Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com

Noam Shazeer* Google Brain noam@google.com

Niki Parmar* Google Research nikip@google.com

Jakob Uszkoreit* Google Research usz@google.com

Llion Jones* Google Research llion@google.com

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

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com

pdf: *Attention Is All You Need*

How Can We Do Natural Language Processing (NLP) Without RNNs or LSTMs?

The question is asking whether it is possible to handle NLP tasks without using traditional architectures like **RNNs** or **LSTMs**. The answer is **yes**, and we can do this using **Transformer** models. Transformers rely entirely on **self-attention** mechanisms to process language, and they do not depend on sequential architectures like RNNs or LSTMs. The Transformer architecture has been shown to be more efficient and provide better results than traditional RNN-based models in many tasks.

Key Concept of the Transformer:

- **Transformer** models are able to compute representations of their input and output by using **self-attention** mechanisms, without relying on sequence-aligned RNNs or convolutional layers.

The definition mentioned in the paper is:

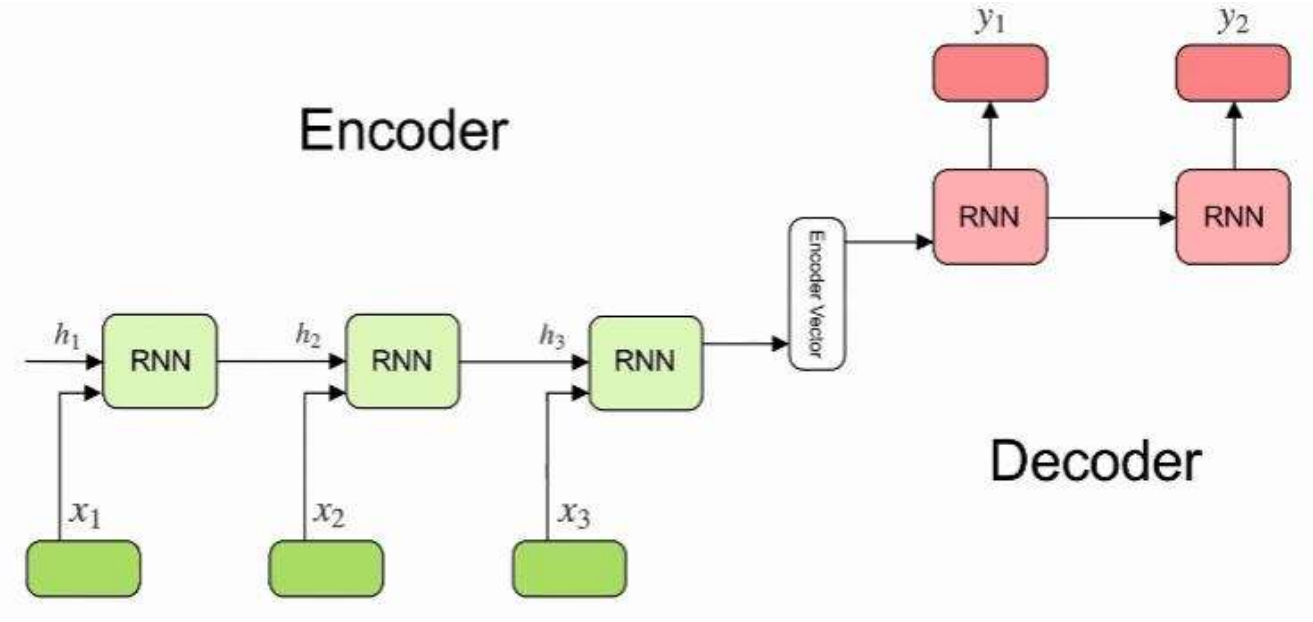
“Transformer is the first transduction model relying entirely on self-attention to compute representations of its input and output without using sequence-aligned RNNs or convolution.”

Model Concept

RNN

1. RNN Encoder-Decoder Model

Press enter or click to view image in full size



The **RNN Encoder-Decoder** model is commonly used for tasks like **neural machine translation**. This architecture consists of two main parts:

- **Encoder**: Processes the input sequence { x_1, x_2, x_3, …, x_n } and converts it into a fixed-length vector (often called the **context vector** or **encoder vector**). The output from the encoder is passed to the decoder.
- Each hidden state in the encoder is updated using the previous hidden state and the current input:

$$h_t = f(h_{t-1}, x_t)$$

Where:

- h_t is the hidden state at time step t,
- x_t is the input at time step t,
- f represents the recurrent function, typically an RNN, GRU, or LSTM cell.

- Decoder: Takes the fixed-length context vector from the encoder and generates an output sequence { y_1, y_2, …, y_m }. Each hidden state in the decoder is updated based on the previous hidden state and the previous output word:

$$s_t = f(s_{t-1}, y_{t-1})$$

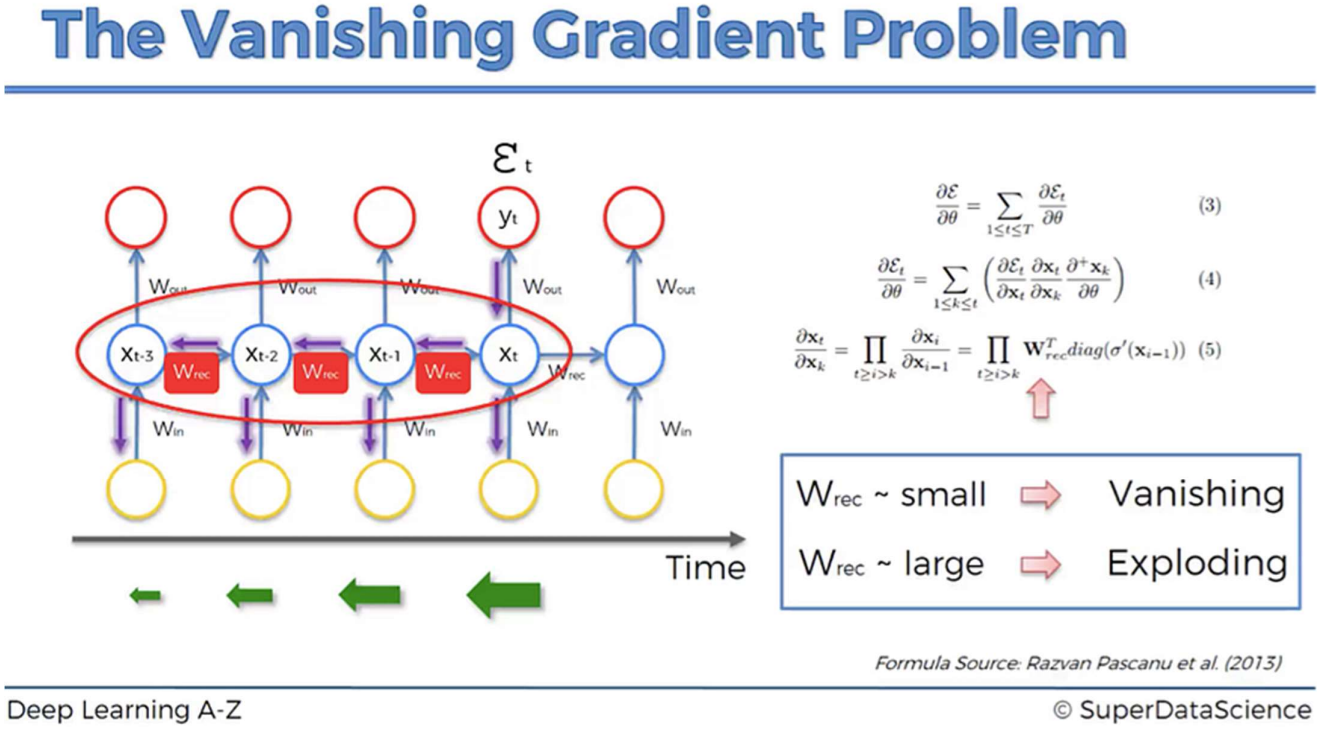
Where:
- s_t is the hidden state of the decoder at time step t,
- y_{t-1} is the previous word generated by the decoder.

This architecture was effective for many tasks before the introduction of Transformers in 2017.

This architecture was effective for many tasks before the introduction of Transformers in 2017.

2. RNN Limitations: The Vanishing Gradient Problem

Press enter or click to view image in full size



Despite the success of RNNs, they have inherent limitations, most notably the **vanishing gradient problem**. This occurs due to the sequential nature of RNNs and the repeated application of the same weight matrix during backpropagation through time.

What is the Vanishing Gradient Problem?

In the process of backpropagation, the gradients (the values used to update the weights of the network) become very small as they are propagated backward through many layers (or time steps). *When the gradients are too small, the network fails to learn long-range dependencies because the earlier layers (or time steps) receive almost no updates.*

The vanishing gradient issue is mathematically tied to the repeated multiplication of small values during the backpropagation process. If the weights in the recurrent weight matrix, Wrec, are small, multiplying them many times results in values that approach zero.

Mathematical Explanation of Vanishing Gradient:

1. **Forward Pass (RNN Update Rule)**:

The recurrent neural network updates its hidden states by multiplying with the same weight matrix Wrec:

$$h_t = W_{\text{rec}} h_{t-1} + W_{\text{in}} x_t$$

Where:
- Wrec is the recurrent weight matrix.
- Win is the input weight matrix.
- h_t is the hidden state at time step t.
- x_t is the input at time step t.

Impact on Learning:

- **Small Weights (Wrec≈ 0): Lead to vanishing gradients.** The gradients become so small that they cannot effectively update the weights, making it difficult for the model to learn long-range dependencies.

- **Large Weights ($W_{rec} \gg 1$): Lead to exploding gradients**, where the gradients grow exponentially and result in unstable training.

3. Solutions to the Vanishing Gradient Problem: LSTM and GRU

To address the vanishing gradient problem, architectures like **LSTM (Long Short-Term Memory)** and **GRU (Gated Recurrent Unit)** were introduced. These architectures use **gates** to control the flow of information and gradients, allowing the model to retain important information for longer periods.

- **LSTM** introduces an **input gate**, **forget gate**, and **output gate** to control what information is stored, forgotten, or used in the cell state. This mitigates the vanishing gradient problem by allowing the gradient to flow through the network more effectively.
- **GRU** simplifies the LSTM by combining some of the gates but still helps prevent the vanishing gradient problem.

4. Why Transformers Do Not Suffer from the Vanishing Gradient Problem

The introduction of **Transformers** addressed many of the limitations of RNN-based models, including the vanishing gradient problem. Transformers do not rely on sequential processing like RNNs, but instead use the **self-attention mechanism**, which allows for parallel computation and avoids the issue of vanishing gradients across long sequences.

Key Differences in Transformer Architecture:

- **Self-Attention Mechanism**: Each word in the input sequence can attend to all other words in the sequence, regardless of their distance. This eliminates the need to process the sequence one step at a time, which is where the vanishing gradient problem occurs in RNNs.
- **Parallelization**: Transformers allow for the entire sequence to be processed in parallel, significantly improving efficiency and making it easier to capture long-range dependencies.

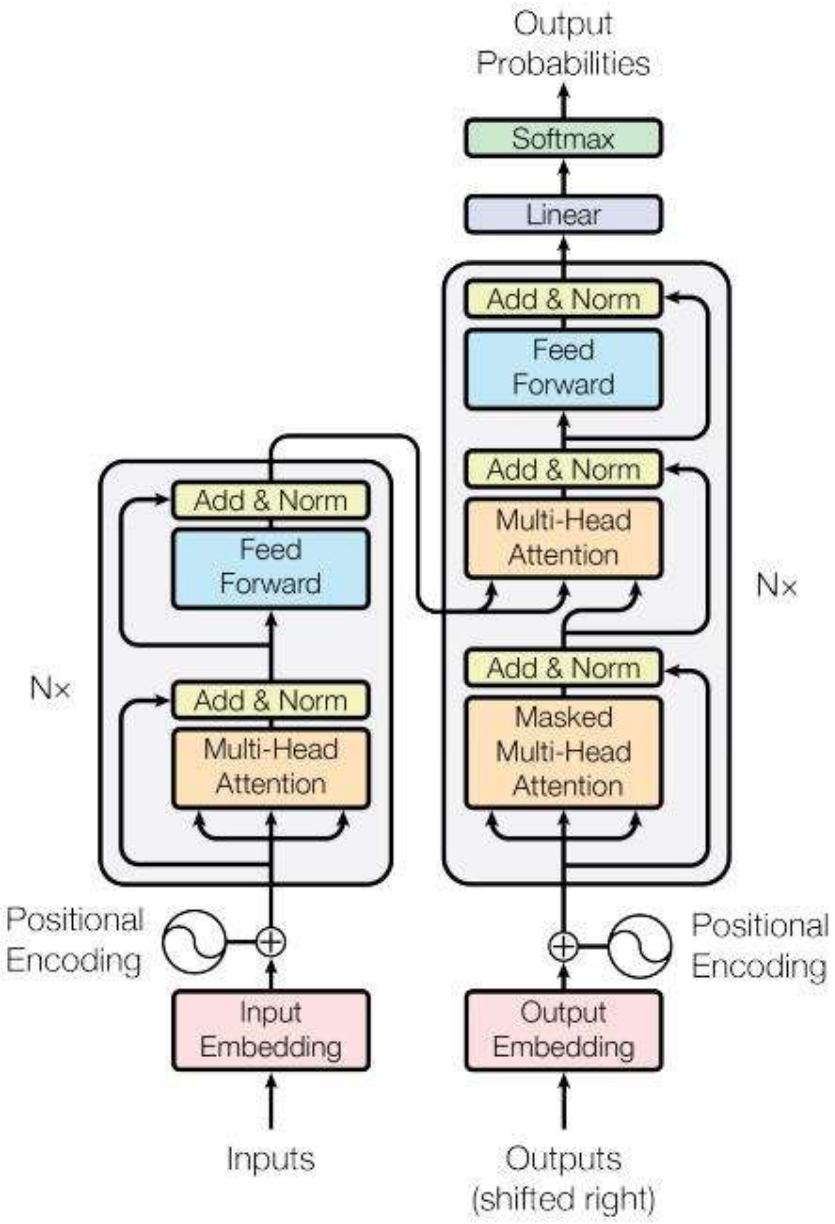
Conclusion:

The **Vanishing Gradient Problem** is a major limitation of RNNs and LSTMs, especially when trying to capture long-range dependencies in sequences. This issue arises due to the repeated multiplication of small gradients during backpropagation, which causes the model to struggle with learning long-term dependencies. While LSTMs and GRUs help mitigate this problem, the introduction of **Transformers** and their reliance on **self-attention** has provided a more efficient and effective solution to this issue.

Transformer Model Architecture

The **Transformer** model was introduced in the famous paper "**Attention Is All You Need**", which highlights the power of **Attention mechanisms** and demonstrates that it's possible to discard **RNNs (Recurrent Neural Networks)** altogether. The message conveyed by the title is that you can solve many problems by simply using the **Attention** mechanism without the need for RNNs. The Transformer not only avoids using RNNs, but it also performs remarkably well in a wide range of tasks.

The overall structure of the Transformer model is similar to the **Encoder-Decoder** architecture, with the **Encoder** on the left and the **Decoder** on the right.



Encoder

The input to the **Encoder** consists of word **embeddings** combined with **positional encodings**. These embeddings are then fed into a unified structure, which can be repeated **N** times, meaning the architecture can have **N layers**.

Each layer in the Encoder is divided into two main components:

- **Self-Attention Layer**: This is the key mechanism that allows each word to focus on other words in the sequence, regardless of their position.
- **Feed-Forward Layer**: A fully connected layer that processes the output of the attention layer.

Additionally, the architecture includes:

- **Skip Connections (Residual Connections)**: This connects the input of each sub-layer to its output, which helps in preventing vanishing gradients and improving convergence during training.
- **Normalization Layer**: Ensures stable learning by normalizing the outputs of each sub-layer.

Despite the presence of multiple layers, the architecture remains relatively simple.

Decoder

The **Decoder** also follows a similar repeatable structure, but its first input is different:

- **First Input**: The first input to the decoder is **prefix information** (typically, the start token for a sequence).
- **Subsequent Inputs**: The following inputs are the embeddings of the previously generated outputs, along with positional encodings.

Like the Encoder, the Decoder structure can be repeated **N** times and is divided into three main components:

1. **Self-Attention Layer**: This layer focuses on the previously generated tokens to predict the next word.
2. **Cross-Attention Layer**: Unlike the Encoder, the Decoder also includes a **cross-attention** layer that attends to the encoder's output. This layer ensures that the decoder takes into account the context provided by the input sequence.
3. **Feed-Forward Layer**: A fully connected layer that processes the outputs from the attention layers.

