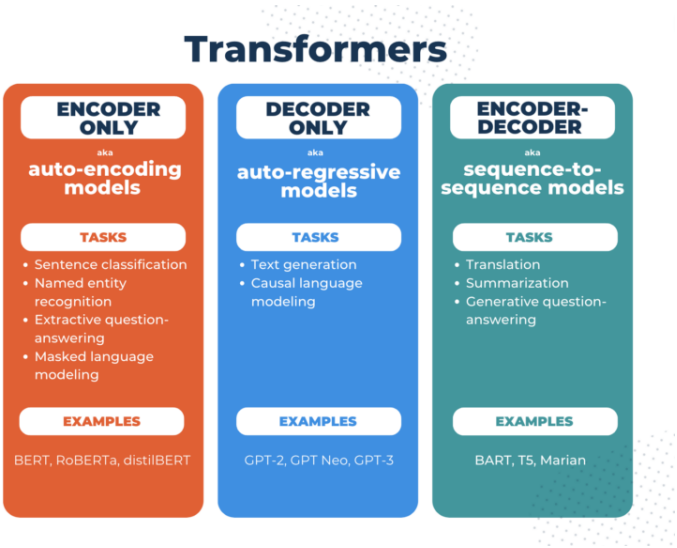


Figure 2: Transformer Types

Transformer models are commonly classified into three types based on their architecture and the types of tasks they are suited for:

1. **Encoder-Only Models (Auto-Encoding Models)**:
   - **Tasks**: Encoder-only models are primarily used for understanding tasks, such as sentence classification, named entity recognition, extractive question-answering, and masked language modeling.
   - **Examples**: BERT, RoBERTa, and distilBERT.

   These models utilize only the encoder component of the transformer, making them ideal for tasks focused on analyzing and understanding input text.

2. **Decoder-Only Models (Auto-Regressive Models)**:
   - **Tasks**: Decoder-only models are designed for generation tasks, including text generation and causal language modeling.
   - **Examples**: GPT-2, GPT Neo, and GPT-3.

   These models rely solely on the decoder part of the transformer architecture and are well-suited for tasks that involve generating text sequentially, where each new word is generated based on previously generated words.

3. **Encoder-Decoder Models (Sequence-to-Sequence Models)**:
   - **Tasks**: Encoder-decoder models are often applied to tasks that require both an input and output sequence, such as translation, summarization, and generative question-answering.
   - **Examples**: BART, T5, and Marian.

   These models use both the encoder and decoder components, making them effective for tasks that involve transforming an input sequence into a different output sequence, like translating text from one language to another or summarizing content.

# Key, Query, Value

In databases, the concept of **key-value** pairs is a method for storing and retrieving data, where:

- **key** uniquely identifies a piece of data.
- **value** is the actual data or information associated with the key.
- **query** is the search term or request used to find the relevant key, and consequently retrieve the associated value.

For example, in a dictionary, the word is the key, and its definition is the value. A query is used to retrieve specific values by matching or searching with a relevant key.

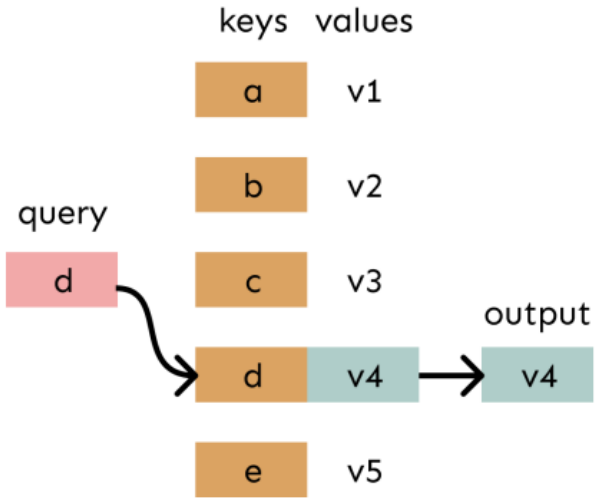


Figure 3: Lookup Table

In transformers, the **query-key-value** mechanism is similar but applied in a dynamic way:

- **Query**: Represents the current word or position in the sequence that is "searching" for related information.
- **Key**: Represents potential information points within the sequence that could be relevant to the query.
- **Value**: Holds the actual data or representation that the model wants to retrieve.

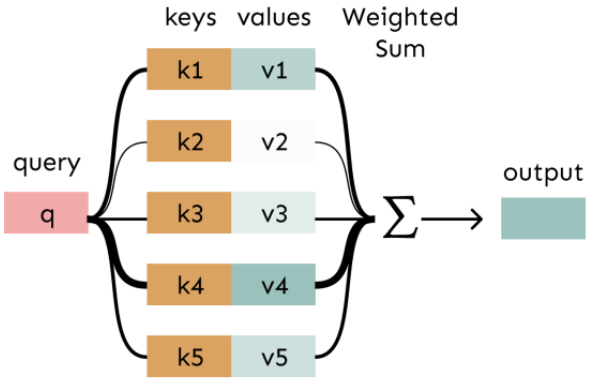


Figure 4: Query-Key-Value

# Self-Attention

Attention, in general, is a method for using a query to retrieve information from a set of key-value pairs by giving more emphasis to values associated with keys that closely match the query. Rather than selecting specific values outright, attention creates a weighted average of all values, placing greater weight on those tied to the keys most similar to the query. In *self-attention*, the same elements are used to define the query, keys, and values, allowing the model to assess relationships within a single sequence.

## The key-query-value self-attention mechanism

There are many forms of self-attention; the form we'll discuss here is currently the most popular. It's called key-query-value self-attention.
Consider a token $x_i$ in the sequence $x_{1:n}$. We can define the following:

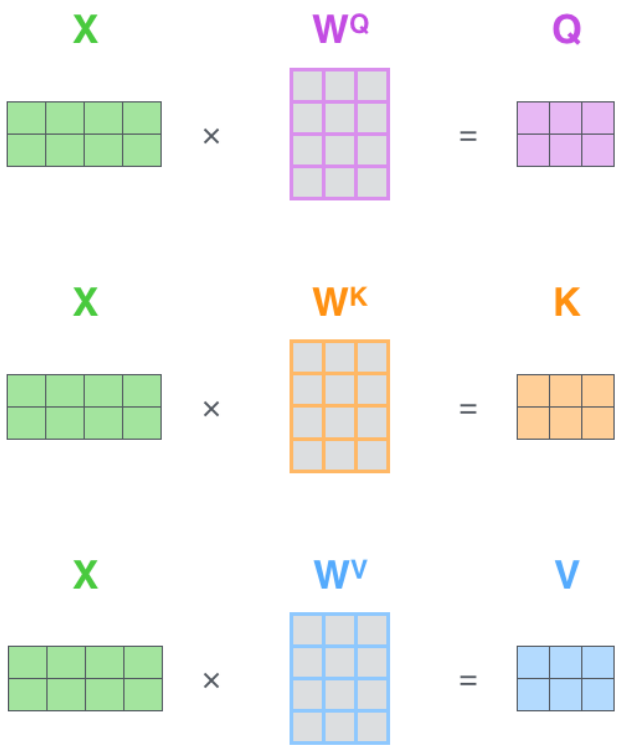


Figure 5: Every row in the X matrix corresponds to a word in the input sentence.

- **Query** $q_i$: For the token $x_i$, the query vector is defined as $q_i = W_Q x_i$, where $W_Q \in \mathbb{R}^{d \times d}$ is a weight matrix.
- **Key** $k_j$: For each token $x_j$ in the sequence $\{x_1, \ldots, x_n\}$, the key vector is defined as $k_j = W_K x_j$, where $W_K \in \mathbb{R}^{d \times d}$ is another weight matrix.
- **Value** $v_j$: For each token $x_j$ in the sequence, the value vector is defined as $v_j = W_V x_j$, where $W_V \in \mathbb{R}^{d \times d}$ is an additional weight matrix.

Our contextual representation $h_i$ of $x_i$ is a linear combination (that is, a weighted sum) of the values of the sequence,

$$h_i = \sum_{j=1}^{n} \alpha_{ij} v_j, \tag{7}$$

where the weights, these $\alpha_{ij}$, control the strength of contribution of each $v_j$. Going back to our key-value store analogy, the $\alpha_{ij}$ softly selects what data to look up. We define these weights by computing the affinities between the keys and the query, $q_i^T k_j$, and then computing the softmax over the sequence:

$$\alpha_{ij} = \frac{\exp\left(\frac{q_i^T k_j}{\sqrt{d}}\right)}{\sum_{j'=1}^{n} \exp\left(\frac{q_i^T k_{j'}}{\sqrt{d}}\right)}. \tag{8}$$

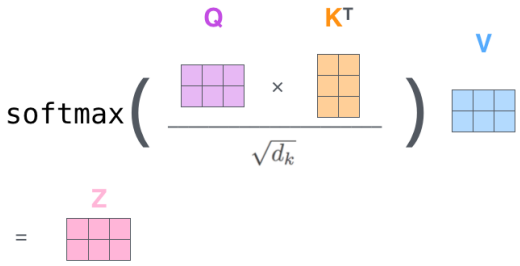


Figure 6: Self-Attention Weight

The $\sqrt{d}$ term is a scaling factor to counteract the effect of large dot products in high dimensions.
Intuitively, what we've done by this operation is take our element $x_i$ and look in its own sequence $x_{1:n}$ to figure out what information (in an informal sense), from what other tokens, should be used in representing $x_i$ in context. Imagine we have a sentence:

*"The cat chased the mouse."*

In a self-attention mechanism, each word in this sentence acts as a **query**, **key**, and **value** at the same time. Here's how it works:

- **Query**: Suppose we want to understand the word *"cat"* better, particularly in relation to the other words in the sentence. Here, *"cat"* becomes the **query**—it "asks" about other words to find which ones are relevant to it.
- **Keys**: All words in the sentence (*"The," "cat," "chased," "the," "mouse"*) are treated as **keys**. Each word has its own key that represents its meaning or features. So, *"mouse"* has a key that represents it, *"chased"* has its key, and so on.
- **Values**: Similarly, each word also has a **value**, which is essentially the information that the model wants to retrieve if a word is considered relevant to the query. The values often represent the actual context or content of the words.

The use of matrices $K, Q, V$ intuitively allow us to use different views of the $x_i$ for the different roles of key, query, and value. We perform this operation to build $h_i$ for all $i \in \{1, \ldots, n\}$.

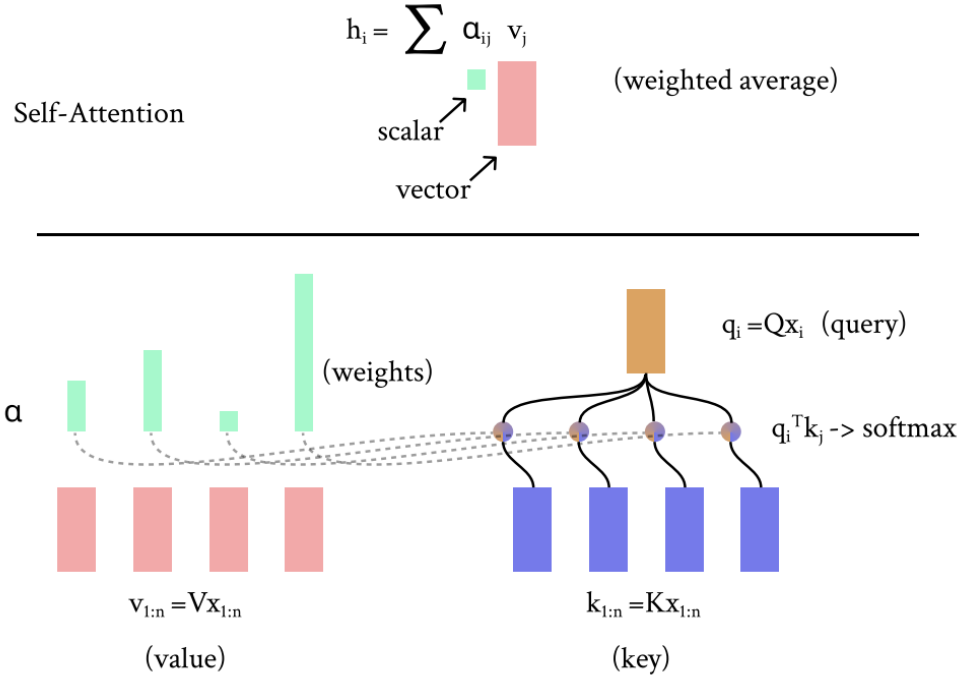


Figure 7: Self-Attention Mechanisim

## Self-attention as routing:

The self-attention mechanism takes $N$ inputs $\mathbf{x}_1, \ldots, \mathbf{x}_N \in \mathbb{R}^D$ (here $N = 3$ and $D = 4$) and processes each separately to compute $N$ value vectors. The $n^{\text{th}}$ output $\text{sa}_n[\mathbf{x}_1, \ldots, \mathbf{x}_N]$ (written as $\text{sa}_n[\mathbf{x}_\bullet]$ for short which is a self-attention block) is then computed as a weighted sum of the $N$ value vectors, where the weights are positive and sum to one.