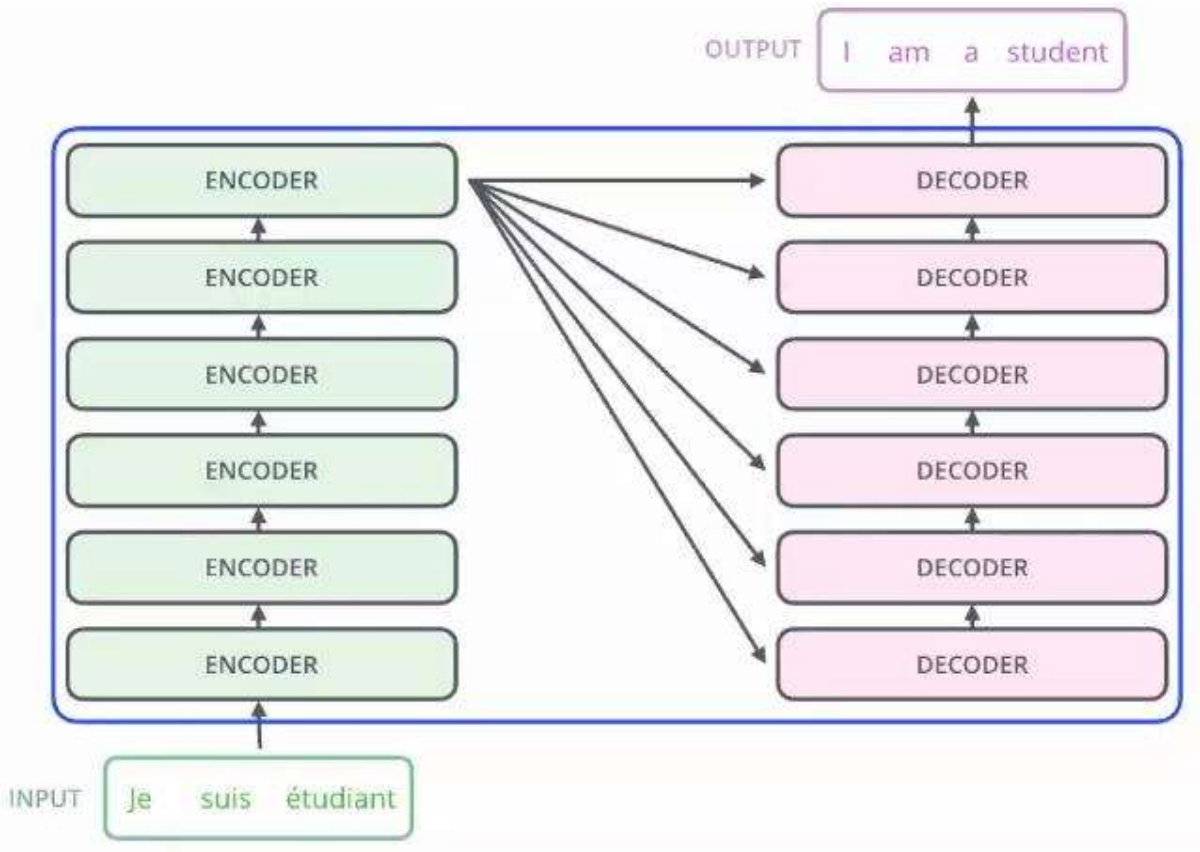
Just like in the Encoder, the Decoder also uses **skip connections** and **normalization layers** to stabilize the training process and improve the flow of gradients.

Output

The final output of the Decoder is passed through:

- A **Linear Layer** (a fully connected layer) to project the output into the required dimensions for the target vocabulary.
- **Softmax**: A softmax function is applied to generate a probability distribution over the possible next words.

Simplified Explanation of Transformer Network Structure



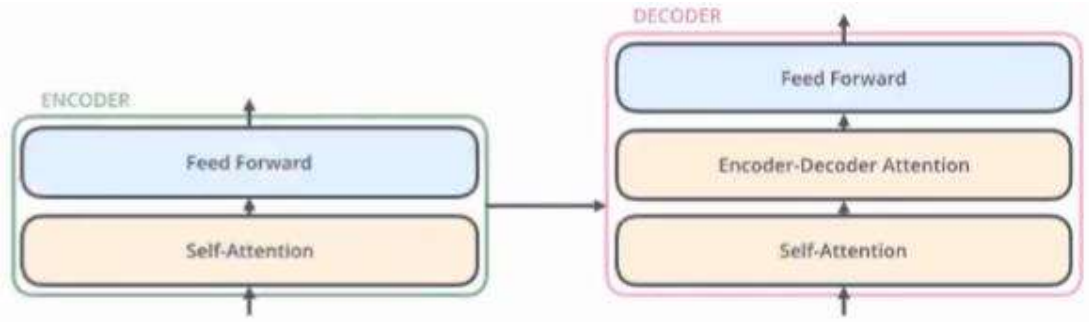
Here's a simplified version of how the Transformer architecture works:

1. The **Encoder** processes the input sequence, using self-attention to allow each word to focus on other words in the sequence. This is done across multiple layers.
2. The **Decoder** takes the prefix information (e.g., the previously generated words) and attends both to the previous tokens (self-attention) and to the output of the encoder (cross-attention) to generate the next word in the sequence.
3. Finally, the decoder's output is passed through a linear layer followed by a softmax layer to predict the next word.

Transformer Encoder Module:

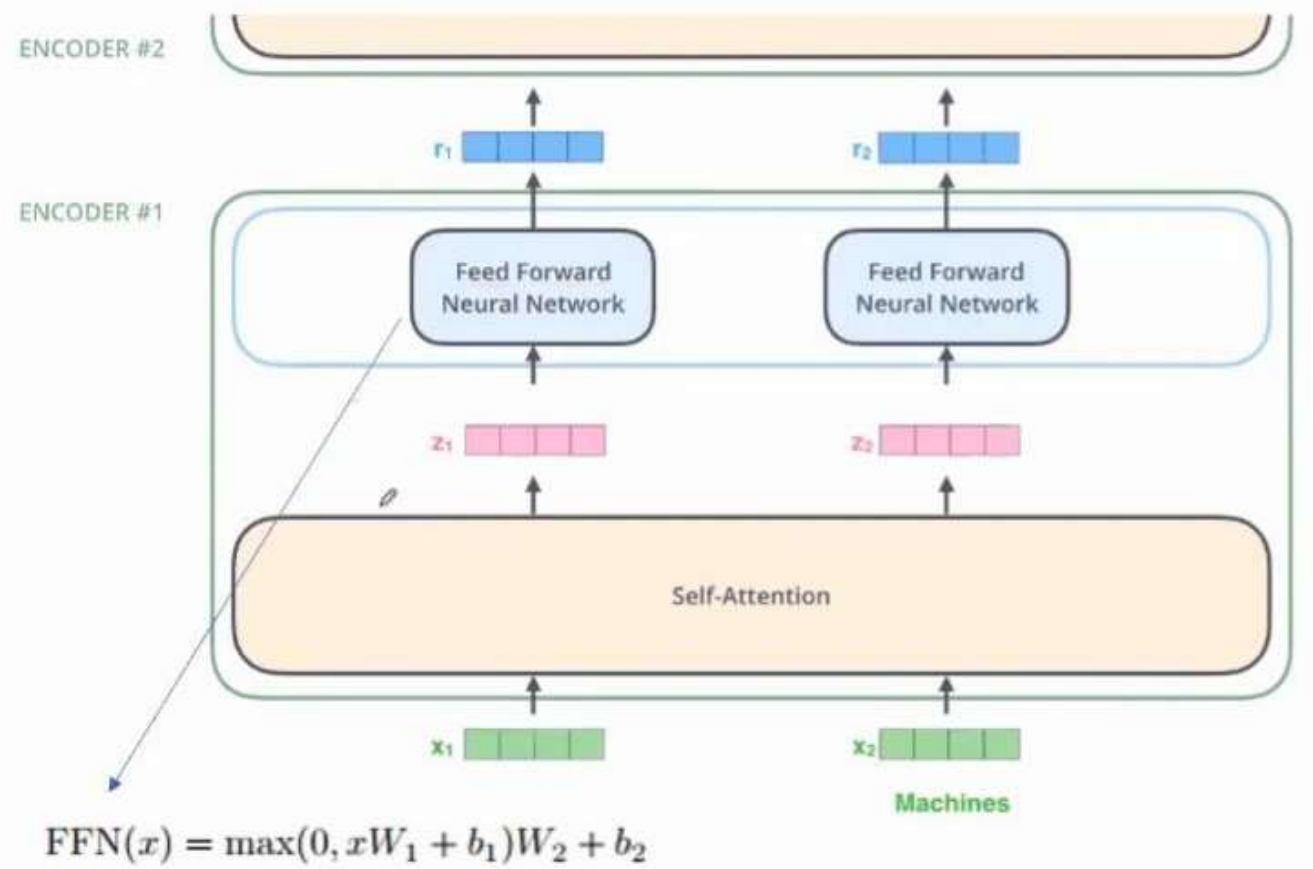
Self-Attention Mechanism

In this simplified version of the network, both the **Encoder** and **Decoder** are represented as having a single module. This module is the core of the Transformer architecture.



Now let's look at the simplest model and how it works. The focus is to understand how the **Attention mechanism** operates inside the **Encoder** and **Decoder**.

Press enter or click to view image in full size

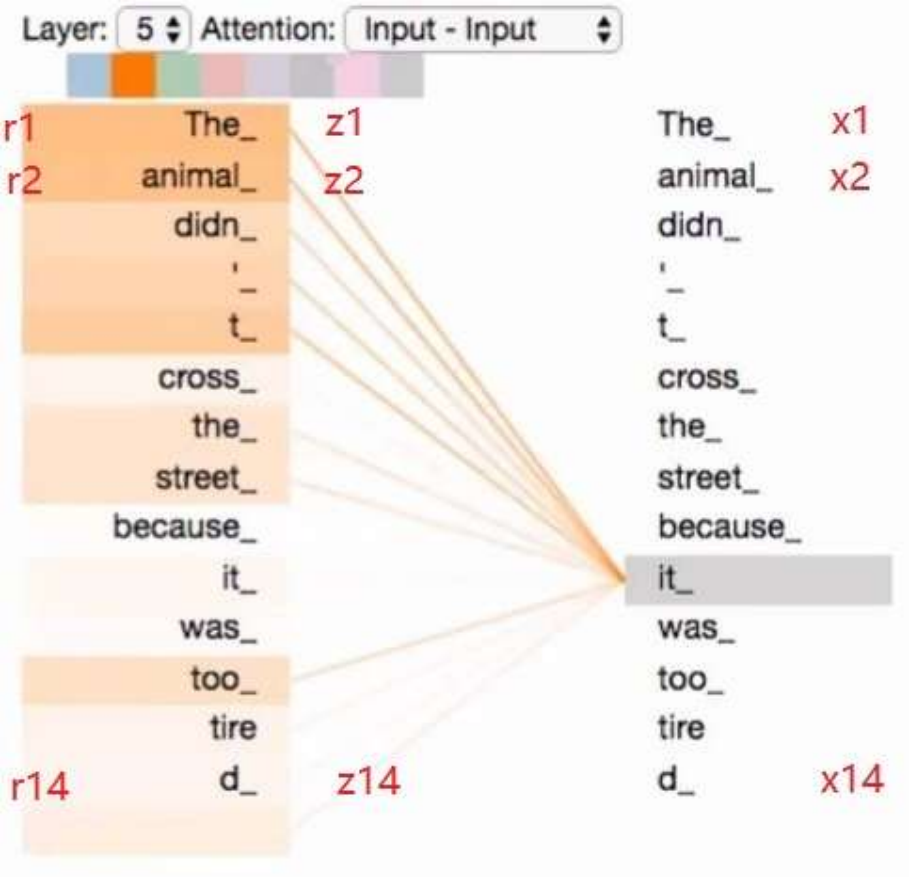


The green box (**Encoder #1**) represents an independent module within the Encoder. Below this green box, the input consists of the **embeddings** of two words. The goal of this module is to transform x_1 into another vector r_1, with the same dimensionality as the input vector. This transformation occurs layer by layer, upwards through the network.
The transformation process is divided into a few steps:
1. The first step is **self-attention**.
2. The second step is a regular **Fully Connected Neural Network (Feed Forward Network).**

However, note that in the **Self-Attention** step, **all input vectors participate together in the process.** In other words, both x_1 and x_2 exchange and mix information with each other, resulting in the intermediate variables z_1 and z_2.

In contrast, the **Fully Connected Neural Network** operates independently on each vector. So, z_1 and z_2 pass separately through the fully connected layers, resulting in the output vectors r_1 and r_2.

Although x_1 and x_2 do not have direct knowledge of each other, thanks to the **Self-Attention** mechanism in the first step, information is exchanged, so r_1 and r_2 each carry information derived from both x_1 and x_2.



If we use an intuitive way to understand Self-Attention, let's assume the sentence on the left side is represented by inputs x_1, x_2, …, x_{14}. These inputs then go through the Self-Attention mechanism, resulting in outputs z_1, z_2, …, z_{14}.

Why is it called **Self-Attention**? It's because each word in the sentence looks at other words to determine how much influence they have on itself. For example, if we consider the word **it**, the attention mechanism will determine which other words in the sentence are most relevant to **it**. If the color represents the strength of the influence (darker meaning stronger), we can see that the words **The** and **Animal** have the most influence on **it**. Therefore, **Self-Attention means that each word in the sentence focuses on the other words in the sentence to determine which ones to pay attention to.**

Attention and **Self-Attention** are both mechanisms used in deep learning models, particularly for tasks like natural language processing and machine translation. They share a similar mathematical formulation but differ in terms of where they are applied and how the information flows. Let me explain their key differences:

1. What are they used for?

- **Attention**:

In a typical **Encoder-Decoder** architecture (like in machine translation), the **Attention** mechanism is used to focus on relevant parts of the input sequence when generating output tokens. For example, in translating an English sentence to Chinese, the **Source** is the English sentence, and the **Target** is the Chinese translation. Here, attention allows the model to align parts of the **Target** sentence (like Chinese words) with the most relevant parts of the **Source** sentence (English words).

- **Self-Attention**:

Self-Attention (or intra-attention) is used within a single sequence, either the **Source** or **Target** , to allow each token (word or element) to focus on other tokens in the same sequence. This is particularly important

for capturing dependencies between distant tokens in the same sentence. For example, *if we have a long sentence in English,* ***Self-Attention*** *allows every word in the sentence to see and attend to every other word, helping the model understand relationships across the sequence.*

2. Where do they occur?

- **Attention**:

This happens between two different sequences — like the **Source** and **Target** in an Encoder-Decoder setup. For instance, when generating a Chinese word (Target), the **Attention** mechanism helps the model focus on the most relevant English word (Source).

- **Self-Attention**:

This occurs within the same sequence, such as within the **Source** or **Target** itself. It's useful when the model needs to understand the relationships between words in a single sentence, like identifying which other words a given word should focus on to understand context.

3. Information Flow:

- **Attention**:

In Attention, the model is working across two sequences. When generating a word in the **Target** sequence (e.g., a Chinese translation), the model uses attention to search across all words in the **Source** sequence (English). This helps the model align and translate the word correctly, based on its relevance to the source.

- **Self-Attention**:

In Self-Attention, the model processes a sequence and lets every word or token focus on every other word in that same sequence. For example, in the sentence "The animal didn't cross the street because it was too tired," the word "it" would need to attend to "animal" to correctly understand that "it" refers to the animal. Self-attention allows this kind of internal relationship to be learned without requiring the sequence to be processed step-by-step like in RNNs.