(a) Output $\text{sa}_1[\mathbf{x}_\bullet]$ is computed as $a[\mathbf{x}_1, \mathbf{x}_1] = 0.1$ times the first value vector, $a[\mathbf{x}_2, \mathbf{x}_1] = 0.3$ times the second value vector, and $a[\mathbf{x}_3, \mathbf{x}_1] = 0.6$ times the third value vector.

(b) Output $\text{sa}_2[\mathbf{x}_\bullet]$ is computed in the same way, but this time with weights of 0.5, 0.2, and 0.3.

(c) The weighting for output $\text{sa}_3[\mathbf{x}_\bullet]$ is different again. Each output can hence be thought of as a different routing of the $N$ values.

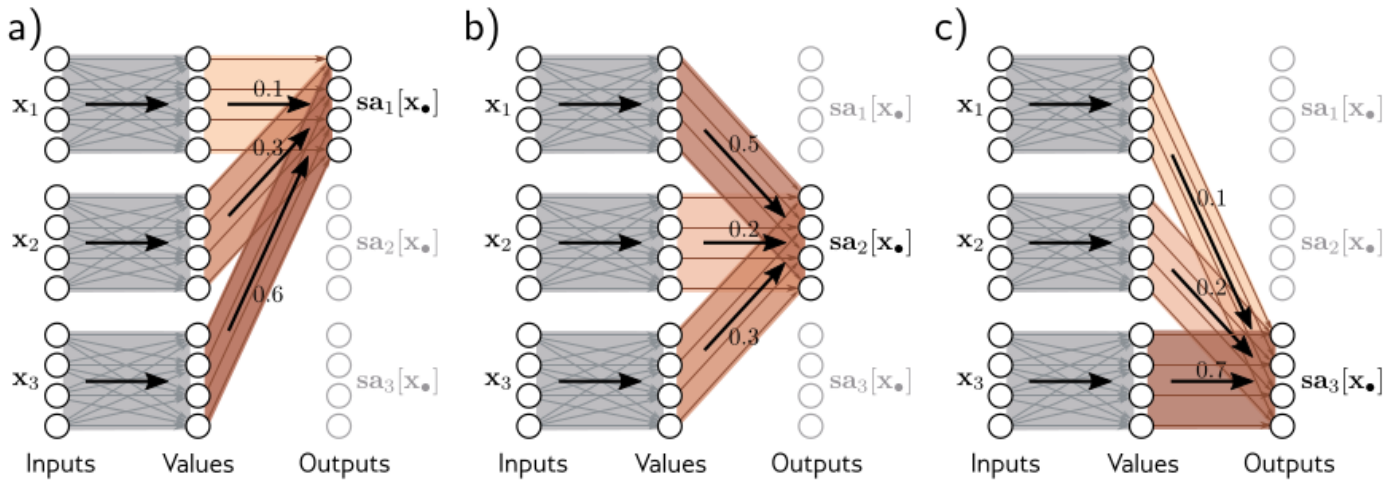


Figure 8: Self-attention as routing.

# Computing and weighting values

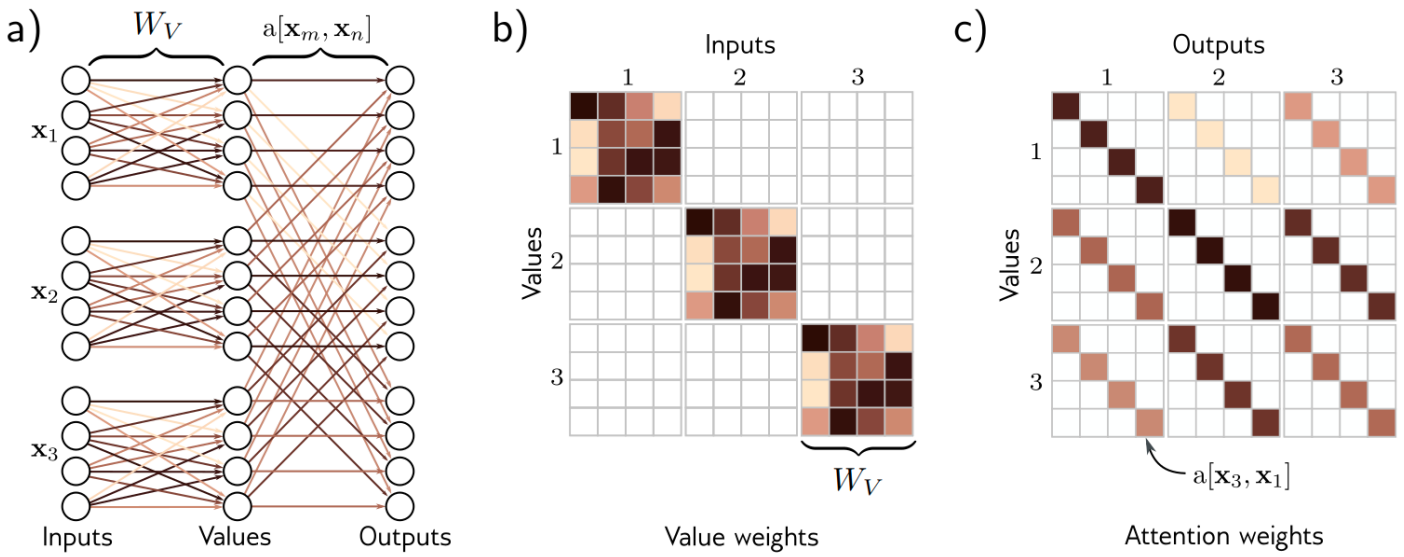


Figure 9: Self-attention for $N = 3$ inputs $\mathbf{x}_n$, each with dimension $D = 4$. Each input $\mathbf{x}_n$ is operated on independently by the same weights $W_v$ (same color equals same weight) and biases $\beta_v$ (not shown) to form the values $\beta_v + W_v\mathbf{x}_n$. Each output is a linear combination of the values, with a shared attention weight $a[\mathbf{x}_m, \mathbf{x}_n]$ defining the contribution of the $m^{\text{th}}$ value to the $n^{\text{th}}$ output. (b) Matrix showing block sparsity of linear transformation $W_v$ between inputs and values. (c) Matrix showing sparsity of attention weights relating values and outputs.

**Panel (b):**

- Each input (like $x_1$) goes through a transformation using weights (shown as $W_v$) to create a "value" representation $V$.
- This panel shows the matrix of $W_v$ between inputs and values, representing how each input is transformed.

**Panel (c):**

- Next, attention weights $a[x_m, x_n]$ are calculated by comparing queries ($Q$) and keys ($K$) across inputs.
- This panel displays the **attention weights** as a matrix, where darker cells indicate stronger similarity between the query of one input and the key of another.
- For example, if the query of $x_1$ closely matches the key of $x_2$, then the value of $x_2$ will have a stronger influence on the output for $x_1$.

**Panel (a):**

- Finally, the values are combined according to the attention weights to produce the final outputs.
- The lines connecting each query with keys show how the attention weights control the influence of each value on each output.
- This panel visually represents how values are routed to produce the outputs, with each line representing an attention weight.

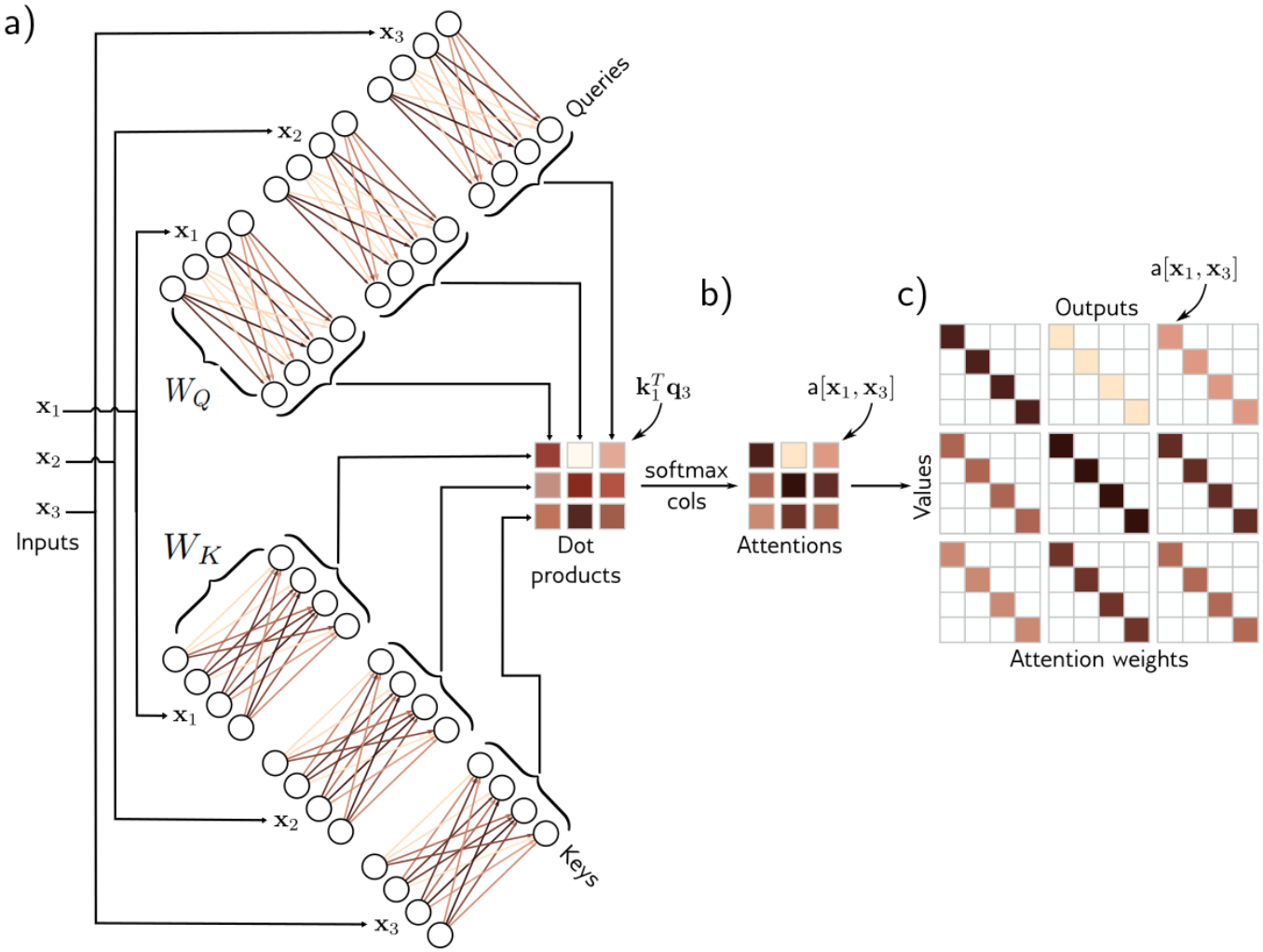


Figure 10: Computing attention weights.

(a) Query vectors $\mathbf{q}_n = \beta_q + W_q\mathbf{x}_n$ and key vectors $\mathbf{k}_n = \beta_k + W_k\mathbf{x}_n$ are computed for each input $\mathbf{x}_n$.

(b) The dot products between each query and the three keys are passed through a softmax function to form non-negative attentions that sum to one.

(c) These route the value vectors (Figure 8) via the sparse matrix from Figure 9-c.

## Self-Attention in Summary

The $n$-th output is a weighted sum of the same linear transformation $\mathbf{v}_\bullet = \beta_v + W_v\mathbf{x}_\bullet$ applied to all of the inputs, where these attention weights are positive and sum to one. The weights depend on a measure of similarity between input $\mathbf{x}_n$ and the other inputs. There is no activation function, but the mechanism is nonlinear due to the dot-product and a softmax operation used to compute the attention weights.
Note that this mechanism fulfills the initial requirements. First, there is a single shared set of parameters $\phi = \{\beta_v, W_v, \beta_q, W_q, \beta_k, W$ This is independent of the number of inputs $N$, so the network can be applied to different sequence lengths. Second, there are connections between the inputs (words), and the strength of these connections depends on the inputs themselves via the attention weights.

### Self-Attention In Matrix Form

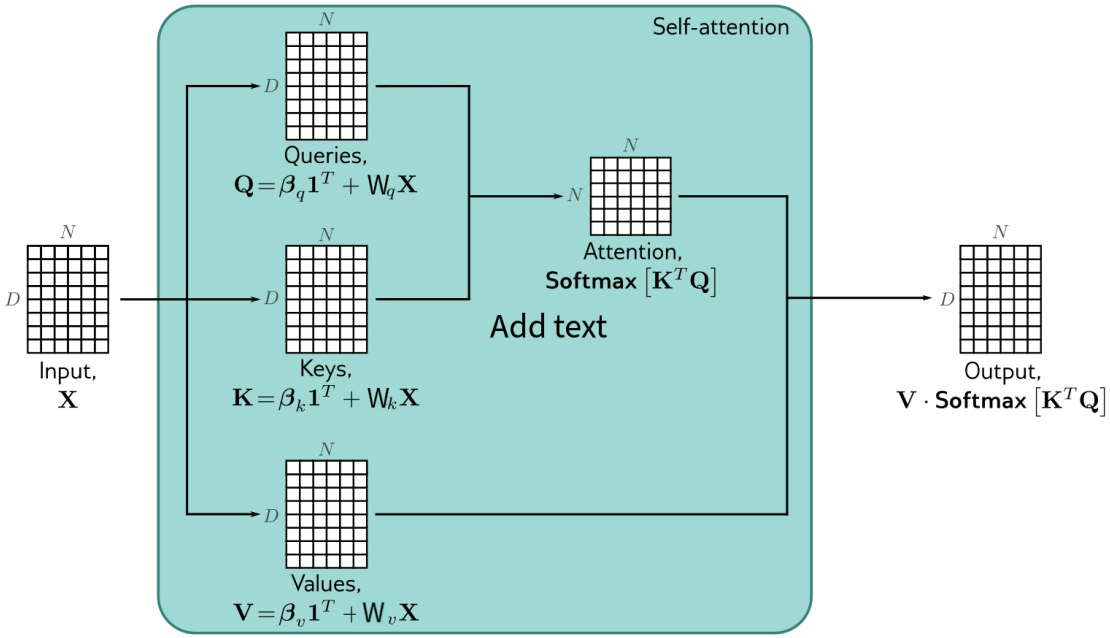


Figure 11: Self-attention in matrix form. Self-attention can be implemented efficiently if we store the $N$ input vectors $\mathbf{x}_n$ in the columns of the $D \times N$ matrix $\mathbf{X}$. The input $\mathbf{X}$ is operated on separately by the query matrix $\mathbf{Q}$, key matrix $\mathbf{K}$, and value matrix $\mathbf{V}$. The dot products are then computed using matrix multiplication, and a softmax operation is applied independently to each column of the resulting matrix to calculate the attentions. Finally, the values are post-multiplied by the attentions to create an output of the same size as the input.

The above computation can be written in a compact form if the $N$ inputs $\mathbf{x}_n$ form the columns of the $D \times N$ matrix $\mathbf{X}$. The values, queries, and keys can be computed as:

$$\begin{aligned}\mathbf{V}[\mathbf{X}] &= \beta_v\mathbf{1}^\top + W_v\mathbf{X} \\ \mathbf{Q}[\mathbf{X}] &= \beta_q\mathbf{1}^\top + W_q\mathbf{X} \\ \mathbf{K}[\mathbf{X}] &= \beta_k\mathbf{1}^\top + W_k\mathbf{X},\end{aligned}$$

where $\mathbf{1}$ is an $N \times 1$ vector containing ones. The self-attention computation is then:

$$\text{Sa}[\mathbf{X}] = \mathbf{V}[\mathbf{X}] \cdot \text{Softmax}\left(\mathbf{K}[\mathbf{X}]^\top\mathbf{Q}[\mathbf{X}]\right),$$

where the function Softmax[·] takes a matrix and performs the softmax operation independently on each of its columns (figure 11). In this formulation, we have explicitly included the dependence of the values, queries, and keys on the input $\mathbf{X}$ to emphasize that self-attention computes a kind of triple product based on the inputs. However, from now on, we will drop this dependence and just write:

$$\text{Sa}[\mathbf{X}] = \mathbf{V} \cdot \text{Softmax}\left(\mathbf{K}^\top\mathbf{Q}\right).$$

## Position representations

Consider the sequence *the oven cooked the bread so*. This is a different sequence than *the bread cooked the oven so*, as you might guess. The former sentence has us making delicious bread, and the latter we might interpret as the bread somehow

breaking the oven. In a recurrent neural network, the order of the sequence defines the order of the rollout, so the two sequences have different representations. In the self-attention operation, there's no built-in notion of order.
To see this, let's take a look at self-attention on this sequence. We have a set of vectors $x_{1:n}$ for *the oven cooked the bread so*, which we can write as

$$x_{1:n} = [x_{\text{the}}, x_{\text{oven}}, x_{\text{cooked}}, x_{\text{the}}, x_{\text{bread}}, x_{\text{so}}] \in \mathbb{R}^{5\times d} \tag{9}$$

As an example, consider performing self-attention to represent the second $x$ in context. The weights over the context are as follows, recalling that $q_i = Qx_i$ for all words, and $k_j = Kx_j$; likewise:

$$\alpha_{so} = \text{softmax}\left(\begin{bmatrix} q_{so}^T k_{\text{the}} \\ q_{so}^T k_{\text{oven}} \\ q_{so}^T k_{\text{cooked}} \\ q_{so}^T k_{\text{the}} \\ q_{so}^T k_{\text{bread}} \\ q_{so}^T k_{so} \end{bmatrix}\right) \tag{10}$$

So, the weight $\alpha_{so,0}$, the amount that we look up the first word, (by writing out the softmax) is,

$$\alpha_{so,0} = \frac{\exp(q_{so}^T k_{\text{the}})}{\exp(q_{so}^T k_{\text{the}}) + \cdots + \exp(q_{so}^T k_{\text{bread}})}. \tag{11}$$

So, $\alpha \in \mathbb{R}^5$ are our weights, and we compute the weighted average in Equation 7 with these weights to compute $h_{so}$.
For the reordered sentence *the bread cooked the oven*, note that $\alpha_{so,0}$ is identical. The numerator hasn't changed, and the denominator hasn't changed; we've just rearranged terms in the sum. Likewise for $\alpha_{so,\text{bread}}$ and $\alpha_{so,\text{oven}}$, you can compute that they too are identical independent of the ordering of the sequence. This all comes back down to the two facts that (1) the representation of $x$ is not position-dependent; it's just $Ew$ for whatever word $w$, and (2) there's no dependence on position in the self-attention operations.

## Position representation through learned embeddings

To represent position in self-attention, you either need to (1) use vectors that are already position-dependent as inputs, or (2) change the self-attention operation itself. One common solution is a simple implementation of (1). We posit a new parameter matrix, $P \in \mathbb{R}^{N\times d}$, where $N$ is the *maximum length of any sequence* that your model will be able to process.

We then simply add embedded representation of the position of a word to its word embedding:

$$\tilde{x}_i = P_i + x_i \tag{12}$$

and perform self-attention as we otherwise would. Now, the self-attention operation can use the embedding $P_i$ to look at the word at position $i$ differently than if that word were at position $j$. This is done, e.g., in the BERT paper (which we go over later in the course.)

### Position representation vectors through sinusoids

A unique encoding based on sine and cosine functions is added to each input embedding, allowing the model to differentiate positions while capturing relative distances. The encoding for each position *pos* and dimension $i$ is calculated as follows:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$
$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$

where $d_{\text{model}}$ is the embedding dimensionality. Even indices use the sine function, and odd indices use the cosine function, producing unique patterns for each position. This approach allows the model to capture both content and positional information, making it possible to learn relationships based on order without relying on sequential models like RNNs.

## The Intuition

You may wonder how this combination of sine and cosine functions can represent position or order. The concept is quite simple. Suppose you want to represent a number in binary format. The binary representation might look as follows:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 : | 0 | 0 | 0 | 0 | 8 : | 1 | 0 | 0 | 0 |
| 1 : | 0 | 0 | 0 | 1 | 9 : | 1 | 0 | 0 | 1 |
| 2 : | 0 | 0 | 1 | 0 | 10 : | 1 | 0 | 1 | 0 |
| 3 : | 0 | 0 | 1 | 1 | 11 : | 1 | 0 | 1 | 1 |
| 4 : | 0 | 1 | 0 | 0 | 12 : | 1 | 1 | 0 | 0 |
| 5 : | 0 | 1 | 0 | 1 | 13 : | 1 | 1 | 0 | 1 |
| 6 : | 0 | 1 | 1 | 0 | 14 : | 1 | 1 | 1 | 0 |
| 7 : | 0 | 1 | 1 | 1 | 15 : | 1 | 1 | 1 | 1 |

Figure 12: Binary Numbers

You can observe the rate of change between the different bits. The least significant bit (LSB) alternates with every number, the second-lowest bit changes every two numbers, and so on.
Using binary values, however, would be inefficient in the continuous world of floating-point numbers. Instead, we use their continuous counterparts—sinusoidal functions. These functions serve a similar purpose to alternating bits. By decreasing their frequencies, we can smoothly transition from "fast-changing" components (analogous to the red bits) to "slower-changing" components (analogous to the orange bits).