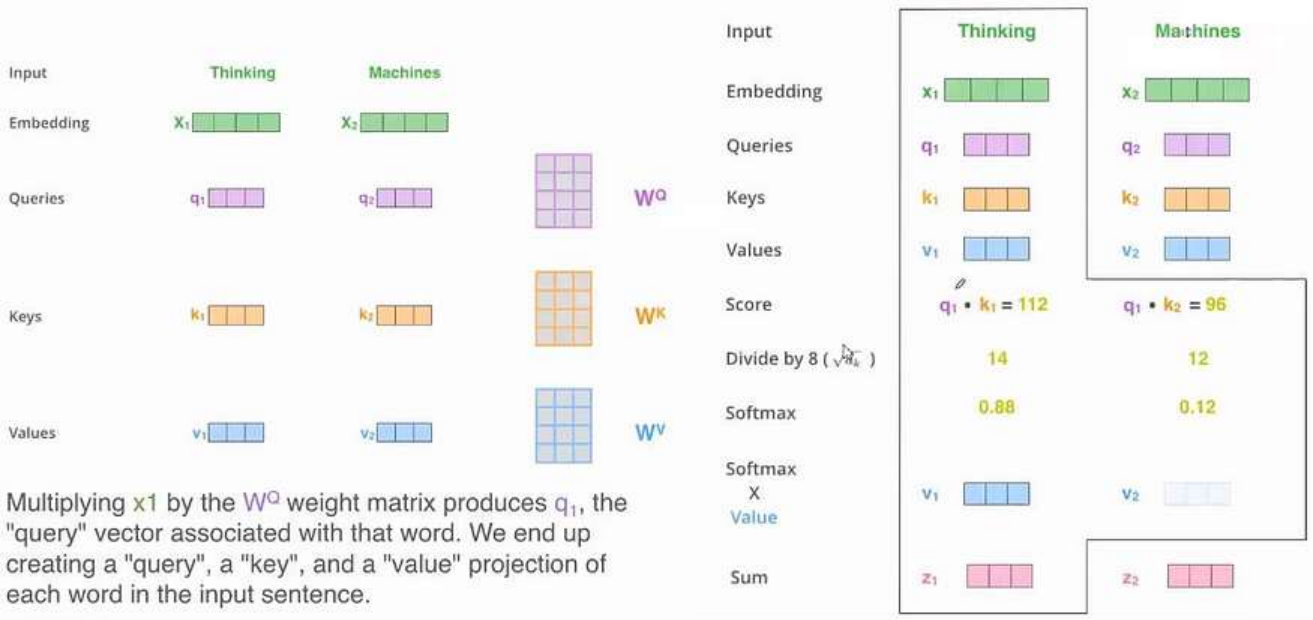
4. Why is Self-Attention Powerful?

Self-Attention, especially in the Transformer architecture, eliminates the need for sequential processing (like in RNNs). Each word can attend to every other word simultaneously. This makes models faster (because they can be parallelized) and better at capturing long-range dependencies. In contrast, **Attention** across sequences is more useful for tasks where you need to align and translate between different languages or different representations.

Summary:

- **Attention**: Aligns different sequences, typically used between **Source** and **Target** in tasks like translation (cross-sequence).
- **Self-Attention**: Works within the same sequence, allowing each element to understand and attend to other elements in the same input, used in models like Transformers.

Press enter or click to view image in full size



The diagram illustrates the Self-Attention mechanism. Given the input word embeddings, x_1 and x_2, we want to transform them into z_1 and z_2.

1. **Initial Transformation:**

First, x_1 is transformed into three different vectors, denoted as Query (q_1), Key (k_1), and Value (v_1). Similarly, x_2 is transformed into **Query (q_2), Key (k_2), and Value (v_2).**

The simplest way to transform one vector into another is by multiplying it with a matrix. Therefore, we need three different matrices:

$$W_Q\,,\ W_K\,,\text{and}\ W_V$$

The transformations are as follows:

$$q_1 = x_1 W_Q, \quad q_2 = x_2 W_Q$$
$$k_1 = x_1 W_K, \quad k_2 = x_2 W_K$$
$$v_1 = x_1 W_V, \quad v_2 = x_2 W_V$$

Here, x_1 and x_2 share the same weight matrices W_q, W_k, and W_v, which means there is some level of information exchange between the words.

2. Computing z_1 and z_2:
The next step is to compute z_1 and z_2 by taking linear combinations of v_1 and v_2. These combinations are weighted by coefficients θ_{11}, θ_{12}, θ_{21}, and θ_{22}:

$$z_1 = \theta_{11} v_1 + \theta_{12} v_2$$
$$z_2 = \theta_{21} v_1 + \theta_{22} v_2$$

3. How to compute the weights (θ)?
The weights θ are computed using the following formula:

$$[\theta_{11}, \theta_{12}] = \text{softmax}\left(\frac{q_1 k_1^\top}{\sqrt{d_k}}, \frac{q_1 k_2^\top}{\sqrt{d_k}}\right)$$
$$[\theta_{21}, \theta_{22}] = \text{softmax}\left(\frac{q_2 k_1^\top}{\sqrt{d_k}}, \frac{q_2 k_2^\top}{\sqrt{d_k}}\right)$$

Here, the dot product between the Query and Key vectors gives a score representing how much attention each word should pay to the other. The scores are scaled by the dimension of the key vectors, d_k, to prevent the values from becoming too large. The softmax function converts these scores into probabilities, so that the sum of the weights is 1.

4. Final Step:
With the computed attention weights θ, we can combine the Value vectors (v_1 and v_2) to get the final outputs z_1 and z_2. These represent the updated representations of the input words after applying self-attention.
Once z_1 and z_2 are computed, they are passed through a fully connected layer to get the final output of the **Encoder**, which would be r_1 and r_2.

Why do we need Query, Key, and Value vectors?

The concept of **Query**, **Key**, and **Value** comes from early work in information retrieval. A **Query** represents a search term, **Key** represents an item to be matched, and **Value** represents the actual content of the item. In self-attention, each word in a sequence has its own query, key, and value vectors. The query searches for the most relevant keys, and the attention weights determine how much influence the corresponding values should have.

Efficient Matrix Multiplications:

To compute the attention for multiple words in a sentence, we can perform matrix multiplications in a parallelized way, which is more efficient than computing them one by one. The matrix form of the attention mechanism is:

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

Where Q, K, and V represent the concatenated matrices of all queries, keys, and values in the sequence.

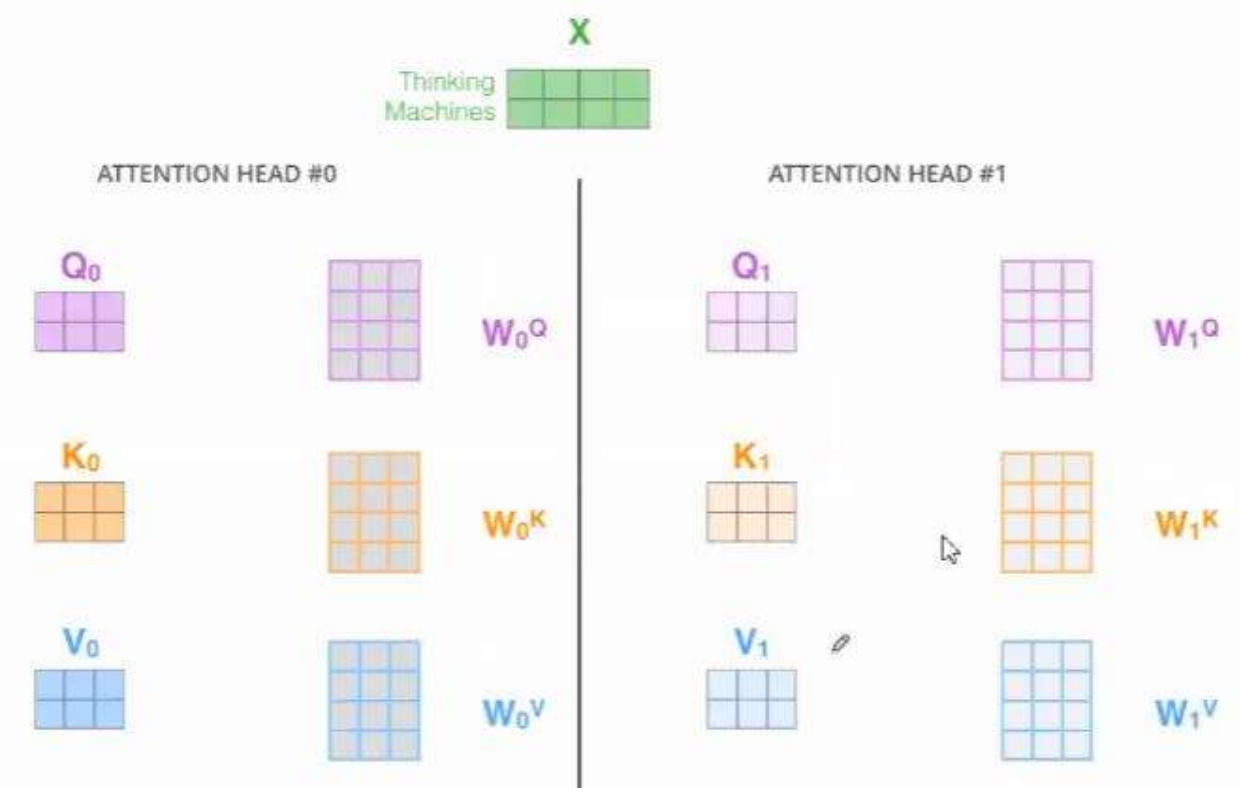
Why divide by $\sqrt{d_k}$?

Dividing by $\sqrt{d_k}$ helps to prevent the dot product values from becoming too large when the dimensionality of the key vectors (d_k) is large. Without this scaling, the softmax function could produce very small gradients during backpropagation, leading to the vanishing gradient problem. The scaling factor ensures that the softmax operates within a reasonable range of values.

Multi-Headed Attention

Press enter or click to view image in full size

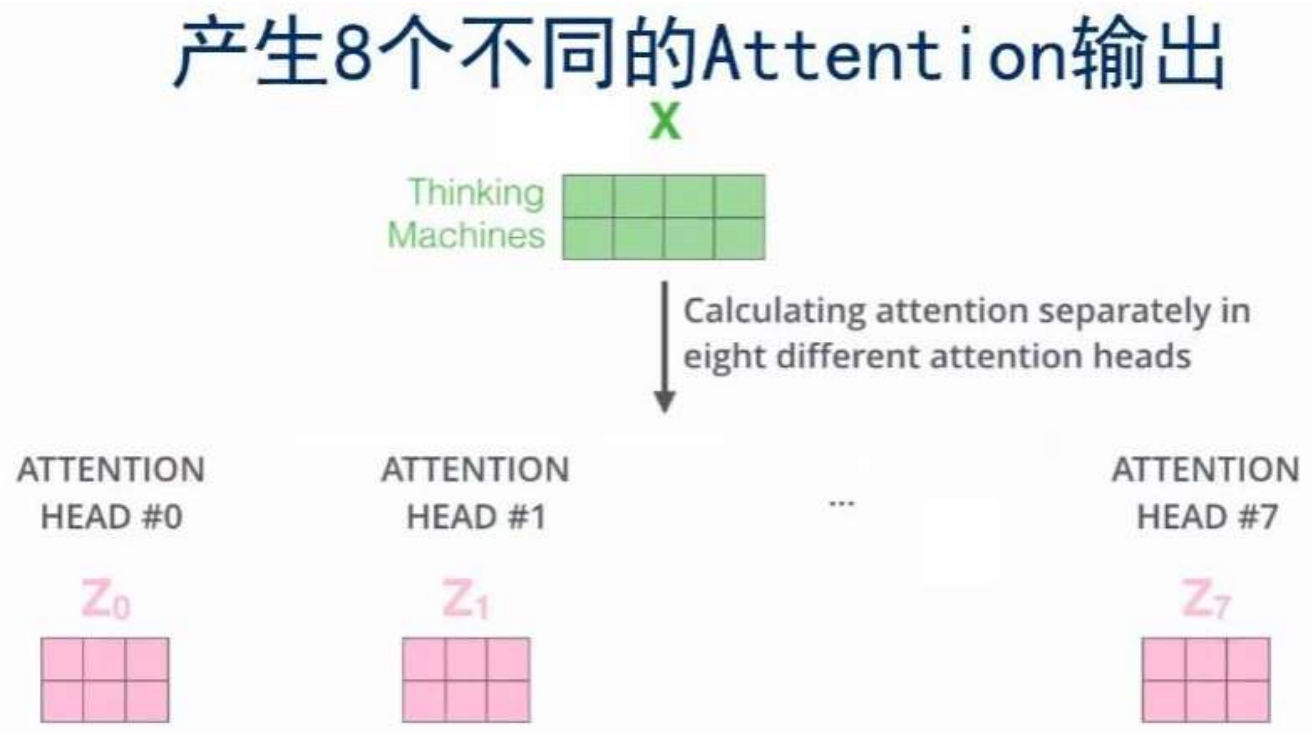
If we use different weight matrices W_Q, W_K, and W_V, we can obtain different **Query** (Q ,), **Key** (K), **and Value** (V) **matrices. Multi-Headed Attention** refers to the use of multiple sets of weight matrices W_Q, W_K, and W_V.



What is the benefit of Multi-Headed Attention?

The main advantage of using Multi-Headed Attention is that it allows the model to capture different types of relationships within the data. By having multiple sets of Q, K, and V matrices, each set can attend to different aspects of the input, thus providing a richer representation of the relationships in the data.
For a given input x, multiple Multi-Headed Attention heads will produce different z outputs, each corresponding to a different aspect or perspective of the input.

Press enter or click to view image in full size



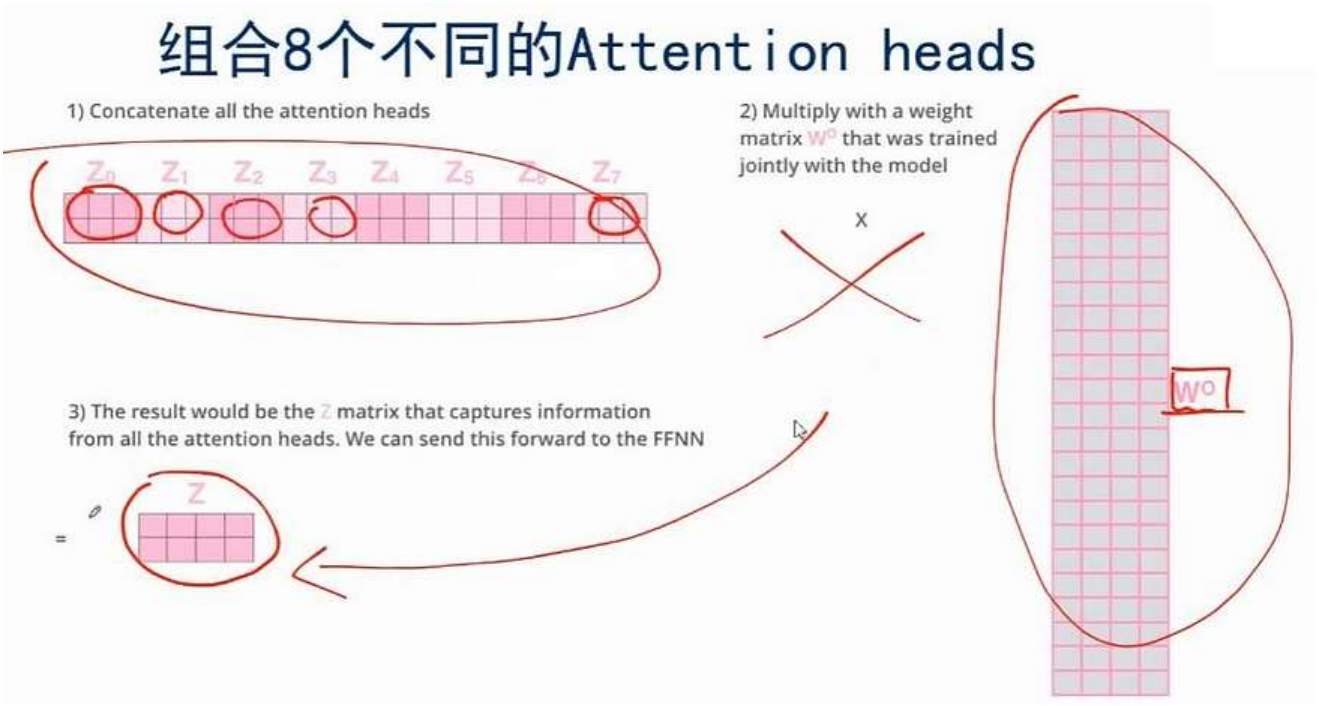
Combining the Results:

Now that each input x has multiple versions of z, how do we combine them into a single z?
The method is straightforward: you concatenate all the different z outputs into a longer vector, and then pass it through a fully connected layer (i.e., multiply it by a matrix) to obtain a shorter vector z.