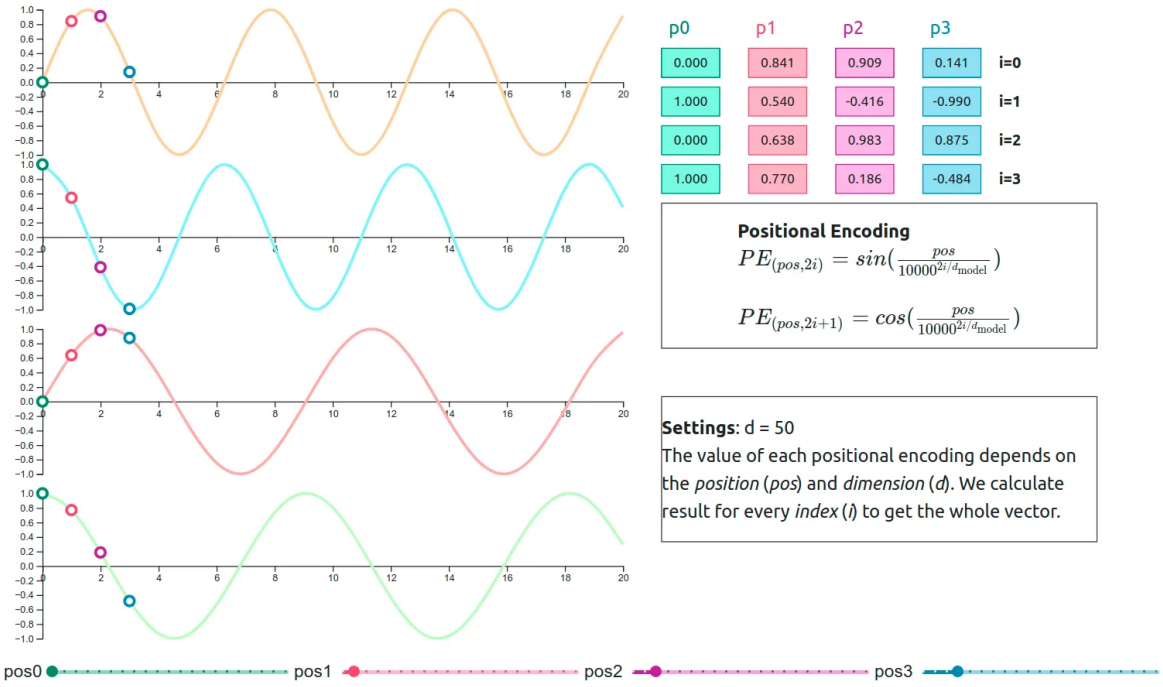


Figure 13: Positional encoding

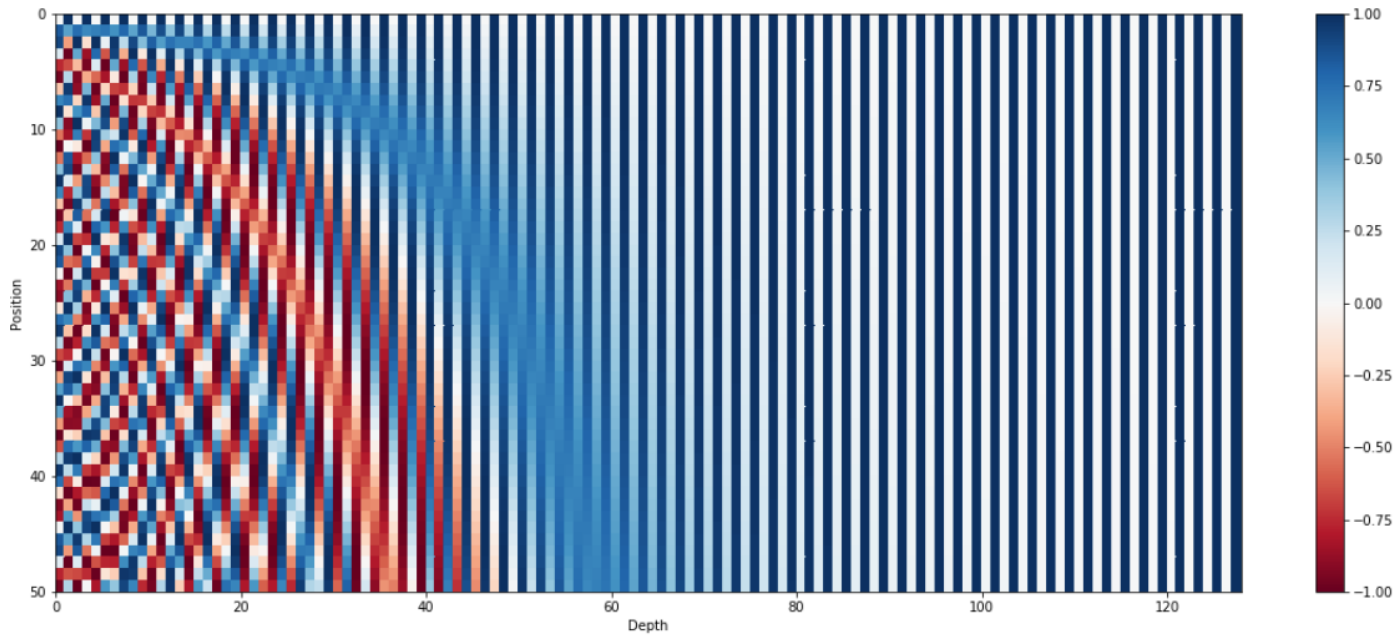


Figure 14: Sinusoidal position encoding

This image visualizes the *sinusoidal positional encodings* for a sequence of **50 words**, each represented by a **128-dimensional positional encoding vector**. Each position (or word) has a unique encoding vector, which combines sine and cosine functions at various frequencies across the 128 dimensions.

**Color Representation:**

- **Colors** represent the values of the encoding:
  - **Red** indicates negative values.
  - **Blue** represents positive values.
  - **White** corresponds to values close to zero.
- This color scheme helps to show the oscillation of values between positive and negative across the encoding dimensions (depth).

**Frequency Patterns:**

- Each dimension (depth) oscillates at a specific frequency.
  - **Lower depths** (closer to the left side) have higher frequencies, meaning the values alternate rapidly between positive and negative across positions. This captures fine-grained positional differences.
  - **Higher depths** (closer to the right) have lower frequencies, meaning they change more slowly across positions, capturing broader, more general positional information.

**Example: Breaking Down Row 10 (Position 10)**

Suppose we focus on the encoding for **Position 10** (the 10th word in the sequence):

- The encoding vector for Position 10 has 128 dimensions, with each dimension capturing a unique frequency component.
- In lower dimensions (left side of the row), we see rapid shifts between red and blue. This indicates high-frequency components, where sine and cosine values alternate quickly, distinguishing nearby positions from each other.

- As we move to higher dimensions (towards the right side), the colors shift more gradually, showing a mix of blue and white with occasional red. These are lower-frequency components that change slowly across positions, encoding long-term positional information.
- Toward the end of the row, dimensions show nearly constant blue values, indicating very low frequencies that add a broad positional context across the sequence.

## Position representation through changing $a$ directly

Instead of changing the input representation, another thing we can do is change the form of self-attention to have a built-in notion of position. One intuition is that all else held equal, self-attention should look at "nearby" words more than "far" words. Attention with Linear Biases is one implementation of this idea. One implementation of this would be as follows:

$$\alpha_i = \text{softmax}\left(k_{1:n}q_i + [-i, \ldots, -1, 0, -1, \ldots, -n]\right), \tag{13}$$

where $k_{1:n}q_i \in \mathbb{R}^n$ are the original attention scores, and the bias we add makes attention focus more on nearby words than far away words, all else held equal. In some sense, it's odd that this works; but interesting!

## Elementwise nonlinearity

Imagine if we were to stack self-attention layers. Would this be sufficient for a replacement for stacked LSTM layers? Intuitively, there's one thing that's missing: the elementwise nonlinearities that we've come to expect in standard deep learning architectures. In fact, if we stack two self-attention layers, we get something that looks a lot like a single self-attention layer:

$$o_i = \sum_{j=1}^{n} \alpha_{ij}^{(2)} \left( \sum_{k=1}^{n} \alpha_{ik}^{(1)} x_k^{(1)} \right) \tag{14}$$

$$= \sum_{k=1}^{n} \left( \alpha_{ik}^{(1)} \sum_{j=1}^{n} \alpha_{ij}^{(2)} \right) x_k^{(1)} \tag{15}$$

$$= \sum_{k=1}^{n} \alpha_{ij}^{*} V^{*} x_k, \tag{16}$$

where $\alpha_{ij}^{*} = \left( \alpha_{ik}^{(1)} \sum_{j=1}^{n} \alpha_{ij}^{(2)} \right)$, and $V^{*} = V^{(2)}V^{(1)}$. So, this is just a linear combination of a linear transformation of the input, much like a single layer of self-attention! Is this good enough?

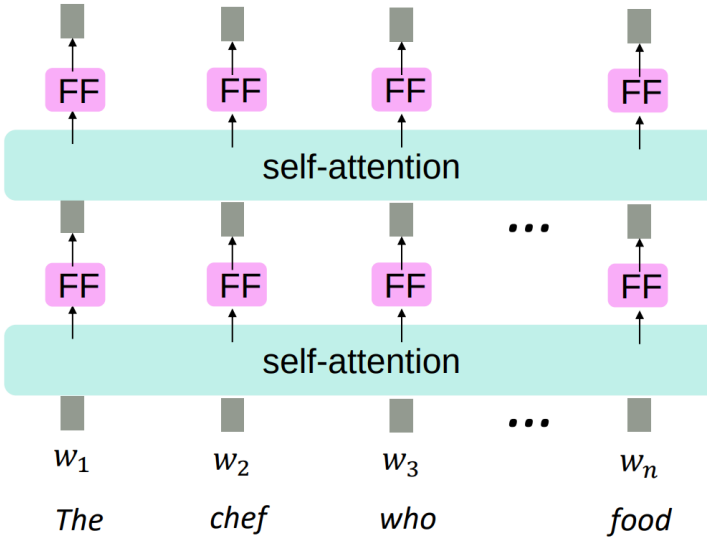


Figure 15: Adding Non-linearity

In practice, after a layer of self-attention, it's common to apply feed-forward network independently to each word representation:

$$h_{\text{FF}} = W_2 \,\text{ReLU}(W_1 h_{\text{self-attention}} + b_1) + b_2, \tag{17}$$

where often, $W_1 \in \mathbb{R}^{5d \times d}$, and $W_2 \in \mathbb{R}^{d \times 5d}$. That is, the hidden dimension of the feed-forward network is substantially larger than the hidden dimension of the network, $d$—this is done because this matrix multiply is an efficiently parallelizable operation, so it's an efficient place to put a lot of computation and parameters.

## Future masking

When performing language modeling like we've seen in this course (often called autoregressive modeling), we predict a word given all words so far:

$$w_t \sim \text{softmax}(f(w_{1:t-1})). \tag{18}$$

where $f$ is function to map a sequence to a vector in $\mathbb{R}^{|\mathcal{V}|}$.

One crucial aspect of this process is that we can't look at the future when predicting it—otherwise the problem becomes trivial. This idea is built-in to unidirectional RNNs. If we want to use an RNN for the function $f$, we can use the hidden state for word $w_{t-1}$:

$$w_t \sim \text{softmax}(h_{t-1}E) \tag{19}$$

$$h_{t-1} = \sigma(W h_{t-2} + U x_{t-1}), \tag{20}$$

and by the rollout of the RNN, we haven't looked at the future. (In this case, the future is all the words $w_t, \ldots, w_n$.)
In a Transformer, there's nothing explicit in the self-attention weight $\alpha$ that says not to look at indices $j > i$ when representing token $i$. In practice, we enforce this constraint simply adding a large negative constant to the input to the softmax (or equivalently, setting $\alpha_{ij} = 0$ where $j > i$):

$$\alpha_{ij,\text{masked}} = \begin{cases} \alpha_{ij} & j \leq i \\ 0 & \text{otherwise} \end{cases} \tag{21}$$

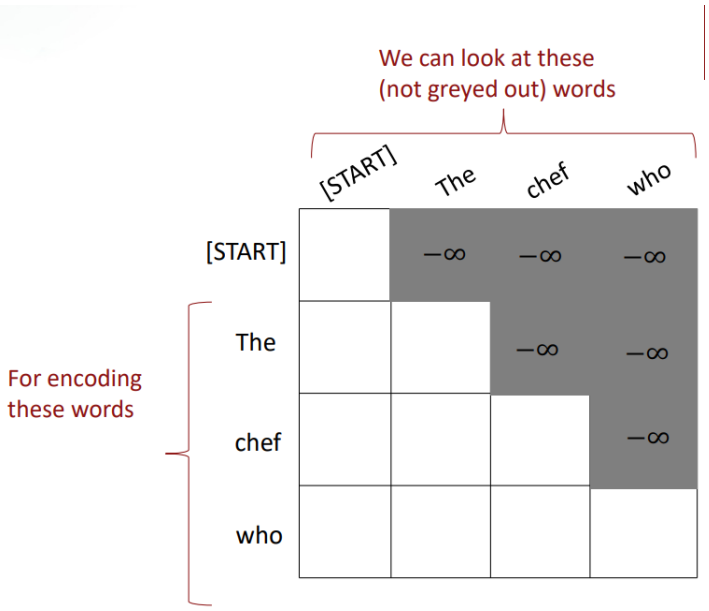


Figure 16: Masking

## Summary of a minimal self-attention architecture

Our minimal self-attention architecture has (1) the self-attention operation, (2) position representations, (3) elementwise nonlinearities, and (4) future masking (in the context of language modeling.)
Intuitively, these are the biggest components to understand. However, as of 2023, by far the most-used architecture in NLP is called the *Transformer*, introduced by, and it contains a number of components that end up being quite important. So now we'll get into the details of that architecture.

## Multi-Head Attention

Multi-head attention allows the model to focus on different parts of the input simultaneously, capturing various types of relationships between words:

$$\text{MultiHead}(V, Q, K) = W^O \cdot \text{concat}(\text{head}_1, \text{head}_2, \ldots, \text{head}_h) \tag{1}$$

Where each head produces:

$$K = W_K^{1^T} x, \quad Q = W_Q^{1^T} x, \quad V = W_V^{1^T} x \tag{2}$$

The output of each head is computed as:

$$z = V \cdot \text{softmax}\left(\frac{Q^T K}{\sqrt{p}}\right) \tag{3}$$

Here, $K$, $Q$, and $V$ represent the Keys, Queries, and Values respectively, which are different projections of the input. The $\sqrt{p}$ term is a scaling factor to counteract the effect of large dot products in high dimensions.

This enhancement improves the performance of the attention layer in two key ways:

- **Expanded Focus on Different Positions:** Multi-headed attention increases the model's capacity to attend to multiple positions in the input simultaneously. In standard attention, each output vector (e.g., $z_1$) contains a weighted mix of all other encodings, but may still be dominated by the current word's own features. For example, in a sentence like "The animal didn't cross the street because it was too tired," it is crucial for the model to understand which word "it" refers to. Multi-headed attention enables this by allowing the model to attend to multiple relevant parts of the sentence.

- **Multiple Representation Subspaces:** Multi-headed attention provides the attention layer with multiple "representation subspaces." Rather than having a single set of Query, Key, and Value weight matrices, multi-headed attention uses multiple sets (eight in the original Transformer model) for each encoder/decoder layer. Each set is randomly initialized and then trained independently, allowing the model to project input embeddings (or representations from lower encoder/decoder layers) into different subspaces. After training, each head captures distinct aspects of the input, enriching the model's ability to learn varied linguistic patterns.

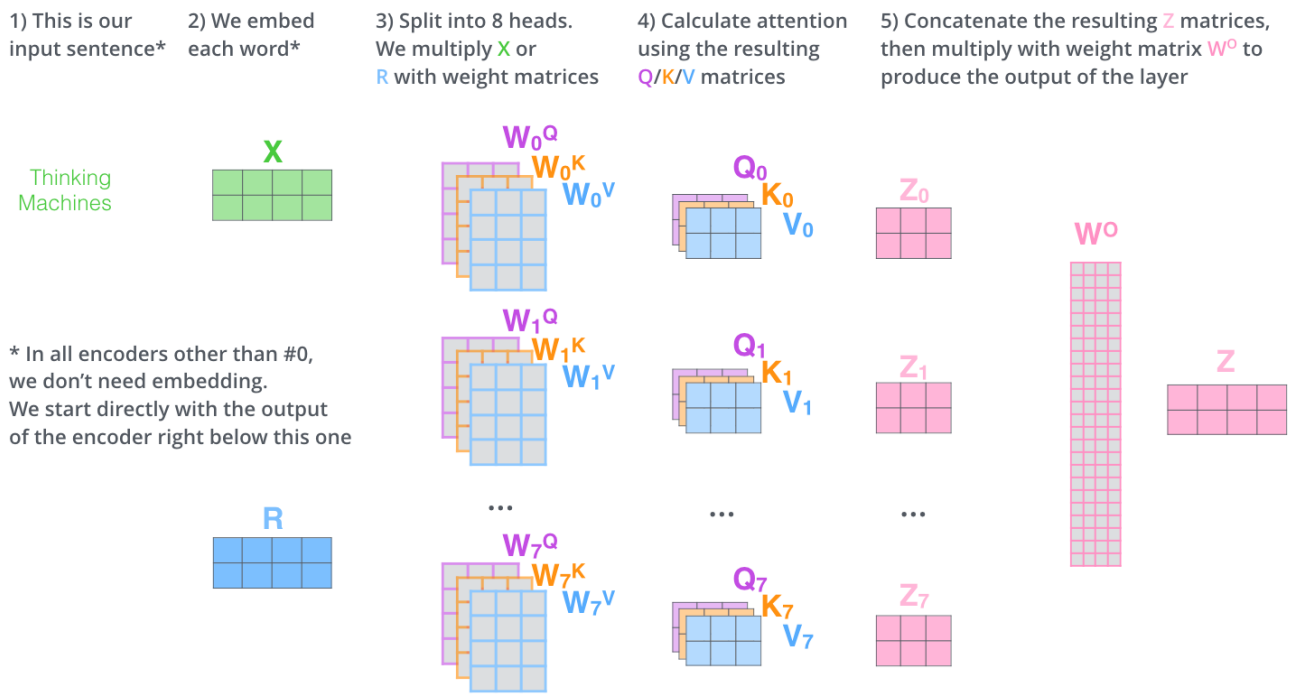


Figure 17: Multihead Attention

If we perform the same self-attention calculation as outlined above, but with eight different sets of weight matrices, we obtain eight distinct $Z$ matrices.
This presents a challenge: the feed-forward layer is designed to accept a single matrix (a single vector per word), not eight separate matrices. Therefore, we need a way to condense these eight matrices into one.
To achieve this, we concatenate the matrices and then multiply them by an additional weight matrix $W_O$.

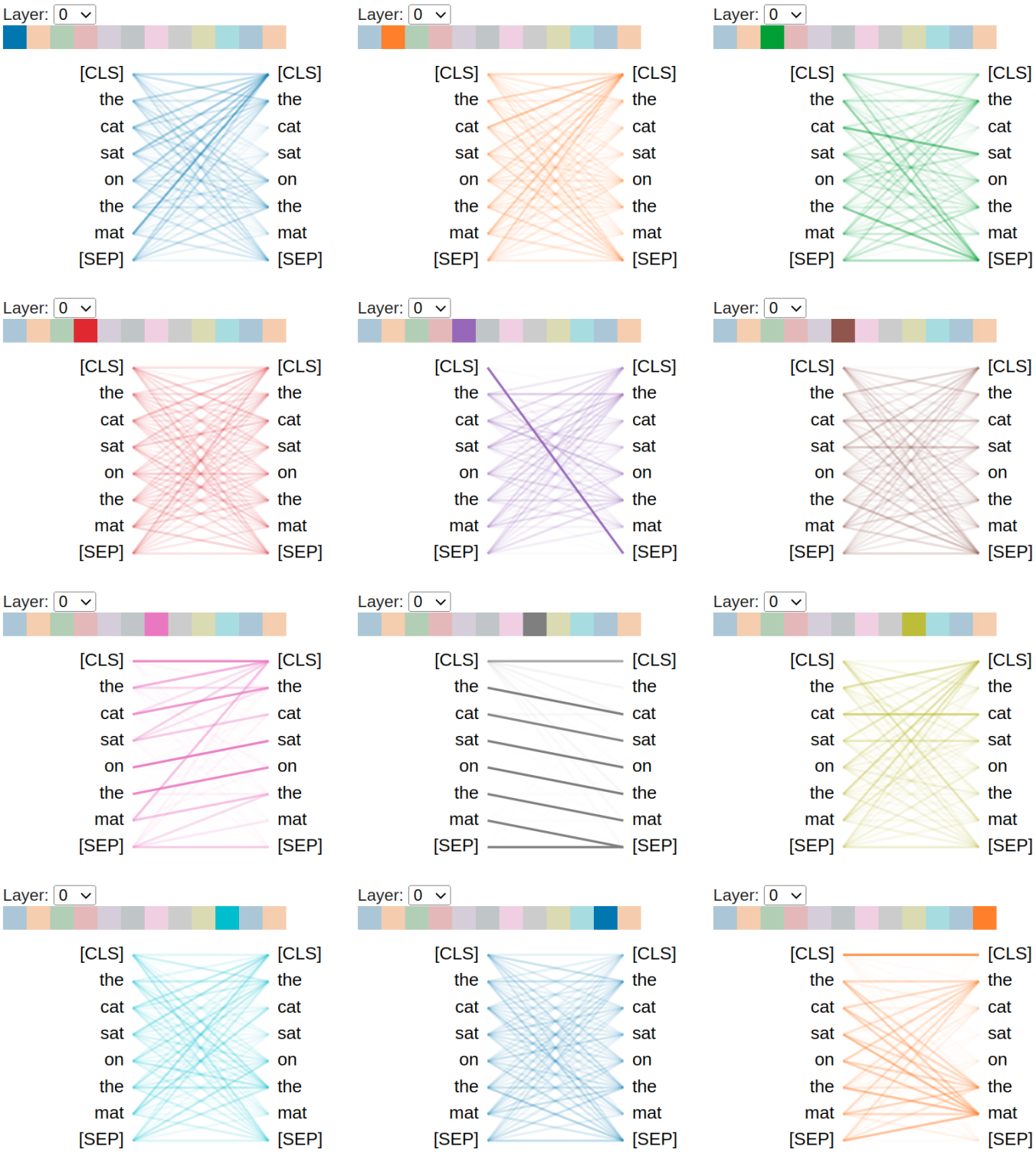


Figure 18: Visualization of attention heads across Multi-Head attention model (12 heads). Each sub-image represents the attention patterns from a distinct attention head, showing how the model distributes its focus across tokens within and between two sentences. Darker and thicker lines indicate stronger attention weights, meaning that particular words or tokens are more closely linked by that attention head. Different colors denote different attention heads, each capturing unique relationships, such as direct token alignment, cross-sentence references, or coreference patterns. Together, these attention heads provide a comprehensive view of how the model interprets and connects information across the sentences.

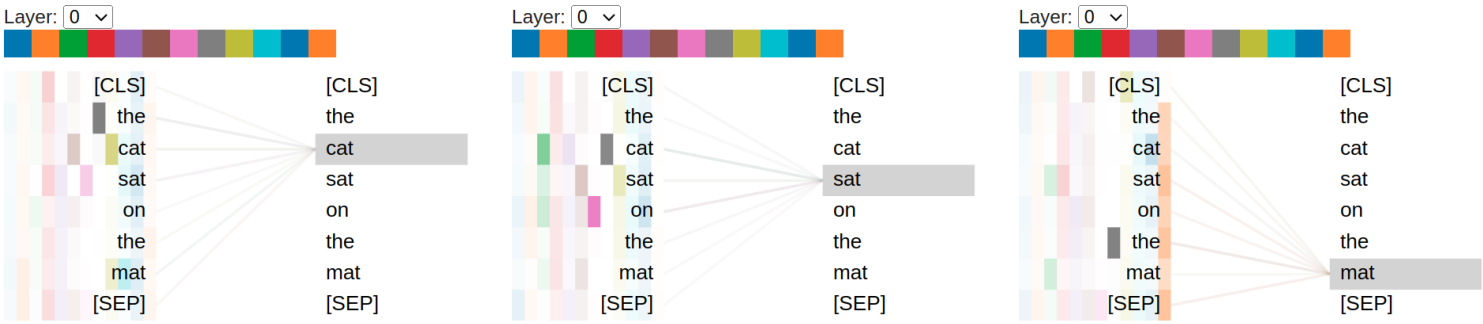


Figure 19: Combined attention patterns across all 12 heads for the words "cat," "sat," and "mat" in the sentence pair "The cat sat on the mat." Each panel shows the cumulative attention from all heads, illustrating how the model gathers contextual information for each focal word by attending to relevant words across both sentences. The left panel focuses on "cat," the middle on "sat," and the right on "mat," capturing a broad understanding of entity, action, and location through multi-headed attention.