Final Step in Multi-Headed Attention:

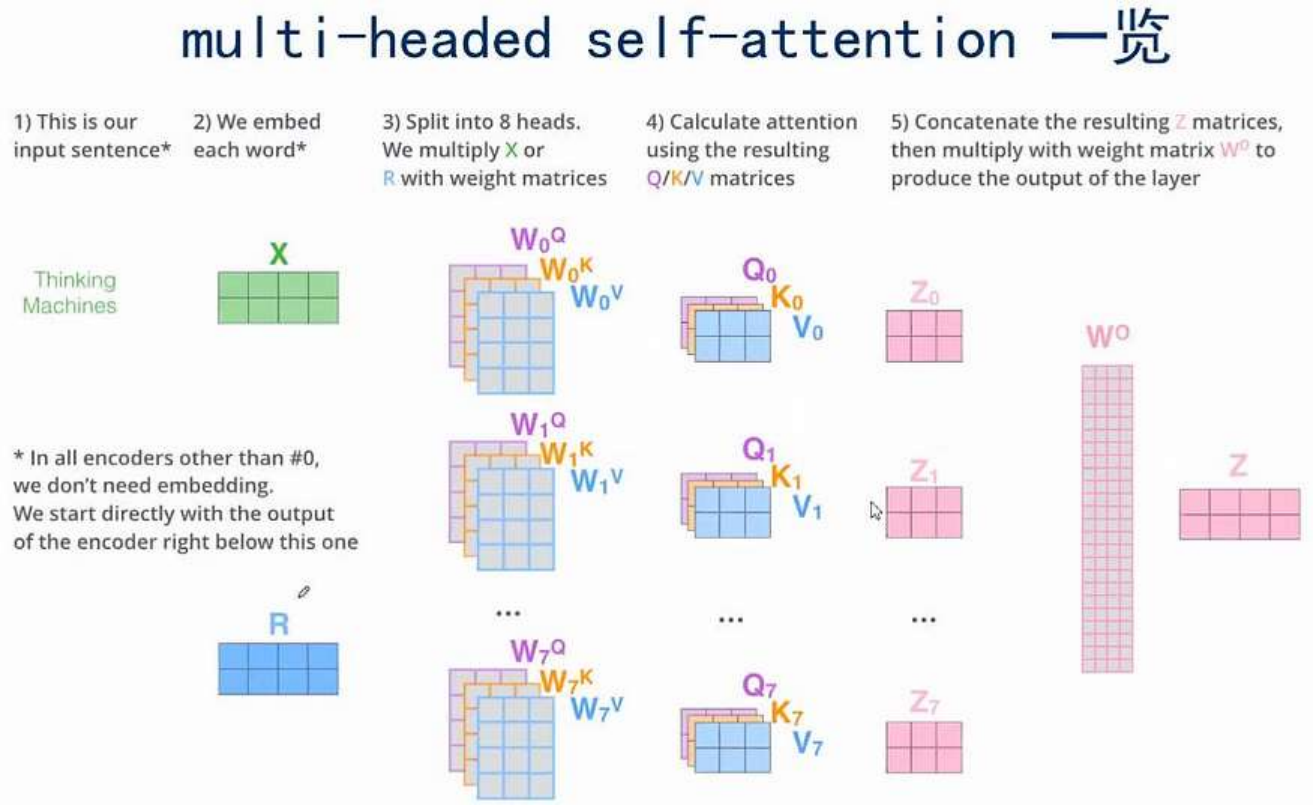
The final step in **Multi-Headed Attention** involves combining the outputs from multiple attention heads into the desired output z. This process ensures that the model captures different aspects of the input from each attention head, and then compresses them into a single output.

Press enter or click to view image in full size



Formula for Multi-Headed Attention:

Press enter or click to view image in full size



The **Multi-Headed Attention** mechanism can be expressed with the following formula:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W_O$$

Where:

- Each head is computed as:

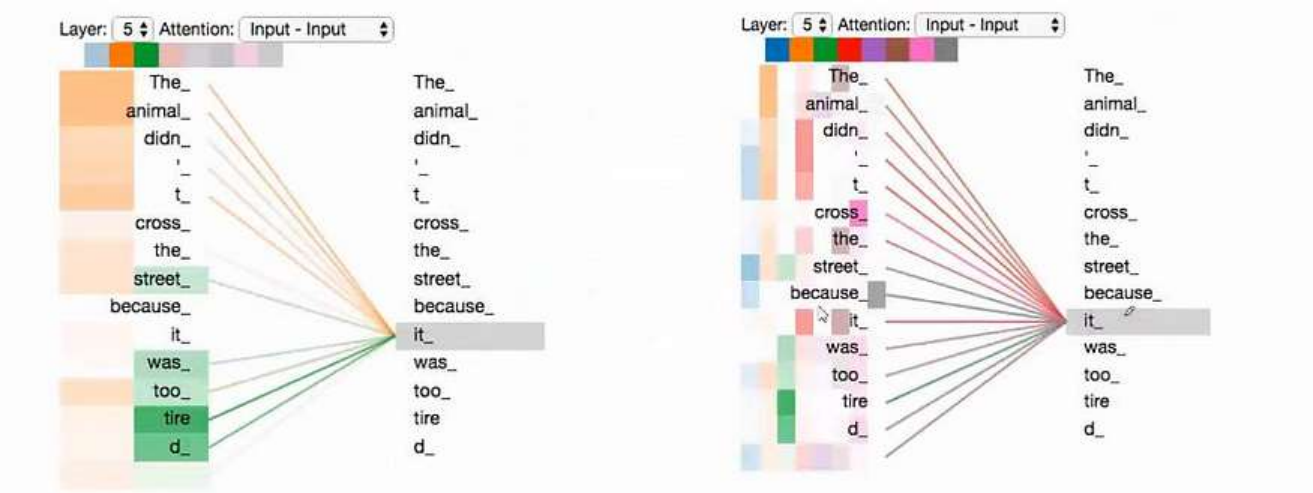
$$\text{head}_i = \text{Attention}(QW_Q^i, KW_K^i, VW_V^i)$$

Press enter or click to view image in full size

Here:

- W_Q^i W_K^i, W_V^i are different weight matrices for each attention head.
- The attention computation for each head is performed independently, after which the results are concatenated.
- Finally, the concatenated result is multiplied by a matrix W_O to produce the final output.

Press enter or click to view image in full size



As we encode the word "it", one attention head is focusing most on "the animal", while another is focusing on "tired" -- in a sense, the model's representation of the word "it" bakes in some of the representation of both "animal" and "tired".

In the image, we see an example of **multi-headed attention** where eight different attention heads are used, each represented by different colors, ranging from blue to gray. Each attention head looks at the sentence and assigns different weights to each word. The darker the color, the higher the weight or focus assigned to that word.

Key Points:

- On the left side, the **orange-colored** attention head focuses the most on the word **"animal"** when processing the word **"it"** . This means that, in this head, the model associates **"it"** with **"animal"** the most, treating **"it"** as a reference to the animal in the sentence.
- On the right side, the **green-colored** attention head focuses more on **"tired"**. This head associates the word **"it"** with a state or condition, treating **"it"** as being tired rather than being a physical object.

Different Perspectives:

- **Orange Head**: In this head, the focus is on identifying **"it"** as a physical object, specifically **"the animal"**. This attention focuses on **what "it" refers to** in terms of objects.
- **Green Head**: In this head, the focus shifts towards the **state** or **condition** of **"it"**, associating **"it"** with the concept of being **"tired"**.

This demonstrates how **multi-headed attention** allows the model to view the same word, **"it"**, from different perspectives. One attention head interprets **"it"** as an object (the animal), while another focuses on the condition (being tired). These different heads capture distinct aspects of the same word.

What is the Significance of Multi-Head Attention in NLP for Semantics and Syntax?

Multi-Head Attention plays a vital role in improving the understanding of both **semantics** (meaning) and **syntax** (structure) in NLP models. Here's why it's important:

1. Understanding Multiple Contexts Simultaneously

Each head in multi-head attention allows the model to attend to different parts of the sentence independently. This is particularly helpful for capturing:

- **Semantics**: Different meanings of words in different contexts. For example, a word like **"bank"** can have different meanings depending on whether the sentence is about finance or rivers. Different attention heads can focus on different contextual clues to resolve this ambiguity.

- **Syntax**: Grammatical relationships between words in a sentence. One head might focus on the relationship between a subject and its verb, while another focuses on object-verb relationships.

2. Handling Ambiguity (Polysemy)

In language, words often have multiple meanings based on context. Multi-head attention helps to resolve these ambiguities by allowing some heads to focus on different meanings of the same word. For instance, in the sentence **"I saw the man with a telescope,"** one attention head might focus on the fact that the **man** had a telescope, while another focuses on the possibility that **I** used the telescope to see the man. This multi-perspective view helps the model understand the sentence more comprehensively.

3. Capturing Long-Range Dependencies

In long sentences, words at the beginning may be semantically or syntactically related to words far down the sentence. Multi-head attention enables the model to capture these long-range dependencies effectively. Some heads can focus on the near-term relationships, while others can focus on distant dependencies.

For example, in the sentence **"The dog that barked loudly chased the cat,"** one head might focus on the relationship between **"dog"** and **"chased,"** while another focuses on the relationship between **"dog"** and **"barked."** This parallel focus ensures that the model retains both syntactic and semantic meaning across distant words.

4. Enriching Word Representations

When processing a word in a sentence, multi-head attention allows the model to look at that word from different angles or "views." One head might focus on the word's role in a grammatical structure (syntax), while another might focus on the word's meaning in the given context (semantics). This results in a richer and more comprehensive representation of each word.

For example, when processing a word like **"he,"** one head might focus on identifying its grammatical role (subject), while another head might focus on resolving who **"he"** refers to (semantic role). This richer representation leads to better language understanding in tasks like translation, summarization, and text generation.

5. Improving Translation and Generation

In machine translation, multi-head attention allows the model to better align words between the source and target languages. Different heads can focus on different parts of the source sentence, helping the model decide how best to translate individual words based on multiple grammatical and semantic clues. This leads to more accurate and nuanced translations.

Conclusion:

Multi-head attention significantly enhances an NLP model's ability to:

- Resolve word meanings based on context (semantics),
- Understand relationships between words (syntax),
- Capture long-range dependencies across sentences, and
- Generate richer word representations for better performance on language tasks like translation and summarization.

By allowing the model to look at the input from multiple perspectives at once, multi-head attention improves the model's ability to understand and generate human language more accurately.

Word Embedding Input in the Encoder

The input to the **Encoder** in a Transformer model consists of word embeddings, which represent words as dense vectors. Typically, there are two main approaches for handling word embeddings:

1. Using Pre-trained Embeddings (Fixed):

In this approach, pre-trained embeddings like **Word2Vec** or **GloVe** are used and treated as fixed. These embeddings are stored in a **Lookup Table** that maps each word to its corresponding pre-trained vector. No further updates are made to these embeddings during training, meaning the model relies on the pre-trained knowledge to represent the words.

2. Trainable Embeddings (Random Initialization or Pre-trained):

In this approach, the embeddings are either initialized randomly or using pre-trained vectors, but they are set as **trainable**. During the training process, the model continuously updates these embeddings as part of the optimization. This approach allows the embeddings to be tailored to the specific task and data the model is being trained on, providing better adaptability.

Transformer's Choice:

The **Transformer** uses the second approach, where the embeddings are initialized (randomly or using pre-trained embeddings) and are **trainable** throughout the training process. This end-to-end training allows the model to refine its word embeddings in conjunction with the rest of the model.

Positional Encoding in Transformers

When inputting data into the Transformer model, it's not enough to only include the **word embeddings** for each token; we also need to add **Positional Encoding**. This means that the entire input embedding to the model is a combination of **Word Embeddings** and **Positional Embeddings**.

The reason we add positional information is that **Transformers** don't have a recurrent structure (like RNNs) or convolutional layers that can process sequence order. Therefore, the model must explicitly be told the relative positions of words in a sequence. Without this, the model wouldn't know which word comes first, second, etc.

Why Positional Encoding is Needed:

- **Transformers lack sequential awareness**: Unlike RNNs or LSTMs that process input step by step, Transformers process all tokens in parallel. This means they don't inherently know the order of words.
- **Encoding word positions**: To make sure the model understands the order of the words, we introduce positional encodings to the embeddings before feeding them into the encoder and decoder. These positional encodings give the model a sense of where each word is located in the sequence, allowing it to consider relative distances between words.

Formula for Positional Encoding:

The positional encoding uses a combination of **sine** and **cosine** functions to compute unique encodings for each position:

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

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

Press enter or click to view image in full size

Where:

- pos is the position of the word in the sentence.
- i is the dimension of the embedding.
- d_{model} is the total dimensionality of the model (e.g., 512 dimensions).

The sine and cosine functions are used because they provide a way to encode positions in a continuous and periodic way. This method allows the model to generalize to sequences longer than what it was trained on, while also preserving the distances between words in the embedding space.

Conclusion:

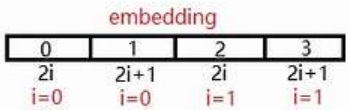
By adding **positional encoding** to word embeddings, the Transformer can process input tokens while keeping track of the order of the sequence, ensuring that both the meaning of the words (via embeddings) and their positions (via positional encoding) are taken into account during model training.

Explanation of Positional Encoding Formula

Press enter or click to view image in full size

- 对于一个单词 w 位于 $pos \in [0, L-1]$, 假设使用 encoding 的维度为 4, 那么这个单词的 encoding 为:

$$e_w = \left[\sin\left(\frac{\text{pos}}{10000^{\circ}}\right), \cos\left(\frac{\text{pos}}{10000^{\circ}}\right), \sin\left(\frac{\text{pos}}{10000^{2/4}}\right), \cos\left(\frac{\text{pos}}{10000^{2/4}}\right)\right]$$
$$= \left[\sin(\text{pos}), \cos(\text{pos}), \sin\left(\frac{\text{pos}}{100}\right), \cos\left(\frac{\text{pos}}{100}\right)\right]$$



- 根据公式:

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

The image explains the use of the **positional encoding formula** in a simple example. Let's break down the content step by step.

For a word w at position pos, with a positional encoding dimension of 4, the encoding is as follows:

Example Encoding:

The encoding for a word w at position pos with an encoding dimension of 4 is:

$$e_w = \left[\sin\left(\frac{\text{pos}}{10000^{0}}\right), \cos\left(\frac{\text{pos}}{10000^{0}}\right), \sin\left(\frac{\text{pos}}{10000^{2/4}}\right), \cos\left(\frac{\text{pos}}{10000^{2/4}}\right)\right]$$

In this case, the encoding vector consists of 4 elements: alternating sine and cosine functions applied to the position of the word in the sentence, with different scaling factors for each dimension.

In simpler terms, for this particular example, the encoding consists of:
- sin of the position scaled by 1000^{00},
- cos of the position scaled by 1000^{00},
- sin of the position scaled by $1000^{01/2}$,
- cos of the position scaled by $1000^{01/2}$.

General Formula:

The general formula for positional encoding is:

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

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

Press enter or click to view image in full size

Where:

- pos is the position of the word in the sequence,
- $2i$ and $2i+1$ refer to even and odd dimensions of the encoding,
- d_{model} is the total number of dimensions in the model's embedding.