## Sequence-Tensor Form

To understand why each head's output has a reduced dimension, it's helpful to look closely at how multi-head self-attention is implemented in code. In practice, *multi-head self-attention is no more expensive than single-head attention* due to the low-rank nature of the transformations applied.
For a single head, let $x_{1:n}$ be a matrix in $\mathbb{R}^{n\times d}$. We compute our value vectors as $x_{1:n}W_V$, and similarly, our keys and queries as $x_{1:n}W_K$ and $x_{1:n}W_Q$, all of which are matrices in $\mathbb{R}^{n\times d}$. The weights for self-attention can then be computed using matrix operations:

$$\alpha = \text{softmax}(x_{1:n}W_Q W_K^T x_{1:n}^T) \in \mathbb{R}^{n\times n}. \tag{25}$$

We can then perform the self-attention operation for all $x_{1:n}$ as:

$$z_{1:n} = \text{softmax}(x_{1:n}W_Q W_K^T x_{1:n}^T)W_V \in \mathbb{R}^{n\times d}. \tag{26}$$

Here's a diagram showing the matrix operations:

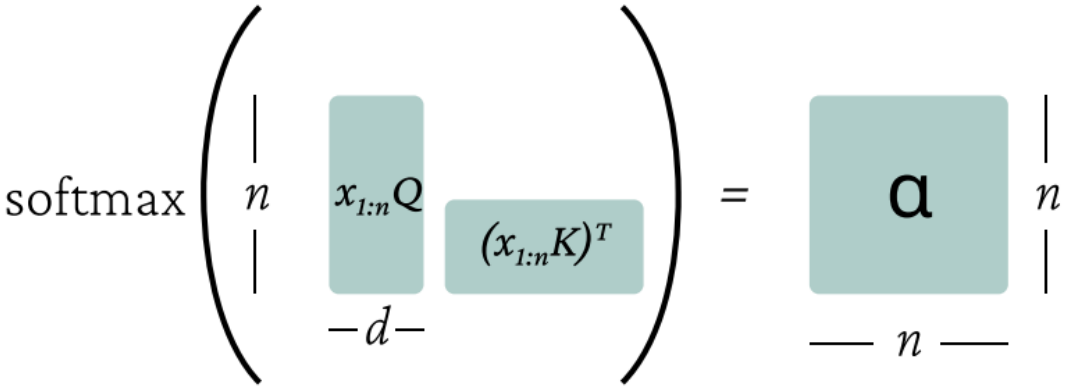


Figure 20: Single-head self-attention matrix operations.

In multi-head self-attention, we first reshape $x_{1:n}W_Q$, $x_{1:n}W_K$, and $x_{1:n}W_V$ into matrices of shape $\mathbb{R}^{n\times d/k}$, effectively splitting the model dimensionality $d$ across $k$ heads, each with dimensionality $d/k$. We then transpose the matrices to shape

$\mathbb{R}^{k \times n \times d/k}$, representing $k$ sequences of length $n$ and dimension $d/k$. This allows us to compute the softmax operation in parallel across heads, treating the heads dimension as an additional "batch-like" axis (note that in practice, there is also a separate batch axis).
This approach ensures that the total computation, aside from the final linear transformation to combine the heads, remains equivalent to that of single-head attention, but is distributed across multiple low-rank heads. The following diagram illustrates how the multi-head operation is conceptually similar to the single-head operation:

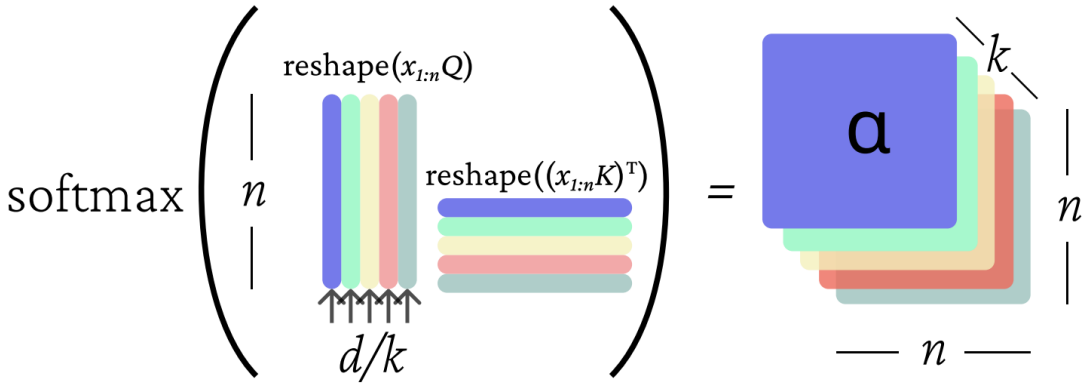


Figure 21: Multi-head self-attention matrix operations, demonstrating parallel attention across heads.

## Transformers

Self-attention is just one part of a larger *transformer* mechanism. This consists of a multi-head self-attention unit (which allows the word representations to interact with each other) followed by a fully connected network $\texttt{mlp}[\mathbf{x}_n]$ (that operates separately on each word). Both units are residual networks (i.e., their output is added back to the original input). In addition, it is typical to add a LayerNorm operation after both the self-attention and fully connected networks. This is similar to BatchNorm but uses statistics across the tokens within a single input sequence to perform the normalization. The complete layer can be described by the following series of operations (figure 21):

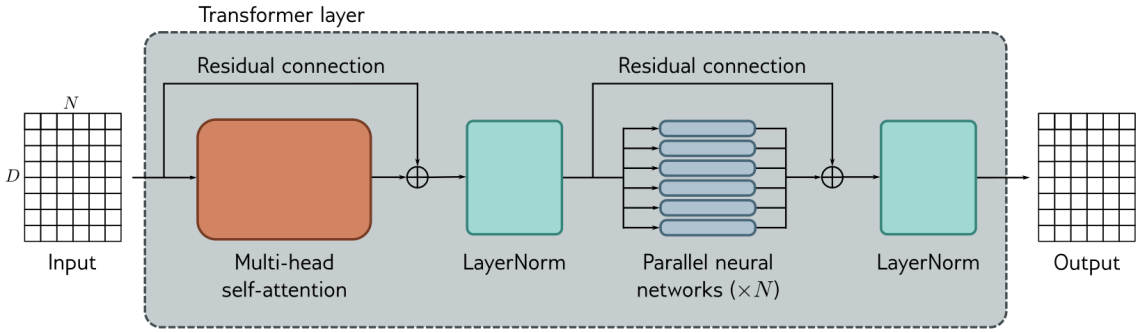


Figure 22: The transformer. The input consists of a $D \times N$ matrix containing the $D$-dimensional word embeddings for each of the $N$ input tokens. The output is a matrix of the same size. The transformer consists of a series of operations. First, there is a multi-head attention block, allowing the word embeddings to interact with one another. This forms the processing of a residual block, so the inputs are added back to the output. Second, a LayerNorm operation is applied. Third, there is a second residual layer where the same fully connected neural network is applied separately to each of the $N$ word representations (columns). Finally, LayerNorm is applied again.

$$\mathbf{X} \leftarrow \mathbf{X} + \text{MhSa}[\mathbf{X}]$$
$$\mathbf{X} \leftarrow \text{LayerNorm}[\mathbf{X}]$$

$$\mathbf{x}_n \leftarrow \mathbf{x}_n + \text{mlp}[\mathbf{x}_n] \quad \forall n \in \{1, \ldots, N\}$$
$$\mathbf{X} \leftarrow \text{LayerNorm}[\mathbf{X}],$$

where the column vectors $\mathbf{x}_n$ are separately taken from the full data matrix $\mathbf{X}$. In a real network, the data passes through a series of these transformers.

### Layer Norm

One important learning aid in Transformers is *layer normalization*. The intuition behind layer normalization is to reduce uninformative variation in the activations at a layer, providing a more stable input to the next layer. Further research indicates that this may be most beneficial not in normalizing the forward pass, but rather in improving gradients during the backward pass.
To achieve this, layer normalization:

- (1) computes statistics across the activations at a layer to estimate the mean and variance of the activations.
- (2) normalizes the activations with respect to those estimates.
- (3) optionally learns (as parameters) an elementwise additive bias and multiplicative gain to soften or de-normalize the activations in a predictable way.

The third part is often omitted, as it may not be crucial and can sometimes even be detrimental to performance.
A key question when understanding how layer normalization affects a network is, "computing statistics over what?" **What constitutes a layer?** In Transformers, statistics are computed independently for each token position (and for each example in the batch) and are shared across the $d$ hidden dimensions. This means that the statistics for the token at position $i$ won't affect the token at position $j \neq i$.
Thus, we compute the statistics for a single token position $i \in \{1, \ldots, n\}$ as

$$\mu_i = \frac{1}{d}\sum_{j=1}^{d} x_{ij}, \quad \sigma_i = \sqrt{\frac{1}{d}\sum_{j=1}^{d}(x_{ij} - \mu_i)^2}, \tag{27}$$

where $\mu_i$ and $\sigma_i$ are scalars, and we compute the layer normalization as

$$\text{LN}(x_i) = \frac{x_i - \mu_i}{\sigma_i}, \tag{28}$$

with $\mu_i$ and $\sigma_i$ broadcasted across the $d$ dimensions of $x_i$. Layer normalization is a valuable tool to have in your deep learning toolbox more broadly.

### Residual Connections

Residual connections add the *input* of a layer to the *output* of that layer:

$$x_{\text{residual}}(x_{1:n}) = f(x_{1:n}) + x_{1:n}, \tag{29}$$

where (1) the gradient flow through the identity function is excellent (the local gradient is 1 everywhere), enabling the learning of much deeper networks, and (2) it is easier to learn the difference of a function from the identity function than to learn the function from scratch. Despite their simplicity, residual connections are essential in deep learning, not just in Transformers!

### Add & Norm

In Transformer diagrams, including Figure 22, layer normalization and residual connections are often combined in a single visual block labeled *Add & Norm*. Such a layer might look like:

$$x_{\text{pre-norm}} = f(\text{LN}(x)) + x, \tag{30}$$

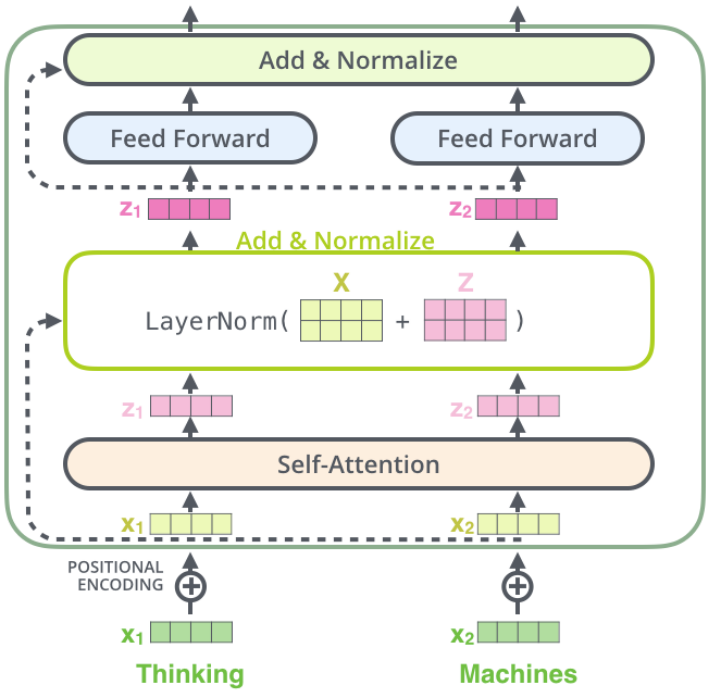


Figure 23: Add & Normalize

where $f$ is either a feed-forward operation or a self-attention operation (known as *pre-normalization*), or like:

$$x_{\text{post-norm}} = \text{LN}(f(x) + x), \tag{31}$$

known as *post-normalization*. It has been found that the gradients of *pre-normalization* are better at initialization, leading to faster training.

### Attention Logit Scaling

Another technique introduced is *scaled dot-product attention*. The dot-product aspect arises from computing the dot products $q_i^T k_j$. The scaling intuition is that as the dimensionality $d$ of the vectors grows large, the dot product of even random vectors (e.g., at initialization) tends to grow roughly as $\sqrt{d}$. To counteract this, we normalize the dot products by $\sqrt{d}$ to prevent scaling:

$$\alpha = \text{softmax}\left(\frac{x_{1:n} Q K^T x_{1:n}^T}{\sqrt{d}}\right) \in \mathbb{R}^{n \times n}. \tag{32}$$

## Types of Transformers

As described in the previous sections, the **Transformer** is an architecture based on self-attention, consisting of stacked *blocks*, each containing self-attention and feed-forward layers, along with components such as *multi-head self-attention*,

*layer normalization*, *residual connections*, and *attention scaling*. In the following sections, we will discuss how these components are combined to form the complete Transformer architecture.
There are three types of transformer models.