Press enter or click to view image in full size

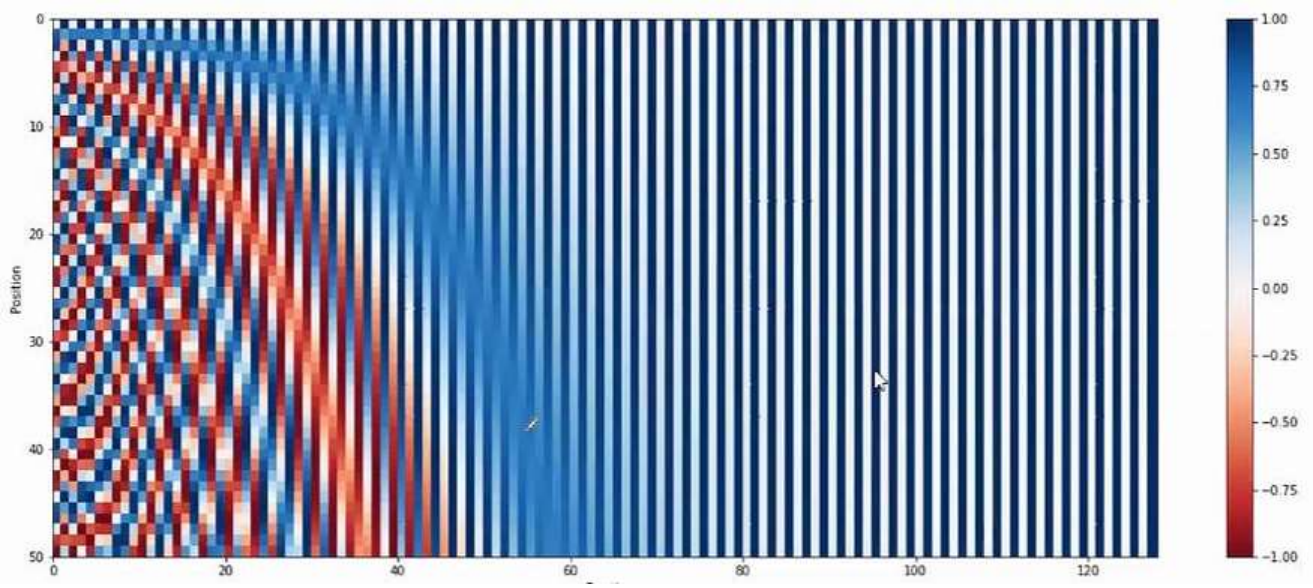
Explanation:

- **Sine for even dimensions**: The sine function is applied for even positions in the embedding.
- **Cosine for odd dimensions**: The cosine function is applied for odd positions.
- **Exponential scaling**: The division by $10000^{\frac{2i}{d_{\text{model}}}}$ ensures that different frequencies are used for different dimensions, capturing both short-range and long-range dependencies.

In the example provided, the embedding dimension $d_{\text{model}} = 4$, so the positional encoding is split across 4 components, with alternating sine and cosine values.

Explanation of Positional Encoding Visualization

Press enter or click to view image in full size



The visualizations show how **positional encoding** works and why it is designed this way. Each row in the heatmap represents the **positional encoding vector** for a specific word in the sentence.

1. Understanding the Positional Encoding Heatmap

In the heatmap:

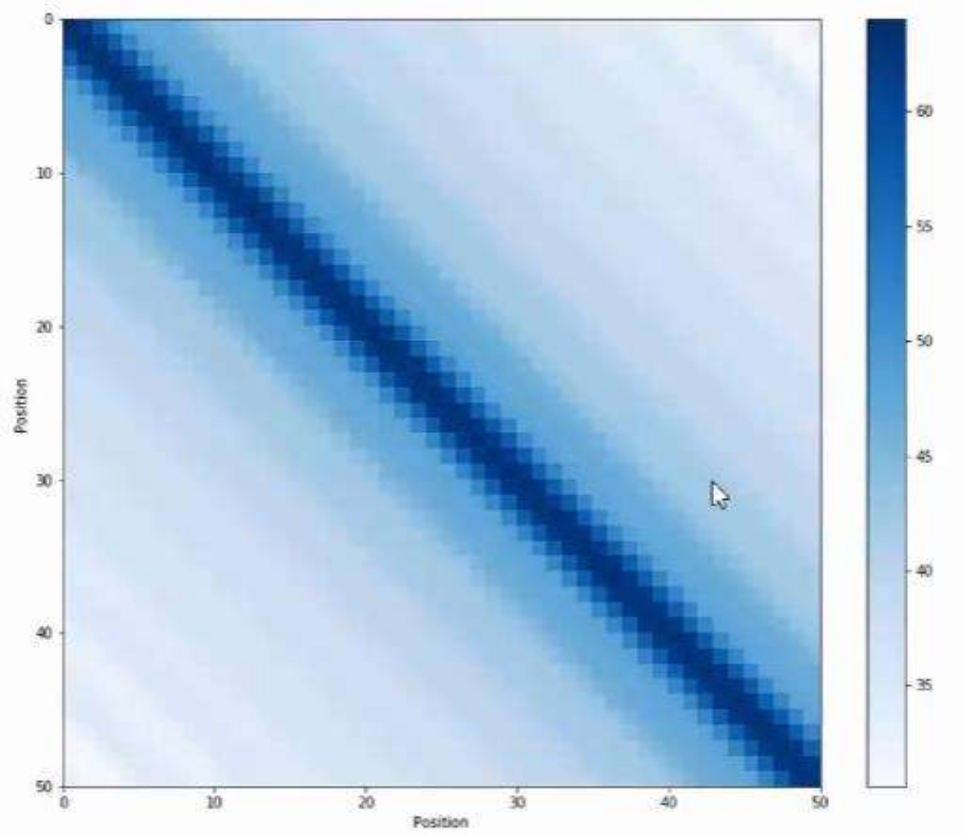
- Each row represents the **positional encoding** for a word in a sentence of length 50 (50 positions).
- Each column represents a specific dimension of the **embedding** (128 dimensions in this case).
- The color shows the value of the sine or cosine function used to calculate the positional encoding. Red indicates higher values, while blue indicates lower values.

Each word in the sentence has a **unique** positional encoding, which is critical for the model to know the order of the words.

2. Positional Encoding is Unique for Each Position

The image demonstrates that every position has a unique encoding. The sine and cosine waves used in the formula create distinct patterns for each position, even for longer sequences. This uniqueness allows the Transformer model to distinguish the position of each word in a sequence, no matter how long the sentence is.

3. Positional Encoding Captures Distance Relationships



The second image shows a matrix of **dot products** between the positional encoding vectors for different positions. The darker the color, the higher the similarity between two positions. The diagonal line shows that each word is most similar to itself (position-to-position comparison). As the distance between two positions increases, their encoding vectors become less similar, which is reflected by the lighter colors further from the diagonal.

This behavior is crucial for the model to understand how far apart two words are, even without processing them in a sequential order. The dot products decrease as the distance between positions increases, which means the encoding captures **relative positions** in the sentence.

Summary:

Positional encoding is a way for Transformer models to capture the order and distance relationships between words in a sentence. Each position is encoded using sine and cosine functions, which ensures that each word has a unique position-related vector. These encodings allow the model to understand relative word positions without relying on sequential processing, which is crucial for the model's performance in tasks like translation, summarization, and language generation.

Skip Connection and Layer Normalization in Transformers

In Transformer models, **Skip Connections** and **Layer Normalization** are used in both the encoder and decoder to stabilize and improve the training process.

1. Skip Connection (Residual Connection):

- **Skip connections** were first introduced in ResNet (Deep Residual Networks) to help neural networks train more effectively. The key idea is to bypass or "skip" certain layers by creating shortcuts from one layer to another. This helps mitigate problems like the **vanishing gradient**, allowing gradients to flow more easily during backpropagation.

- In the Transformer, a skip connection is added around each **sublayer** (Self-Attention and Feed Forward layers). This means that the output of a sublayer is added to its input, allowing the network to retain the original information while also learning new transformations.
- **Mathematically**:

$$\text{LayerNorm}(x + \text{Sublayer}(x))$$

Press enter or click to view image in full size

Where:

- x is the input to the sublayer.
- $\text{Sublayer}(x)$ is the output of the sublayer (either Self-Attention or Feed Forward).
- The addition $x + \text{Sublayer}(x)$ is the skip connection.

2. Layer Normalization:

- After adding the skip connection, **Layer Normalization** is applied. Unlike **Batch Normalization**, which normalizes across a batch of data, **Layer Normalization** normalizes across the features of a single data point.
- This is particularly useful in Transformers, where sequences can vary in length and are processed in parallel. Layer normalization helps stabilize the network by ensuring the inputs to each layer have a consistent distribution, which improves convergence during training.

3. Where and How are They Applied?

- In the Transformer model, **Skip Connection** and **Layer Normalization** are applied **twice** in each encoder and decoder block:

1. **After the Self-Attention layer**:

a. The input to the self-attention layer is added to its output, followed by layer normalization.

b. Formula:

$$\text{LayerNorm}(x + \text{Self-Attention}(x))$$

2. After the Feed Forward layer:

a. The output of the self-attention block (after skip connection and normalization) is passed to the feed-forward layer, and then the same process is repeated.

b. Formula:

$$\text{LayerNorm}(x + \text{FeedForward}(x))$$

4. Why Use Skip Connections and Layer Normalization?

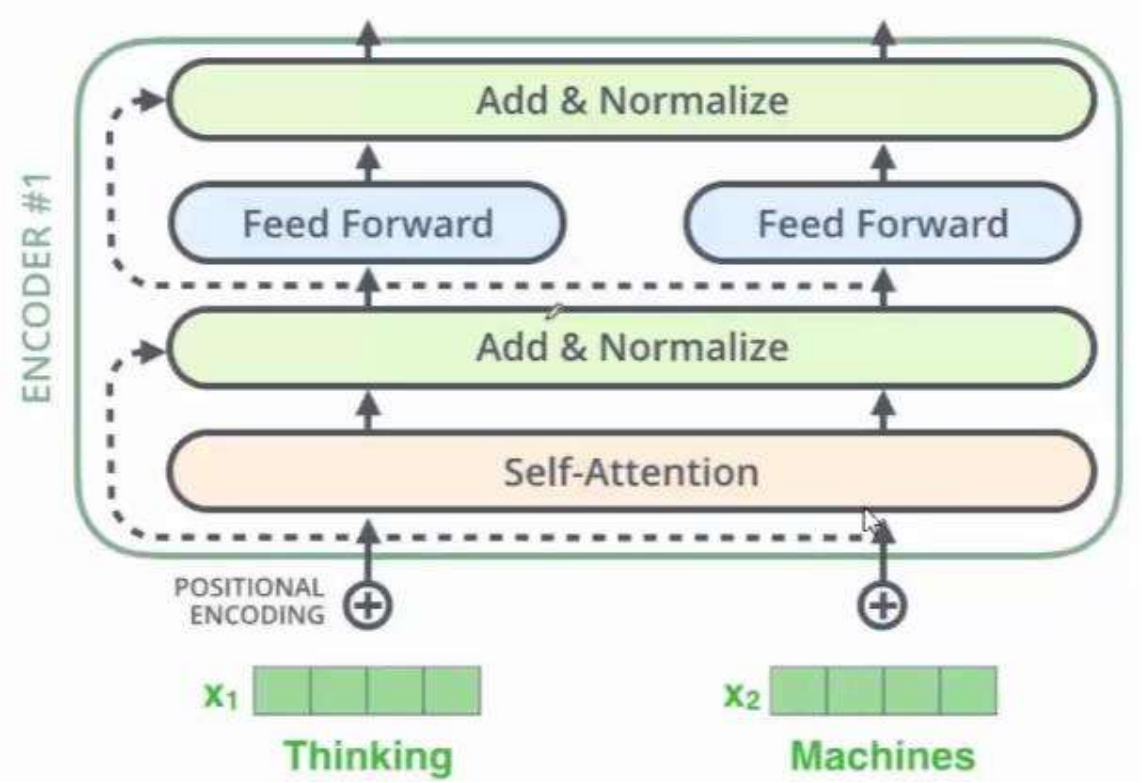
1.Skip Connections:

- Help maintain the flow of gradients during training, preventing the vanishing gradient problem.
- Allow the model to learn additional complex patterns without losing the original information.

2.Layer Normalization:

- Ensures stable learning by normalizing the output of each sublayer.
- Improves training speed and overall model performance by keeping the network's activations in a stable range.

Visual Explanation:



In the provided diagram, you can see how the "Add & Normalize" blocks are placed:

- After the **Self-Attention** layer.
- After the **Feed Forward** layer.

The dashed lines represent the skip connections, which add the original input to the output of each sublayer before normalization.

Layer Normalization vs. Batch Normalization

In Transformers, **Layer Normalization** is used instead of **Batch Normalization**. Layer Normalization helps stabilize the training process by normalizing the activations across the features of a single data point, rather than across the batch.

1. What is Layer Normalization?

Layer Normalization normalizes the activations of the neurons in each layer for a single training example. This means it considers all the features of a single data point and normalizes them to have a mean of 0 and a standard deviation of 1.

Mathematical Formula:

Given a layer with activations a_i for a single training example, Layer Normalization is calculated as:

$$\mu^t = \frac{1}{H}\sum_{i-1}^{H} a_i^t$$

$$\sigma^t = \sqrt{\frac{1}{H}\sum_{i=1}^{H}(a_i^t - \mu^t)^2}$$

$$h^t = f\left(\frac{g}{\sigma^t} \odot (a^t - \mu^t) + b\right)$$

Where:
- μ^t is the mean of activations for the layer,
- σ^t is the standard deviation,
- H is the number of features,
- g and b are learnable parameters for scaling and shifting,
- f is an activation function,
- a^t is the activation vector for the current layer.

2. Layer Normalization vs. Batch Normalization

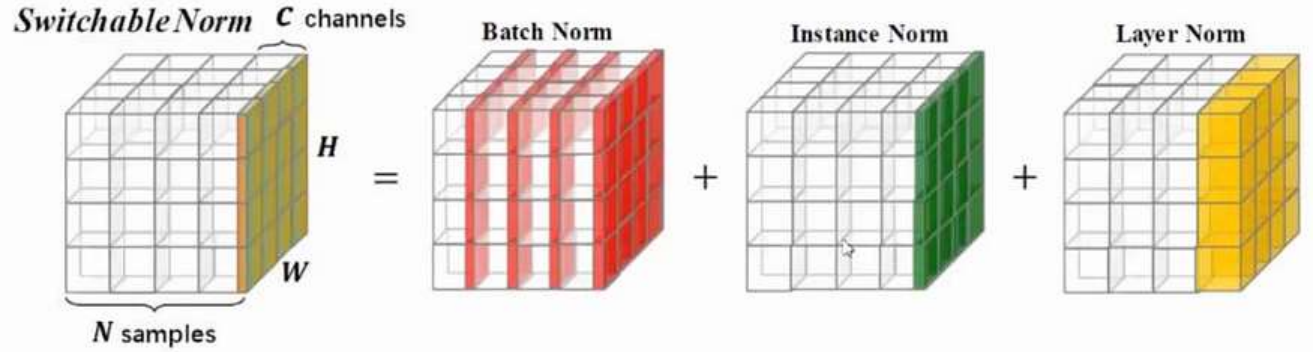
- **Batch Normalization** normalizes the activations across the batch dimension, meaning it depends on the entire batch to calculate the mean and standard deviation.
- **Layer Normalization**, on the other hand, **only considers the features of a single data point**. It normalizes across the feature dimension, ignoring other samples in the batch. This makes it more suitable for models like Transformers, where sequence lengths can vary, and batch size might be small.

3. Why Use Layer Normalization in Transformers?

- **Independence from Batch Size**: Unlike Batch Normalization, which depends on the batch size, Layer Normalization works well regardless of batch size, making it more suitable for sequential models where the batch size can vary.
- **Handles Variable Sequence Lengths**: In NLP tasks, sequences can vary in length. Layer Normalization can handle these variations more gracefully because it operates on a single data point.
- **Stabilizes Training**: It normalizes the output of each layer, ensuring that the activations remain within a reasonable range, which helps stabilize the training process and speed up convergence.

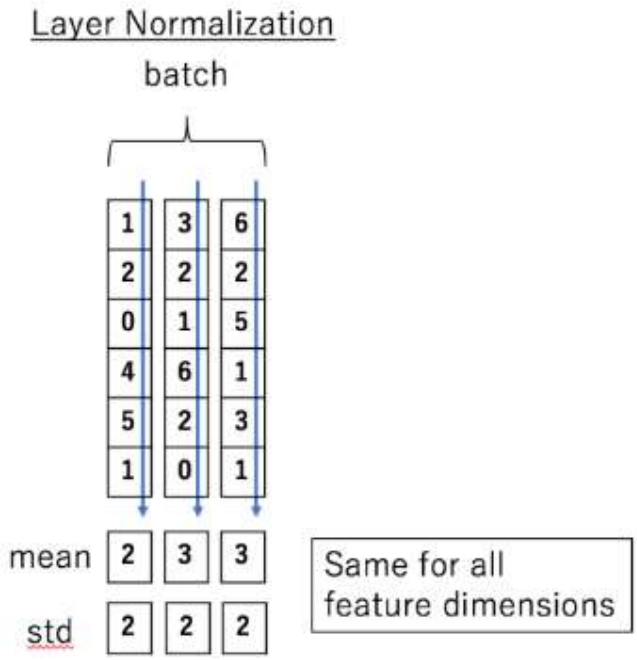
4. Visualization

Press enter or click to view image in full size



In the images:

- **Batch Norm** (red block): Normalizes across the batch for each feature dimension.
- **Instance Norm** (green block): Normalizes across spatial dimensions.
- **Layer Norm** (yellow block): Normalizes across all features of a single data point.



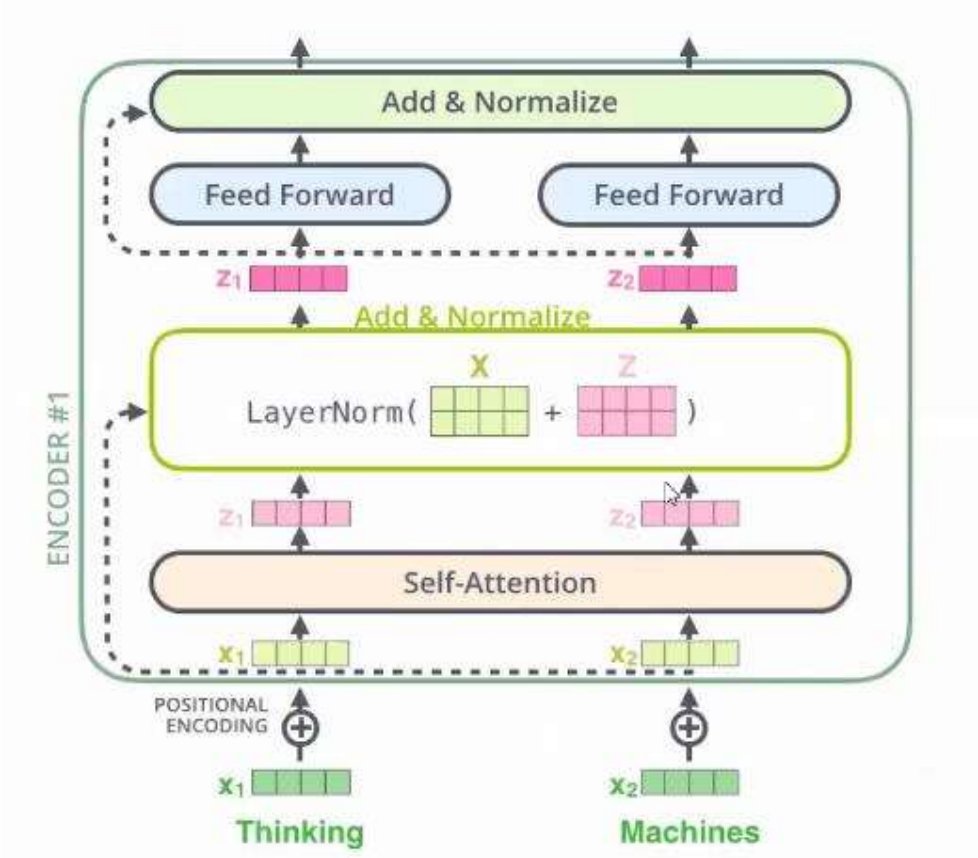
In the example shown, Layer Normalization calculates the mean and standard deviation across **each row (representing a single training example)**, and then normalizes the activations for that row.

Summary:

- **Layer Normalization** normalizes across the features of each individual training example, making it independent of the batch size.
- This is crucial for models like Transformers, where sequences have varying lengths and the batch size might not be large.
- **Formula**: It scales the activations to have a mean of 0 and a standard deviation of 1 using the formulas provided above.

This normalization helps the model train more effectively by keeping the activations stable throughout the layers.

Encoder Module Summary



The diagram provides an overview of the key components and workflow of an **Encoder** block in the Transformer model. Here's a step-by-step breakdown:

1. Input Embeddings and Positional Encoding

- Each input word (e.g., "Thinking" and "Machines") is first converted into a dense vector representation, called an **embedding**.
- **Positional encoding** is then added to these embeddings to provide information about the position of each word in the sentence. This is essential because Transformers process all words simultaneously and need positional information to understand the order.

2. Self-Attention Mechanism

- The input embeddings with positional encodings are fed into the **Self-Attention** layer. This mechanism allows the model to focus on different words in the sentence while processing each word, capturing contextual relationships.
- After processing through self-attention, the output for each word (denoted as z_1 and z_2) captures the contextual meaning considering the entire sentence.

3. Add & Normalize (First Layer)

- The output of the Self-Attention layer is passed to the first **Add & Normalize** step:
- **Add**: A **skip connection** (or residual connection) is applied where the original input (embeddings) is added to the output of the self-attention layer.
- **Normalize**: **Layer Normalization** is applied to the summed result to ensure stable learning by normalizing the output of each laye

$$\text{LayerNorm}(X + \text{Self-Attention}(X))$$

4. Feed Forward Network

- The normalized output is then passed through a **Feed Forward Network (FFN)**. The FFN is applied independently to each position (word) in the sequence. It consists of two linear transformations with a ReLU activation in between:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

Where:
- W_1, W_2 and b_1, b_2 are learnable parameters.

5. Add & Normalize (Second Layer)

- After the FFN, another **Add & Normalize** step is performed:
- **Add**: A skip connection adds the output of the first Add & Normalize layer to the output of the FFN.
- **Normalize**: The result is again normalized using Layer Normalization.

The formula for this step is:

$$\text{LayerNorm}(X + \text{FFN}(X))$$

6. Output of the Encoder Layer

- The final output of the encoder layer, after both the self-attention and feed-forward steps, is a set of vectors (z_1, z_2, …) that represent each word in its enriched contextualized form.

Summary

- **Self-Attention**: Captures the relationships between words in the sentence.
- **Add & Normalize**: Stabilizes training by normalizing the outputs of the self-attention and feed-forward layers while preserving the input information through skip connections.
- **Feed Forward**: Provides a non-linear transformation to enhance the expressiveness of the model.

The encoder block can be stacked multiple times to form a deep Transformer model, allowing it to capture complex language patterns and relationships.

Decoder Module

Let's now examine the Decoder module. In the Transformer architecture, the Decoder is also composed of **N = 6 layers**, each stacked with sublayers. There are three primary differences between the Encoder and Decoder:

1. **Masked Multi-Headed Attention in Sublayer 1**

The first sublayer in the Decoder uses a **Masked Multi-Headed Attention** mechanism. This masking prevents the model from accessing subsequent positions in the target sequence during training, thus avoiding information leakage.

The attention calculation is modified by adding a mask M to the scores:

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

Here, M is a mask matrix where positions corresponding to illegal connections (future positions) are set to $-\infty$, so after the softmax, these positions have zero probability.

2. Encoder-Decoder Attention in Sublayer 2

The second sublayer is an **Encoder-Decoder Multi-Headed Attention**. This allows the Decoder to attend over the output of the Encoder stack. In this sublayer:

- The queries Q come from the previous Decoder layer.
- The keys K and values V come from the output of the Encoder.

The attention mechanism enables the Decoder to focus on relevant parts of the input sequence.

3. Position-wise Feed-Forward Network in Sublayer 3

The third sublayer is a standard **Position-wise Feed-Forward Network** (FFN), applied independently to each position:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

This introduces non-linearity and further maps the input to a desired output dimension.

After processing through these sublayers, a **Linear Layer** and a **Softmax Layer** are applied to the output to generate the probabilities for the next word in the sequence:

$$\text{Probabilities} = \text{softmax}(xW_O + b_O)$$

Here, W_o and b_o are the weights and biases of the Linear layer, and x is the output from the last sublayer.

Summary of Key Formulas

1. **Masked Multi-Headed Attention:**

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

2. Encoder-Decoder Attention Components:

- Queries from Decoder:

$$Q = \text{Decoder Output} \times W_Q \quad Q = \text{Decoder Output} \times W_Q$$

- Keys and Values from Encoder:

$$K = \text{Encoder Output} \times W_K \quad K = \text{Encoder Output} \times W_K$$
$$V = \text{Encoder Output} \times W_V \quad V = \text{Encoder Output} \times W_V$$

3. Position-wise Feed-Forward Network:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

4. Output Probabilities:

$$\text{Probabilities} = \text{softmax}(xW_O + b_O)$$

These components work together to allow the Decoder to generate the output sequence by attending to both the previously generated tokens and the encoded representations of the input sequence.

Decoder's Masked Multi-Head Attention Input During Training

In the training phase of the Transformer model, the **Decoder's initial input** is the target sequence Y from the training data. However, this input is **shifted to the right by one position**, a process known as **"Shifted Right"**. This shift ensures that at each time step, the model predicts the next word using only the previous words, without peeking ahead.

Why Shift the Target Sequence Right?

- Prediction Alignment: At time step T, the model aims to predict the word Y_t.
- Preventing Leakage: To avoid the model seeing the actual Y_t during prediction, we provide it with Y_{t-1} as input.
- Sequential Learning: Shifting the sequence right allows the model to learn dependencies by using previous outputs to predict the next token.

Illustrative Example: Translating "我爱中国" to "I Love China"

Before Shift (Original Target Sequence):

Position	Token
0	"I"
1	"Love"
2	"China"

After Shifted Right (Input to Decoder):

Position	Token
0	</s> (Start Token)
1	"I"
2	"Love"
3	"China"

Explanation:

- We insert a start token </s> at the beginning.
- Each token is moved one position to the right.
- This setup ensures that the model uses the start token to predict “I”, uses “I” to predict “Love”, and so on.

Step-by-Step Processing:

Time Step 1

- **Decoder Input:** Start Token </s> **+ Positional Encoding**
- **Encoder Output:** Encoded representation of “**我**爱中国”
- **Decoder Output:** Predicts **“I”**

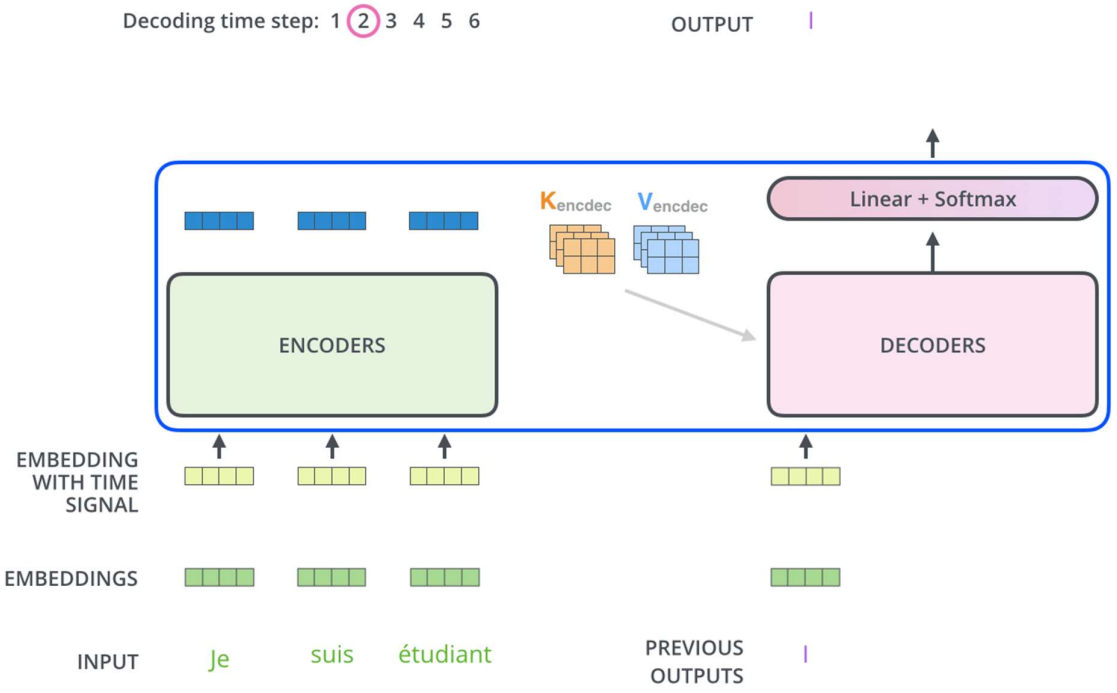
Time Step 2

- **Decoder Input:** </s> + “I” **+ Positional Encoding**
- **Encoder Output:** Encoded representation of “**我**爱中国”
- **Decoder Output:** Predicts **“Love”**

Time Step 3

- **Decoder Input:** </s> + “I” + “Love” **+ Positional Encoding**
- **Encoder Output:** Encoded representation of “**我**爱中国”
- **Decoder Output:** Predicts **“China”**

Press enter or click to view image in full size



Key Components:

1. **Shifted Input Sequence:**

- The Decoder's input at each time step T is:

Press enter or click to view image in full size

$$\text{Input}_T = [< /s >, Y_0, Y_1, \ldots, Y_{T-1}]$$

- This sequence includes all tokens up to Y_{t-1}.

2. Masked Multi-Head Attention:

- The Decoder uses a mask to prevent attention to future tokens.
- **Attention Formula with Mask:**

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

- Q, K, and V are derived from the Decoder's input.
- M is the mask matrix where:

$$M_{i,j} = \begin{cases} 0, & \text{if } j \leq i \\ -\infty, & \text{if } j > i \end{cases}$$

- This mask ensures that position i cannot attend to positions greater than i (future positions).

3. Positional Encoding:

- Adds position information to the tokens.

- Helps the model understand the order of the sequence.

Why Masking is Necessary:

- **Autoregressive Modeling:** Ensures the model predicts the next word based only on previous words.
- **Preventing Future Information:** Stops the model from using information from future tokens during training, which would be unrealistic during actual generation.

Summary:

- **Shifted Right Input:** Aligns the Decoder's input so it only contains known tokens up to the current position.
- **Masking Mechanism:** Prevents the model from considering future tokens in the sequence.
- **Process Flow:**

1. **Input Preparation:** Shift the target sequence right and add positional encoding.
2. **Encoding Source Sequence:** The Encoder processes the source sequence ("我爱中国") and generates embeddings.
3. **Decoding with Attention:**

- At each time step, the Decoder uses the shifted input and applies masked multi-head attention.

- It attends to the relevant parts of the Encoder's output to generate the next word.

- **Outcome:** The model learns to generate the target sequence one word at a time, using only the information available up to that point.

Understanding Masking in Transformer Models

In Transformer models, **masking** plays a crucial role in controlling the flow of information and ensuring that the model processes sequences appropriately. There are two primary types of masks used:

1. **Padding Mask**
2. **Sequence Mask**

These masks help the model handle variable-length sequences and prevent it from "cheating" by looking ahead at future tokens during training.

1. Padding Mask

What is a Padding Mask?

In natural language processing tasks, sequences (sentences) within a batch can have varying lengths. To process these sequences in parallel, they need to be padded to the same length. This involves:

- **Shorter sequences**: Add padding tokens (usually zeros) to match the length of the longest sequence.
- **Longer sequences**: Truncate to the maximum allowed length.

However, the padding tokens do not carry meaningful information and should not influence the model's predictions. The **Padding Mask** is used to prevent the model from attending to these padded positions.

How is the Padding Mask Applied?

- The Padding Mask is a tensor with boolean values indicating which positions are padding (usually True for padding positions and False for valid tokens).

- During the **Scaled Dot-Product Attention** calculation, the mask is added to the attention scores.
- Positions corresponding to padding tokens are set to a large negative value (e.g., $-\infty$), so after applying the softmax, their attention weights become negligible (approaching zero).

Mathematical Representation:

Let's denote:

- Q : Query matrix
- K : Key matrix
- V : Value matrix
- M_{padding} : Padding mask matrix

The attention mechanism with the padding mask is calculated as:

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

Where:

- M_{padding} adds a large negative number to the positions of padding tokens.

Example:

Suppose we have a batch of sequences padded to a length of 6:

- **Sequence 1**: ["I", "love", "NLP", **PAD**, **PAD**, **PAD**]
- **Sequence 2**: ["Transformers", "are", "great", "models", **PAD**, **PAD**]

The Padding Mask for these sequences would be:

- **Sequence 1 Mask**: [0, 0, 0, 1, 1, 1]
- **Sequence 2 Mask**: [0, 0, 0, 0, 1, 1]

Here, 1 indicates a padding position.

2. Sequence Mask

What is a Sequence Mask?

The **Sequence Mask** (also known as the **Look-Ahead Mask**) is used in the Decoder's Self-Attention mechanism to prevent the model from accessing future tokens during training. This ensures that the prediction at time step t depends only on the known outputs at time steps less than or equal to t .

Why is the Sequence Mask Necessary?

- **Autoregressive Property**: In sequence generation tasks, the model should only consider past and present tokens to predict the next token.
- **Prevent Information Leakage**: Without masking, the model could "see" future tokens, which would make training unrealistic and compromise the model's ability to generate coherent sequences during inference.