- **Encoder:** An encoder transforms the text embeddings into a representation that can support a variety of tasks.
- **Decoder:**A decoder predicts the next token to continue the input text.
- **Encoder - Decoder:** Encoder-decoders are used in sequence-to-sequence tasks, where one text string is converted into another (e.g., machine translation).

## Transformer Encoder

A Transformer Encoder takes a single sequence $w_{1:n}$ and performs no future masking. It embeds the sequence with $E$ to make $x_{1:n}$, adds the position representation, and then applies a stack of independently parameterized *Encoder Blocks*, each of which consisting of (1) multi-head attention and Add & Norm, and (2) feed-forward and Add & Norm. So, the output of each Block is the input to the next.
In the case that one wants probabilities out of the tokens of a Transformer Encoder (as in masked language modeling for BERT, which we'll cover later), one applies a linear transformation to the output space followed by a softmax.
*Uses of the Transformer Encoder.* A Transformer Encoder is great in contexts where you aren't trying to generate text autoregressively (there's no masking in the encoder so each position index can see the whole sequence), and want strong representations for the whole sequence (again, possible because even the first token can see the whole future of the sequence when building its representation).

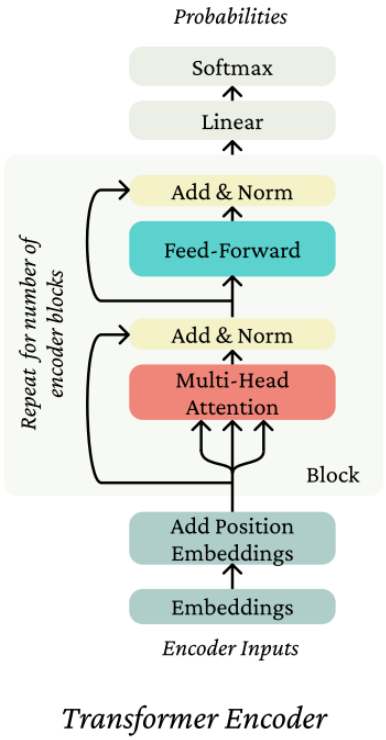


Figure 24: Transformer Encoder

# Transformer Decoder

To build a Transformer autoregressive language model which can generate texts, one uses a Transformer Decoder. These differ from Transformer Encoders simply by using future masking at each application of self-attention. This ensures that the informational constraint (no cheating by looking at the future!) holds throughout the architecture. Famous examples of this are GPT-2, GPT-3 and BLOOM .
**Decoder with Masked Self-Attention:** The decoder uses masked self-attention (Refer to **Future masking** section) to ensure each word only attends to previous words, not future ones. For example, in the sentence **"I am a ___"**, the model only attends to the words before the blank to predict the next word. This is crucial for maintaining the autoregressive property of language generation.

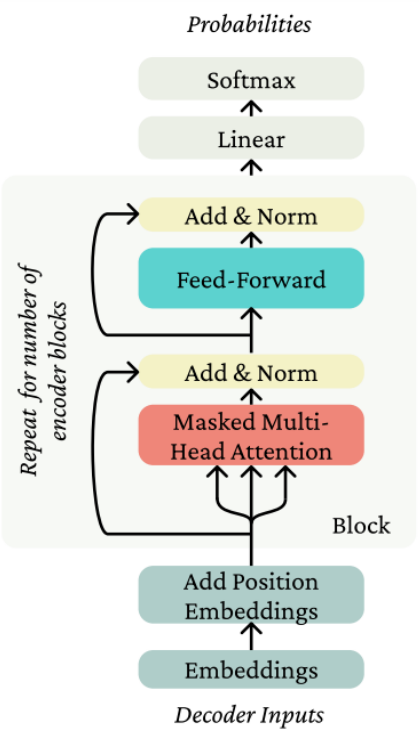


Figure 25: Transformer Decoder

# Transformer Encoder-Decoder

A Transformer encoder-decoder takes as input two sequences.

- The first sequence $x_{1:n}$ is passed through a Transformer Encoder to build contextual representations.
- The second sequence $y_{1:m}$ is encoded through a modified Transformer Decoder architecture in which *cross-attention* (which we haven't yet defined!) is applied from the encoded representation of $y_{1:m}$ to the output of the Encoder.

So, let's take a quick detour to discuss cross-attention; it's not too different from what we've already seen.

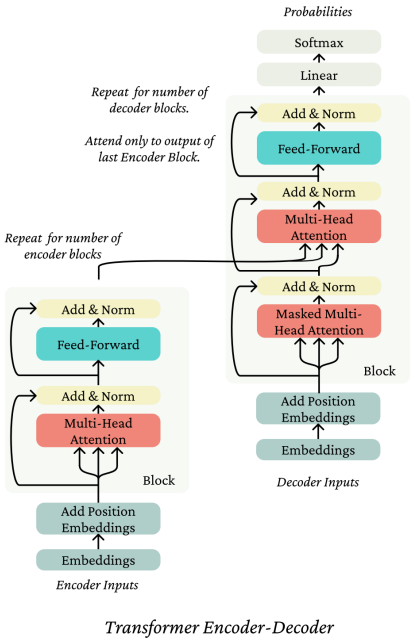


Figure 26: Encoder - Decoder

## Cross-Attention

Cross-attention uses one sequence to define the keys and values of self-attention, and another sequence to define the queries. You might think, hey wait, isn't that just what attention always was before we got into this self-attention business? Yeah, pretty much. So if

$$h_{1:n}^{(x)} = \text{TransformerEncoder}(w_{1:n}^{(x)})$$

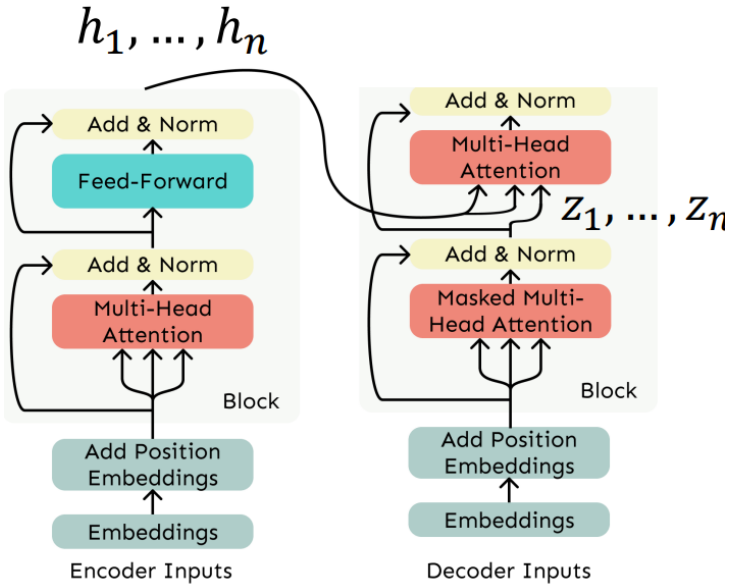


Figure 27: Cross attention

and we have some intermediate representation $h_{1:m}^{(y)}$ of sequence $y_{1:m}$, then we let the queries come from the decoder (the $h^{(y)}$ sequence) while the keys and values come from the encoder:

$$\begin{aligned} q_i &= Qh_i^{(y)} \quad i \in \{1, \ldots, m\} \\ k_j &= Kh_j^{(x)} \quad j \in \{1, \ldots, n\} \\ v_j &= Vh_j^{(x)} \quad j \in \{1, \ldots, n\} \end{aligned}$$

and compute the attention on $q, k, v$ as we defined for self-attention. Note in Figure 6 that in the Transformer Encoder-Decoder, cross-attention always applies to the output of the Transformer encoder.

## Uses of the encoder-decoder / Encoder - Decoder Attention

An encoder-decoder is used when we'd like bidirectional context on something (like an article to summarize) to build strong representations (i.e., each token can attend to all other tokens), but then generate an output according to an autoregressive decomposition as we can with a decoder. While such an architecture has been found to provide better performance than decoder-only models at modest scale, it involves splitting parameters between the encoder and decoder, and most of the largest Transformers are decoder-only.

## The Final Linear and Softmax Layer

The decoder stack produces a vector of floating-point numbers as its output. How is this transformed into a word? This is achieved through the final *Linear* layer, followed by a *Softmax* layer.

The Linear layer is a fully connected neural network that projects the vector output from the decoder stack into a much larger vector, known as the *logits vector*.

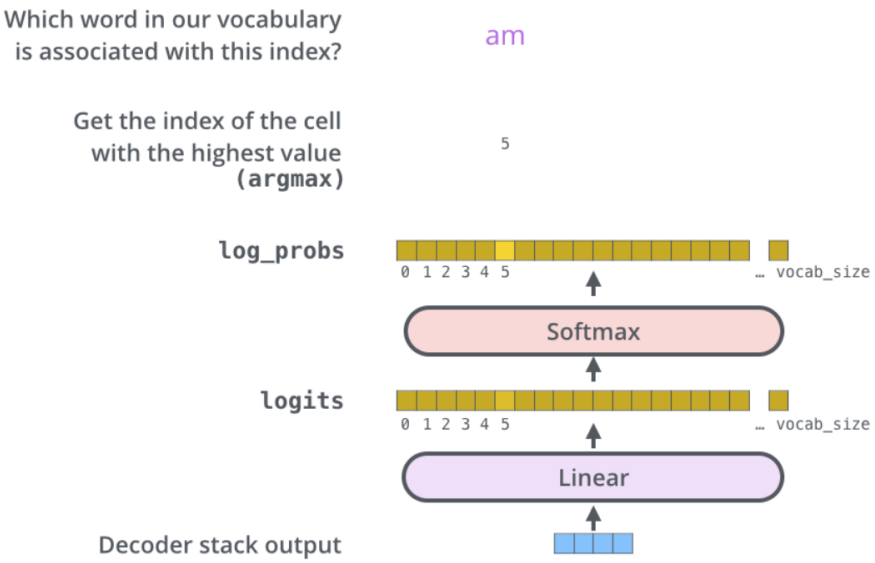


Figure 28: Final layer and Softmax layer

Suppose our model has learned an "output vocabulary" of 10,000 unique English words from its training dataset. In this case, the logits vector would have 10,000 cells—each cell representing the score of a unique word. This logits vector is the output of the model after the Linear layer.
The Softmax layer then converts these scores into probabilities (all positive and summing to 1). The word corresponding to the cell with the highest probability is selected as the output for this time step.