How is the Sequence Mask Applied?

- The Sequence Mask is an upper triangular matrix (often represented as a lower triangular matrix with ones on the diagonal and below).
- It masks out (sets to zero) the attention weights for future positions.
- Combined with the padding mask if necessary.

Mathematical Representation:

Press enter or click to view image in full size

Let's denote:

- M_{sequence} : Sequence mask matrix

The attention mechanism with the sequence mask is calculated as:

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

Press enter or click to view image in full size

Where:

- M_{sequence} is structured to mask out future positions.

Sequence Mask Matrix Example:

Press enter or click to view image in full size

For a sequence length of 5, the sequence mask M_{sequence} would be:

$$M_{\text{sequence}} = \begin{bmatrix} 0 & -\infty & -\infty & -\infty & -\infty \\ 0 & 0 & -\infty & -\infty & -\infty \\ 0 & 0 & 0 & -\infty & -\infty \\ 0 & 0 & 0 & 0 & -\infty \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

- **Explanation**: At time step t positions greater than t are masked (set to $-\infty$).

Combining Padding Mask and Sequence Mask:

In the Decoder's Self-Attention, both masks are combined:

$$M_{\text{total}} = M_{\text{padding}} + M_{\text{sequence}}$$

This ensures that the model neither attends to padding positions nor future tokens.

Example Scenario

Assume:

- **Maximum sequence length**: 10
- **Padding Mask**: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
- **Current sentence length**: 5 tokens (including a start token)

At Time Step 3 (Predicting the 3rd Token):

- **Decoder Input**: Start token + First two tokens
- **Sequence Mask**: [1, 1, 1, 0, 0, 0, 0, 0, 0, 0]

Combining Masks:

- **Combined Mask**:

$$M_{\text{total}} = M_{\text{padding}} + M_{\text{sequence}} = [1, 1, 1, 0, 0, 0, 0, 0, 0, 0]$$

- Positions beyond the current token are masked out.

Summary

- **Padding Mask**:
- Used to ignore padding tokens in sequences.
- Applied in all attention mechanisms (both Encoder and Decoder).
- Masks out positions that are padding tokens.
- **Sequence Mask**:
- Used to prevent the model from looking ahead in the sequence.
- Applied only in the Decoder's Self-Attention.
- Masks out future positions beyond the current time step.

Key Points:

- Masks are essential for proper sequence modeling in Transformers.
- The combination of masks ensures that the model focuses on relevant information without being influenced by padded or future tokens.
- By applying these masks, the model can effectively learn dependencies and generate coherent sequences during inference.

Example of Using BERT Inputs in Code

Here's how you might use these inputs with a BERT model from the Hugging Face Transformers library:

Annotated Explanation of BERT Code Example

Below is a concise explanation of how to use BERT inputs with the Hugging Face Transformers library, along with brief annotations.

```
from transformers import BertTokenizer, BertModel
import torch

# Initialize the tokenizer and model with pre-trained weights
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')  # Loads the tokenizer
model = BertModel.from_pretrained('bert-base-uncased')          # Loads the BERT model

# Sample input text
text = "Hello, how are you?"
```

```
# Tokenize the input text
encoding = tokenizer(
    text,
    return_tensors='pt',   # Returns PyTorch tensors
    padding=True,          # Pads sequences to the same length
    truncation=True,       # Truncates sequences longer than max_length
    max_length=128         # Sets maximum length of sequences
)

# Extract inputs for the model
input_ids = encoding['input_ids']              # Token IDs of the input text
attention_mask = encoding['attention_mask']    # Mask to avoid attending to padding tokens
token_type_ids = encoding.get('token_type_ids') # Segment token indices (useful for sentence pairs)

# Forward pass through the model
outputs = model(
    input_ids=input_ids,
    attention_mask=attention_mask,
    token_type_ids=token_type_ids
)

# Get the last hidden state
last_hidden_state = outputs.last_hidden_state   # Tensor of shape [batch_size, sequence_length, hidden_size]
```

Brief Explanation:

- **Import Libraries:**
- **BertTokenizer**: Converts text to token IDs.
- **BertModel**: Pre-trained BERT model without any task-specific heads.
- **torch**: PyTorch library for tensor operations.
- **Initialize Tokenizer and Model:**
- Load pre-trained tokenizer and model weights using 'bert-base-uncased'.
- **Sample Input Text:**
- Define the text you want to process with BERT.
- **Tokenization:**
- **tokenizer()**: Converts text to token IDs and creates attention masks.
- **return_tensors='pt'**: Outputs PyTorch tensors.
- **padding=True**: Pads sequences to the same length.
- **truncation=True**: Truncates sequences longer than max_length.
- **max_length=128**: Sets maximum sequence length.
- **Extract Inputs:**
- **input_ids**: Tensor containing token IDs.
- **attention_mask**: Tensor indicating real tokens (1) vs. padding tokens (0).

- **token_type_ids**: Identifies different sequences in tasks involving sentence pairs.
- **Model Forward Pass:**
- Pass the inputs to the BERT model to obtain outputs.
- The model computes embeddings for each token, considering the context provided by the entire sequence.
- **Retrieve Outputs:**
- **last_hidden_state**: Contains contextualized embeddings for each token in the input sequence.
- Shape: [batch_size, sequence_length, hidden_size].
- Useful for downstream tasks like classification, question answering, etc.

Key Points:

- **Tokenization Process:**
- Converts text to tokens, adds special tokens [CLS] and [SEP], and maps tokens to IDs.
- Handles padding and truncation to ensure consistent input lengths.
- **Attention Mask:**
- Ensures that the model does not focus on padding tokens during attention computations.
- **Model Outputs:**
- **last_hidden_state**: Main output for further processing.
- Other outputs may include pooled outputs or hidden states from intermediate layers.
- **Usage Example:**
- For a single sentence, token_type_ids may not be necessary.
- In tasks like sentence pair classification, token_type_ids help differentiate between the two sentences.

Decoder's Encoder-Decoder Attention Layer

In the Transformer architecture, the decoder generates the output sequence by attending to both its previous outputs and the encoded representations of the input sequence. The **encoder-decoder attention** mechanism within the decoder is crucial for allowing the decoder to focus on relevant parts of the input when producing each token.

What is Encoder-Decoder Attention?

The encoder-decoder attention layer in the decoder enables the model to align and extract information from the encoder's output. This mechanism helps the decoder decide which parts of the input sequence are most relevant to the current decoding step.

How Does It Work?

- Queries (Q): Generated from the decoder's previous layer output.
- Key (K) and Value (V): Obtained from the encoder's output.

By using the decoder's state to query the encoder's outputs, the model can focus on specific positions in the input sequence that are most relevant to generating the next output token.

Mathematical Formulation

1. **Compute Queries, Keys, and Values:**

Press enter or click to view image in full size

$$Q = X_{\text{decoder}} W_Q$$
$$K = X_{\text{encoder}} W_K$$
$$V = X_{\text{encoder}} W_V$$

- X_{decoder} : Input to the decoder layer (output from the previous decoder layer).
- X_{encoder} : Output from the encoder.
- W_Q , W_K , W_V : Weight matrices.

2. Scaled Dot-Product Attention:

Press enter or click to view image in full size

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

- d_k : Dimensionality of the keys (used for scaling).

Explanation of the encoder_decoder_attention **Function**

The encoder_decoder_attention function is a crucial component in the Transformer architecture's decoder. It allows the decoder to attend to the encoder's outputs, effectively enabling the model to focus on relevant parts of the input sequence when generating each token in the output sequence.

Below, I'll explain the encoder_decoder_attention function from a code perspective, providing detailed annotations to help you understand how it works.

1. Purpose of encoder_decoder_attention

- **Functionality**: Allows the decoder to incorporate information from the encoder's output.
- **Role in Transformer**: Bridges the gap between the encoder and decoder by enabling the decoder to focus on relevant parts of the input sequence.

2. Code Implementation

I'll provide a simplified implementation of the encoder_decoder_attention function using PyTorch, along with detailed explanations.

```
import torch
import torch.nn as nn
import torch.nn.functional as F

def encoder_decoder_attention(query, key, value, mask=None, num_heads=8):
    """
    Performs multi-head encoder-decoder attention.

    Args:
```

```
        query: Tensor of shape (batch_size, tgt_seq_len, d_model)
            Decoder's input embeddings or outputs from the previous decoder layer.
        key:   Tensor of shape (batch_size, src_seq_len, d_model)
            Encoder's output embeddings.
        value: Tensor of shape (batch_size, src_seq_len, d_model)
            Same as key (in encoder-decoder attention).
        mask:  Optional tensor for masking (batch_size, 1, 1, src_seq_len)
        num_heads: Number of attention heads.

    Returns:
        Output tensor of shape (batch_size, tgt_seq_len, d_model)
    """

    d_model = query.size(-1)
    assert d_model % num_heads == 0, "d_model must be divisible by num_heads"
    d_k = d_model // num_heads  # Dimension per head

    batch_size = query.size(0)

    # Linear projections
    W_q = nn.Linear(d_model, d_model)
    W_k = nn.Linear(d_model, d_model)
    W_v = nn.Linear(d_model, d_model)
    W_o = nn.Linear(d_model, d_model)

    # Apply linear projections to query, key, and value
    Q = W_q(query)  # (batch_size, tgt_seq_len, d_model)
    K = W_k(key)    # (batch_size, src_seq_len, d_model)
    V = W_v(value)  # (batch_size, src_seq_len, d_model)

    # Split into num_heads heads and reshape
    Q = Q.view(batch_size, -1, num_heads, d_k).transpose(1, 2)  # (batch_size, num_heads, tgt_seq_len, d_k)
    K = K.view(batch_size, -1, num_heads, d_k).transpose(1, 2)  # (batch_size, num_heads, src_seq_len, d_k)
    V = V.view(batch_size, -1, num_heads, d_k).transpose(1, 2)  # (batch_size, num_heads, src_seq_len, d_k)

    # Scaled dot-product attention
    # Compute attention scores
    scores = torch.matmul(Q, K.transpose(-2, -1)) / torch.sqrt(torch.tensor(d_k, dtype=torch.float32))
    # Apply mask if provided
    if mask is not None:
        scores = scores.masked_fill(mask == 0, float('-inf'))

    # Calculate attention weights
    attention_weights = F.softmax(scores, dim=-1)  # (batch_size, num_heads, tgt_seq_len, src_seq_len)

    # Compute attention output
    attention_output = torch.matmul(attention_weights, V)  # (batch_size, num_heads, tgt_seq_len, d_k)

    # Concatenate heads
    attention_output = attention_output.transpose(1, 2).contiguous().view(batch_size, -1, d_model)  #
(batch_size, tgt_seq_len, d_model)
```

```
    # Final linear layer
    output = W_o(attention_output)  # (batch_size, tgt_seq_len, d_model)

    return output
```

3. Detailed Explanation

Imports and Function Definition

```
import torch
import torch.nn as nn
import torch.nn.functional as F

def encoder_decoder_attention(query, key, value, mask=None, num_heads=8):
    # Function body...
```

Imports:

- torch: PyTorch library for tensor operations.
- torch.nn: Contains neural network layers.
- torch.nn.functional: Contains functions for neural network operations, such as activation functions.

Function Arguments:

- query: The decoder's input (shape: (batch_size, tgt_seq_len, d_model)).
- key and value: The encoder's output (shape: (batch_size, src_seq_len, d_model)).
- mask: Optional mask tensor to prevent attention to certain positions.
- num_heads: Number of attention heads (default is 8).

Parameter Initialization

```
d_model = query.size(-1)
assert d_model % num_heads == 0, "d_model must be divisible by num_heads"
d_k = d_model // num_heads  # Dimension per head

batch_size = query.size(0)
```

- **d_model**: The dimensionality of the model (embedding size).
- **d_k**: The dimensionality of each attention head.
- **batch_size**: Number of samples in the batch.

Linear Projection Layers

```
# Linear projections
W_q = nn.Linear(d_model, d_model)
W_k = nn.Linear(d_model, d_model)
W_v = nn.Linear(d_model, d_model)
W_o = nn.Linear(d_model, d_model)
```

- **Purpose**: To project the query, key, and value tensors into a higher-dimensional space.
- **Note**: In practice, these layers are usually defined in the __init__ method of a class, but for simplicity, they are included here.

Applying Linear Projections

```
# Apply linear projections to query, key, and value
Q = W_q(query)  # (batch_size, tgt_seq_len, d_model)
K = W_k(key)    # (batch_size, src_seq_len, d_model)
V = W_v(value)  # (batch_size, src_seq_len, d_model)
```

- **Q**, K, V: The projected query, key, and value tensors.

Reshaping for Multi-Head Attention

```
# Split into num_heads and reshape
Q = Q.view(batch_size, -1, num_heads, d_k).transpose(1, 2)  # (batch_size, num_heads, tgt_seq_len, d_k)
K = K.view(batch_size, -1, num_heads, d_k).transpose(1, 2)  # (batch_size, num_heads, src_seq_len, d_k)
V = V.view(batch_size, -1, num_heads, d_k).transpose(1, 2)  # (batch_size, num_heads, src_seq_len, d_k)
```

- **Purpose**: Split the tensors into multiple heads to allow the model to attend to information from different representation subspaces.
- **transpose(1, 2)**: Swaps the dimensions to bring num_heads to the second dimension.

Computing Scaled Dot-Product Attention

```
# Compute attention scores
scores = torch.matmul(Q, K.transpose(-2, -1)) / torch.sqrt(torch.tensor(d_k, dtype=torch.float32))
```

- **scores**: Represents the compatibility of the queries with the keys.
- **Scaling**: Dividing by sqrt(d_k) stabilizes gradients.

Applying Mask (if provided)

```
# Apply mask if provided
if mask is not None:
    scores = scores.masked_fill(mask == 0, float('-inf'))
```

- **Purpose**: Prevents attention to certain positions (e.g., padding tokens).
- **masked_fill**: Replaces positions where mask == 0 with negative infinity, so after applying softmax, the attention weights at these positions become zero.

Calculating Attention Weights

```
# Calculate attention weights
attention_weights = F.softmax(scores, dim=-1)  # (batch_size, num_heads, tgt_seq_len, src_seq_len)
```

- **attention_weights**: Probabilities that determine how much each key/value pair contributes to the output.

Computing Attention Output

```
# Compute attention output
attention_output = torch.matmul(attention_weights, V)  # (batch_size, num_heads, tgt_seq_len, d_k)
```

- **Purpose**: Aggregates the values weighted by the attention weights.

Concatenating Heads

```
# Concatenate heads
attention_output = attention_output.transpose(1, 2).contiguous().view(batch_size, -1, d_model)  #
(batch_size, tgt_seq_len, d_model)
```

- **transpose(1, 2)**: Swaps back the dimensions to prepare for concatenation.
- **contiguous()**: Ensures the tensor is stored contiguously in memory.
- **view()**: Reshapes the tensor to combine the multiple heads back into a single vector.

Final Linear Layer

```
# Final linear layer
output = W_o(attention_output)  # (batch_size, tgt_seq_len, d_model)
```

- **Purpose**: Projects the concatenated output back to the desired dimensionality.

Conclusion

The encoder_decoder_attention function is pivotal in enabling the decoder to utilize the encoder's outputs effectively. By computing attention scores between the decoder's queries and the encoder's keys, the model can focus on relevant parts of the input sequence when generating each output token.

Understanding this function and its implementation is crucial for grasping how the Transformer model achieves state-of-the-art performance in tasks like machine translation, text summarization, and more.