# Decoder Process with Cross-Attention

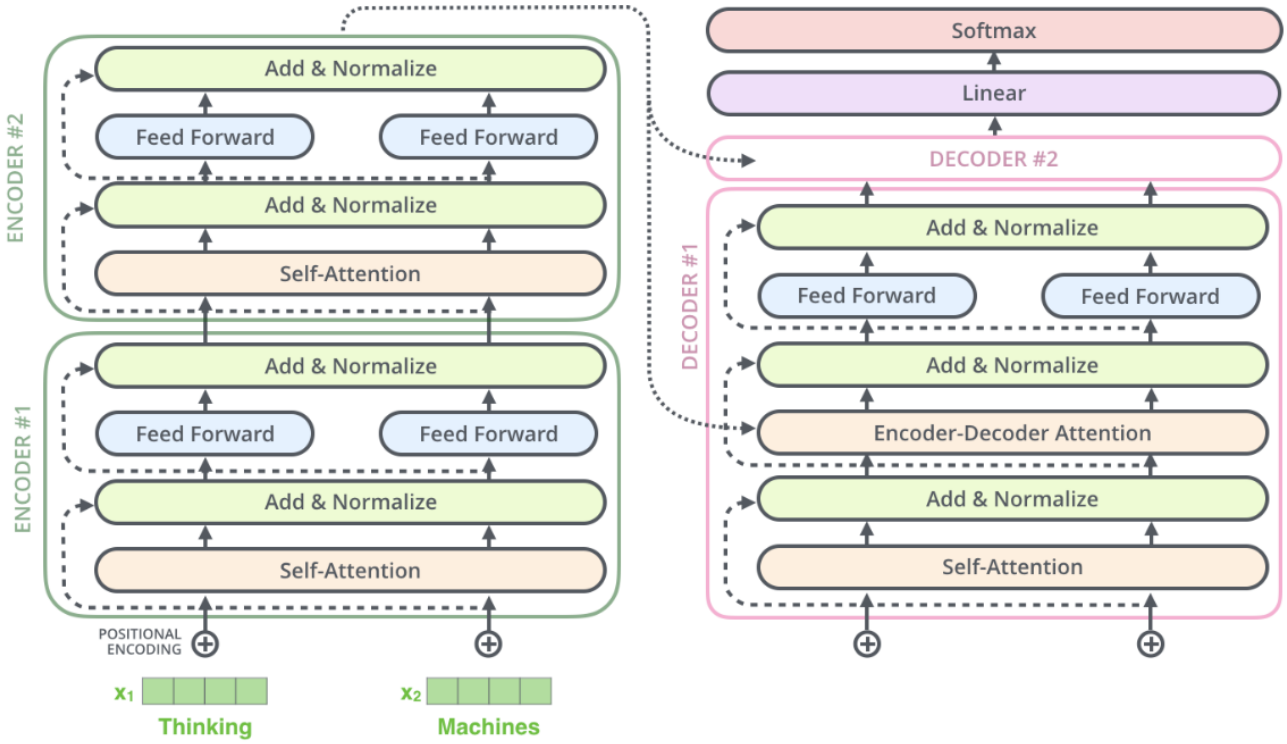


Figure 29: Encoder-Decoder breakdown

The Transformer architecture uses an encoder-decoder structure where the decoder leverages self-attention, cross-attention, and feed-forward layers to generate outputs. Here's a breakdown of the decoder's process:

- **Step 1: Self-Attention in Decoder**
  Each decoder block starts by performing *self-attention* on the target sequence. This self-attention layer allows the decoder to attend to earlier tokens in the output sequence, building representations based on previously generated words or tokens. Importantly, a *causal mask* is applied to prevent attending to future tokens, maintaining the autoregressive nature of text generation.

- **Step 2: Cross-Attention (Encoder-Decoder Attention)**
  After self-attention, the decoder block performs *encoder-decoder attention*, also known as *cross-attention*. Here, each token in the decoder attends to the entire output of the encoder. This enables the decoder to incorporate contextual information from the source sequence, allowing it to "look back" at the input while generating each token. The encoder's final representations act as keys and values for this attention mechanism, while the decoder's output from the previous self-attention layer serves as the query.

- **Step 3: Feed-Forward Layer**
  The cross-attention output is then passed through a *feed-forward layer* to further process and transform the representation. This layer, which is independently parameterized in each decoder block, helps refine the token representations and adds non-linearity.

- **Step 4: Add & Normalize**
  After each major operation (self-attention, cross-attention, and feed-forward), *residual connections* (adding the input to the output of each layer) and *layer normalization* are applied. These ensure stable gradients and allow the network to retain information from earlier layers.

- **Step 5: Stacked Decoder Layers**
  The output from each decoder block becomes the input for the next block. The decoder is typically composed of multiple such blocks stacked together, allowing it to build complex representations that incorporate both source and target sequence context.

- **Step 6: Final Linear and Softmax Layers**
  The final output of the stacked decoder blocks is passed through a linear layer and then a softmax layer, producing a probability distribution over the vocabulary for each token position. This distribution is used to predict the next word in the sequence.

## Encoder-Decoder Example

Let's go over an example of how a Transformer model translates the French phrase "Je suis étudiant" into English as "I am a student." The process involves step-by-step encoding and decoding to generate each word in the target sentence.

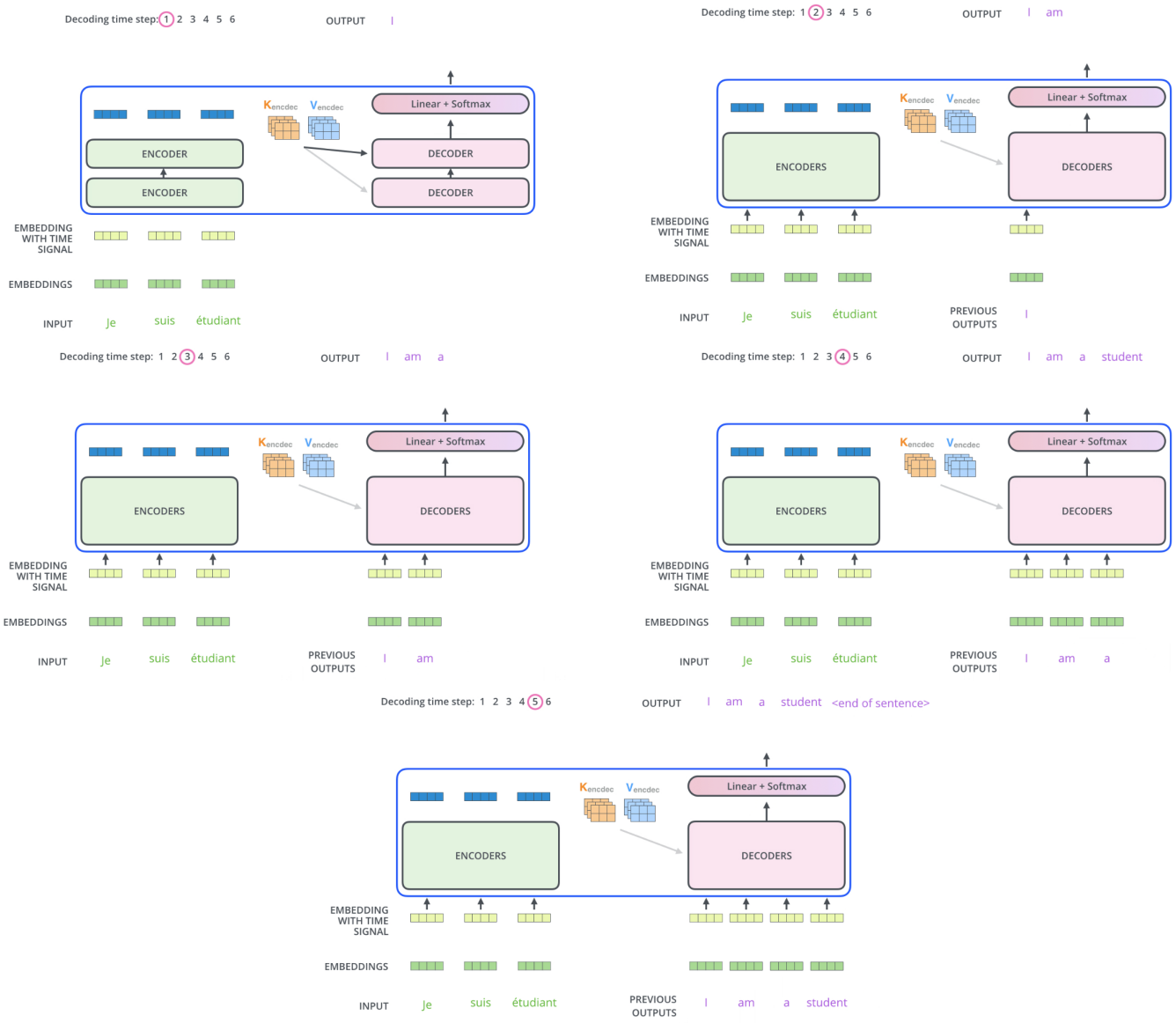


Figure 30: Visualization of encoder-decoder process

- **Encoding Phase**:
  - The input sentence "Je suis étudiant" is first tokenized and embedded. Each token is combined with a positional encoding to form the input embeddings.
  - These embeddings are passed through multiple encoder layers to produce contextualized representations for each word. These representations capture the meaning of each word in relation to the others in the sentence.

- **Decoding Step-by-Step**:
  - **Time Step 1**:
    * At the first decoding step, the decoder receives a special start-of-sequence token (<s>), which initiates the generation process.
    * The decoder attends to the encoder's output representations (using the keys and values from the encoder) to create a context-sensitive output for the first word.
    * After processing through the decoder layer, the output is passed through a linear layer followed by a softmax function, generating a probability distribution over the vocabulary. The most probable word, "I," is selected as the output.
  - **Time Step 2**:
    * In the second step, the decoder takes the previously generated word, "I," along with the encoded input.
    * It uses self-attention to incorporate context from its previous output and cross-attention to draw information from the encoder's output.
    * The word "am" is selected as the most probable next word after passing through the softmax layer.
  - **Time Step 3**:
    * The process repeats with the decoder inputting "I am" as context for predicting the next word.
    * The model selects "a" as the next word, using both self-attention on its previous outputs and cross-attention on the encoder's output.
  - **Time Step 4**:
    * The decoder now uses "I am a" to predict the next word.
    * The model attends to "étudiant" in the encoder's output and generates "student" as the next word, completing the main translation.
  - **Time Step 5**:
    * A special end-of-sequence token (<end>) is generated, signaling the end of the sentence.
- **Final Output**:
  - The final translated sentence, "I am a student," is complete, and the model stops generating further words after the end-of-sequence token.

# Drawbacks of Transformers

- **Quadratic Computational Complexity in Self-Attention**:

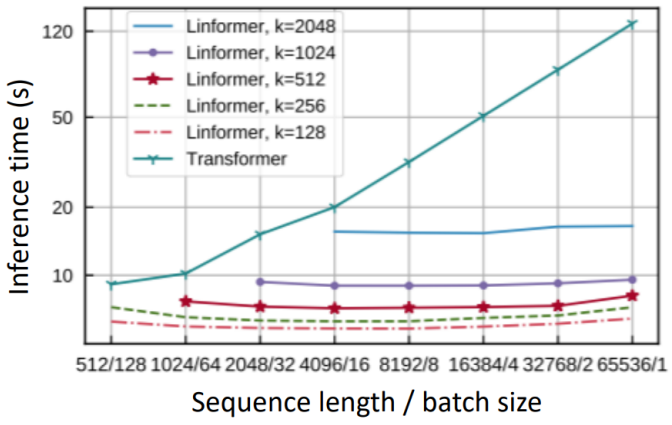


Figure 31: Quadratic Cost

  - In the current Transformer models, self-attention requires computing all pairs of interactions between tokens, resulting in computational costs that grow *quadratically* with the sequence length.

  - While recurrent models grew linearly with sequence length, Transformers require $O(n^2 d)$ operations, where $n$ is the sequence length and $d$ is the dimensionality.
  - For example, with a moderate sentence length $n = 30$, the computational cost can reach around 900 operations. With larger sequence lengths like $n = 50,000$, the cost becomes prohibitive, limiting the ability to work on very long documents.

- **Position Representations**:
  - Transformers rely on absolute positional encodings to represent the order of tokens. However, this method might be suboptimal for capturing complex dependencies.
  - Researchers have explored alternatives, such as:
    * *Relative Position Attention* (Shaw et al., 2018), which encodes relationships between tokens based on relative positions.
    * *Dependency Syntax-Based Position* (Wang et al., 2019), which integrates syntactic dependencies for better context understanding.

# Conclusion

Transformers and their attention mechanisms have revolutionized natural language processing by effectively addressing the challenges of long-range dependencies and parallelization. By combining multi-head attention, positional encoding, and masked self-attention in the decoder, transformers provide a powerful and flexible architecture for a wide range of language tasks.

The key innovations of transformers include:

- **Self-attention**: Allowing each word to attend to all other words in the sequence, capturing complex relationships.
- **Multi-head attention**: Enabling the model to focus on different aspects of the input simultaneously.
- **Positional encoding**: Providing sequence order information without recurrence or convolution.
- **Parallelization**: Allowing for efficient training and inference on modern hardware.

These features have made transformers the foundation for state-of-the-art models in NLP, such as BERT, GPT, and T5, and have even found applications in other domains like computer vision and speech processing. As research continues, we can expect further refinements and innovations building upon the transformer architecture.

# References


[1] Simon J.D. Prince. *Understanding Deep Learning*. MIT Press, 2023.

[2] John Hewitt. "Self-Attention & Transformers.".

[3] Alammar, J (2018). The Illustrated Transformer .

[4] https://towardsdatascience.com/understanding-positional-encoding-in-transformers-dc6bafc021ab