Output of the Decoder in the Transformer

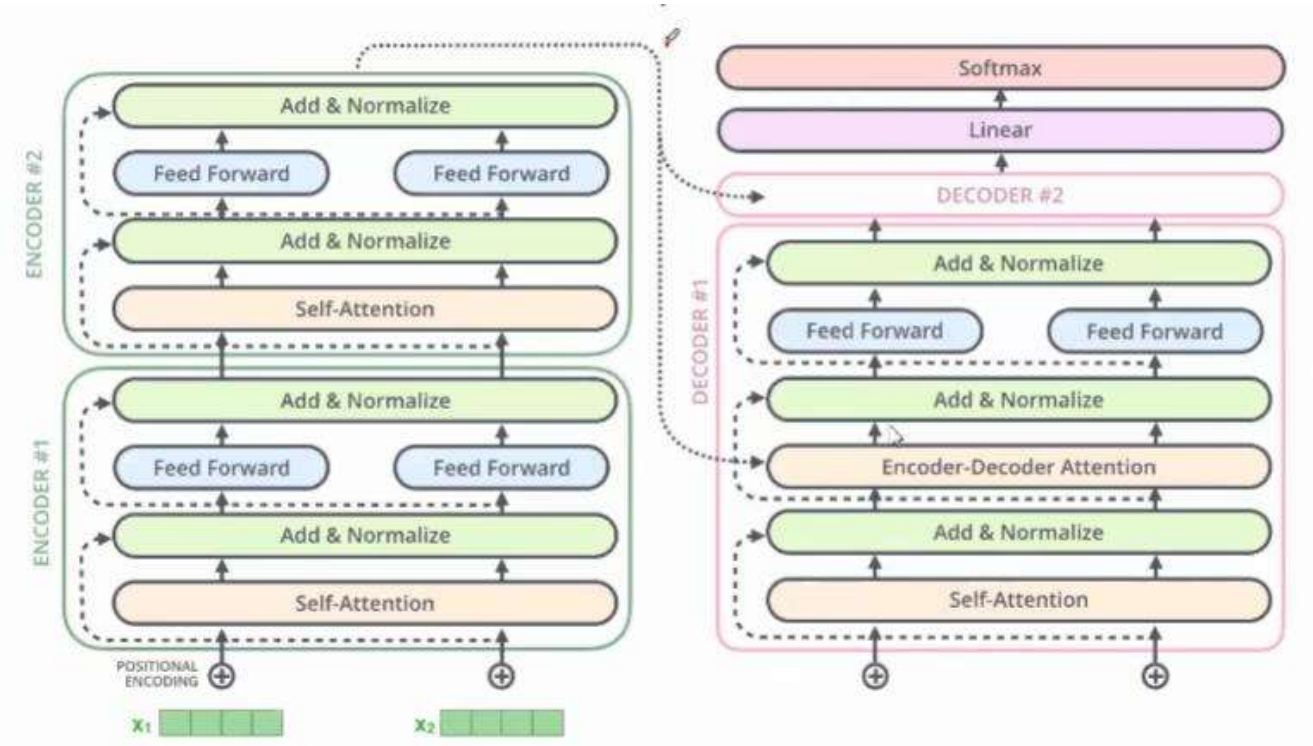
The diagram shows the structure of the **Decoder** in the Transformer model, which is similar to the **Encoder** but with some key differences. The Decoder is responsible for generating the output sequence (e.g., a translated sentence) based on the encoded input sequence from the Encoder.

Key Differences Between Decoder and Encoder:

1. **Additional Layer**: The Decoder has an extra layer called the **Encoder-Decoder Attention** layer. This layer allows the Decoder to focus on relevant parts of the input sequence (encoded by the Encoder) when generating each word of the output.
2. **Final Linear + Softmax Layer**: At the end of the Decoder, there is a **Linear layer** followed by a **Softmax layer**. This layer maps the Decoder's output to a vocabulary-sized vector, predicting the next word in the sequence.

How the Decoder Works Step-by-Step:

Press enter or click to view image in full size



1. Input Embeddings and Positional Encoding:

- The Decoder takes the embedding of the entire input sentence and a special start token $$ $$ as input. The positional encoding is added to these embeddings to provide information about the order of the words.

2. Self-Attention Layer:

- The **Self-Attention** layer in the Decoder allows the model to focus on different parts of the output sequence generated so far.
- This helps the Decoder decide what to generate next based on the already generated part of the output.

3. Encoder-Decoder Attention Layer:

- This layer performs attention over the Encoder's output, allowing the Decoder to focus on relevant parts of the input sentence.
- For each word the Decoder generates, it can "look back" at the input sentence (through the attention mechanism) to find the most relevant information.

4. Add & Normalize:

- Like the Encoder, the Decoder also uses **Skip Connections** and **Layer Normalization** after each attention and feed-forward layer to stabilize training.

5. Feed Forward Layer:

- Similar to the Encoder, this layer applies a non-linear transformation independently to each word position, further processing the information.

6. Final Linear + Softmax Layer:

- The output of the final feed-forward layer is passed through a **Linear** layer that expands the output to match the size of the vocabulary.

- The **Softmax** function is applied to this output to convert it into a probability distribution over the vocabulary. The word with the highest probability is chosen as the next word in the sequence.

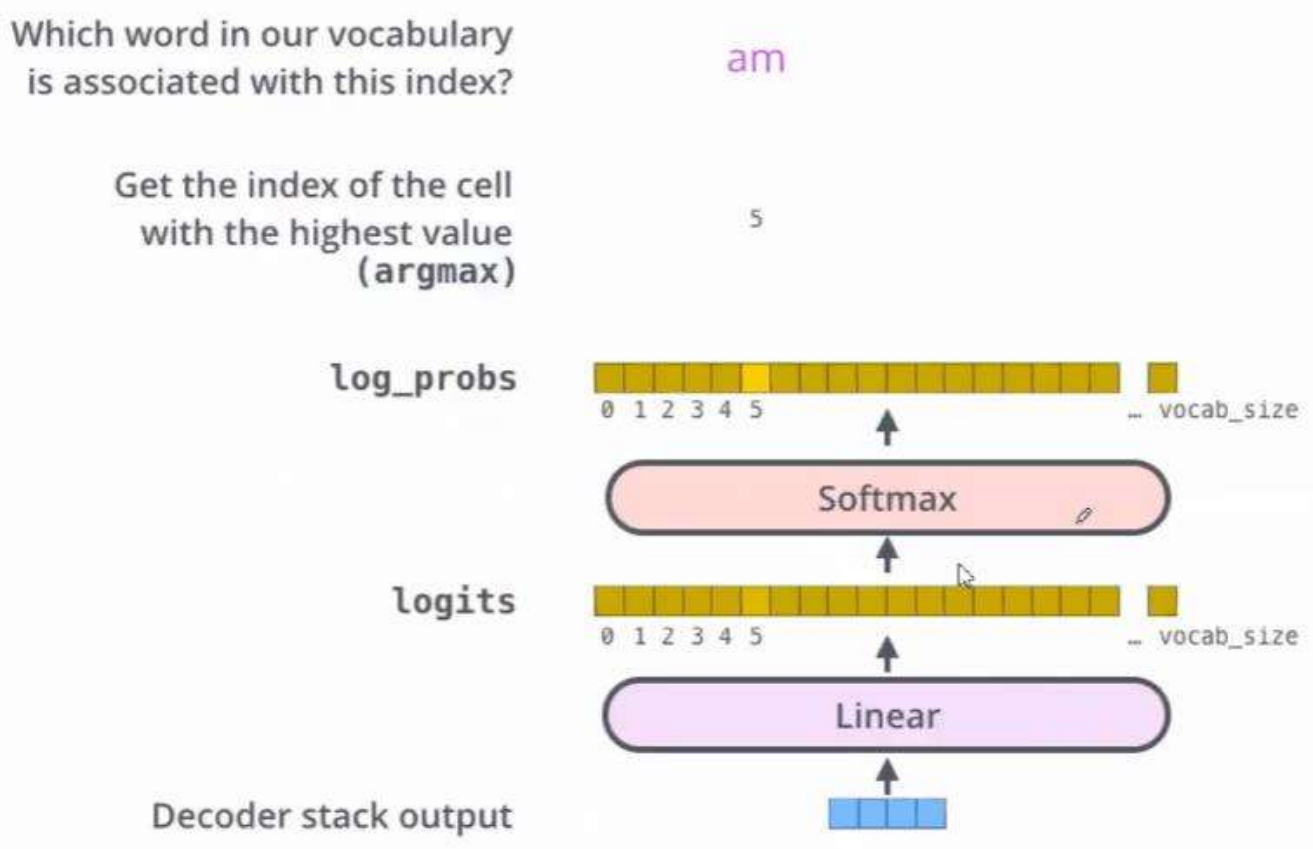
Example Workflow During Inference:

Assuming the model is trained, the decoding process can be summarized in these steps:

1. Start Token: Provide the start token `<s>` along with the Encoder's output to the Decoder. The Decoder predicts the first word. For instance, it outputs "I".
2. Next Word: Now, the Decoder receives `<s> I` as input along with the Encoder's output. It predicts the next word, "am".
3. Continue Decoding: The process continues with `<s> I am` as input to the Decoder. It predicts the next word, "a".
4. Further Prediction: The input `<s> I am a` leads to the prediction of "student".
5. End of Sequence: Finally, `<s> I am a student` is fed to the Decoder, and it predicts the end token `<eos>`, indicating the sentence is complete.
6. Output Sequence: The generated sentence is "I am a student".

Summary:

Press enter or click to view image in full size



- **Decoder Layers**: The Decoder consists of Self-Attention, Encoder-Decoder Attention, and Feed Forward layers, each followed by Add & Normalize.
- **Encoder-Decoder Attention**: Allows the Decoder to focus on the relevant parts of the input sequence.

- **Output Generation**: Uses the softmax layer to predict the next word in the sequence until the end token is produced.

This process enables the Transformer Decoder to generate coherent and contextually appropriate outputs based on the input provided by the Encoder.

Training Tricks: Label Smoothing and Learning Rate Warm-Up

Two training techniques used in the Transformer model to improve performance: **Label Smoothing** and **Learning Rate Warm-Up**

1. Label Smoothing (Regularization)

What is Label Smoothing?

Label smoothing is a regularization technique used to prevent the model from becoming too confident about its predictions. Typically, during training, the model is trained with "hard" labels, where the true label is assigned a probability of 1, and all other classes are assigned a probability of 0. This can lead to overfitting, where the model becomes overly confident in its predictions and fails to generalize well to unseen data.

Traditional One-Hot Encoding:

$$P_i = \begin{cases} 1, & \text{if } i = y \\ 0, & \text{if } i \neq y \end{cases}$$

Modified with Label Smoothing:

Label smoothing modifies this by assigning a slightly lower probability to the true class and distributing some probability mass to the other classes. This is done using a small smoothing parameter ϵ:

Press enter or click to view image in full size

$$P_i = \begin{cases} 1 - \epsilon, & \text{if } i = y \\ \frac{\epsilon}{K-1}, & \text{if } i \neq y \end{cases}$$

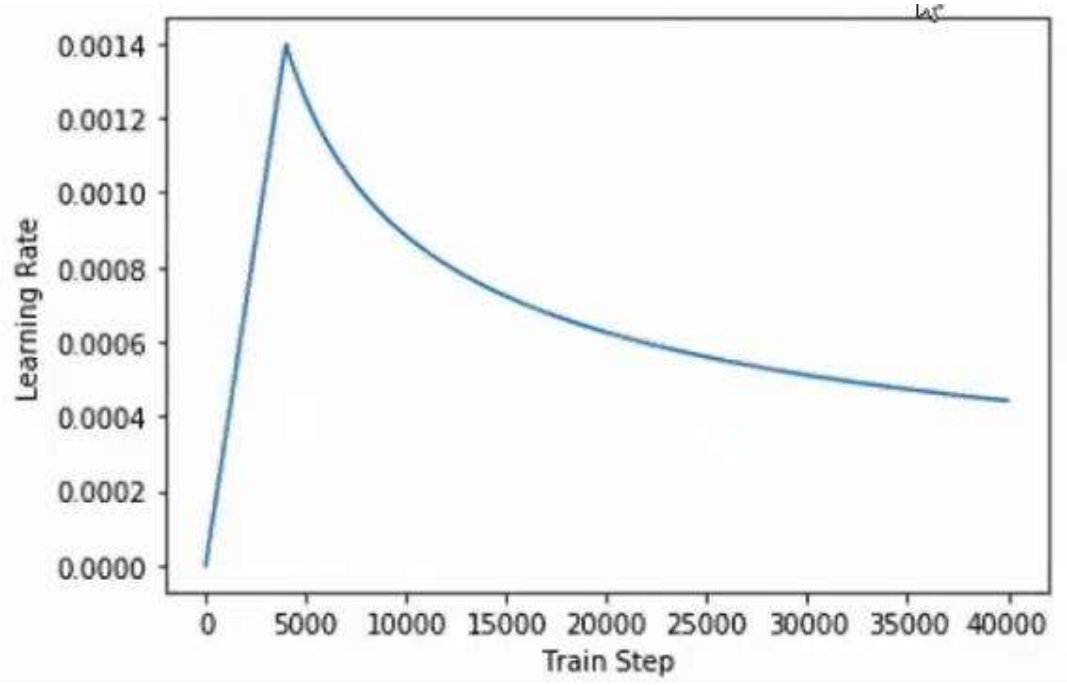
Where:

- K is the total number of classes,
- ϵ is a small hyperparameter (e.g., 0.1) that controls the amount of smoothing.

By using label smoothing, the model is discouraged from becoming too confident in its predictions, improving its ability to generalize to new data.

2. Learning Rate Warm-Up

What is Learning Rate Warm-Up?



The second training trick is the **learning rate warm-up** strategy. When training deep neural networks, it's common to start with a small learning rate to prevent the model from making large updates to the weights initially. This "warm-up" period allows the model to slowly adapt to the learning process.

- **Warm-Up Phase**: The learning rate starts at a small value and gradually increases during the initial stages of training.
- **Decay Phase**: After the warm-up phase, the learning rate is reduced according to a predefined schedule, helping the model converge to a minimum.

Why Use These Techniques?

- **Label Smoothing** helps prevent the model from overfitting by making it less confident in its predictions, which can lead to better generalization on unseen data.
- **Learning Rate Warm-Up** ensures stable training in the initial phase, reducing the chances of large, unstable updates to the model parameters.

Summary:

- **Label Smoothing** modifies the target labels to prevent overconfidence in the model's predictions.
- **Learning Rate Warm-Up** helps in the early stages of training by gradually increasing the learning rate, stabilizing the training process.

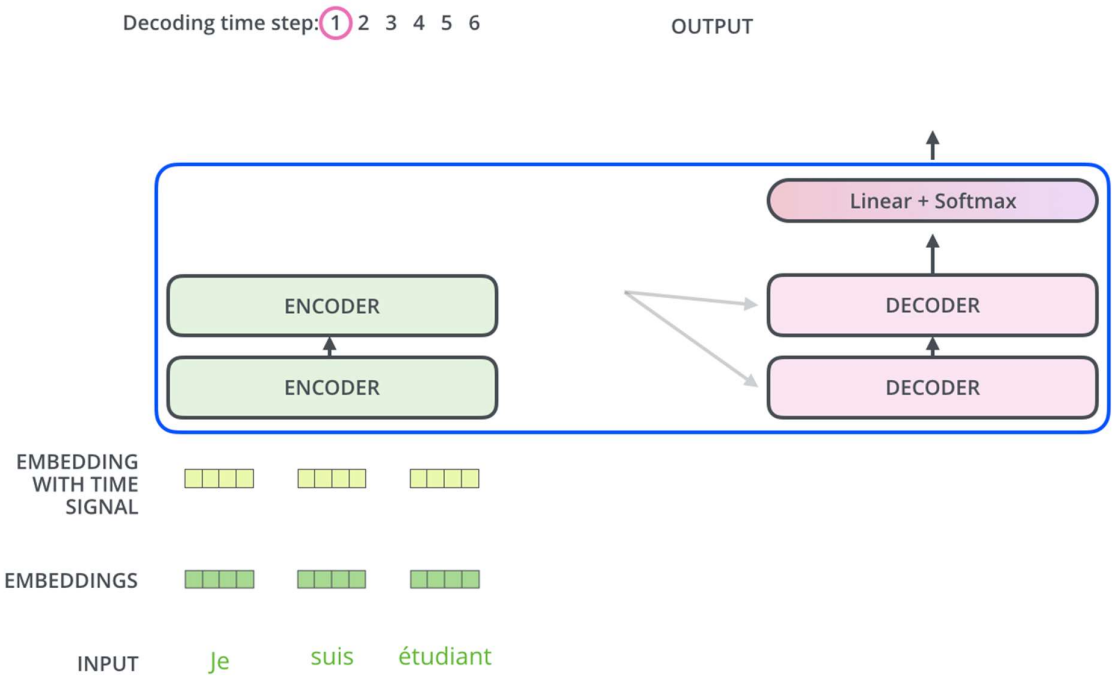
These techniques are used to enhance the Transformer model's training efficiency and robustness, leading to better performance on various natural language processing tasks.

Transformer Dynamic Workflow

The Transformer model operates in two main stages: **Encoding** and **Decoding**. Let's break down each of these stages in detail.

1. Encoding Stage

Press enter or click to view image in full size



Input Sequence Processing: The **Encoder** starts by processing the input sequence. This sequence could be a sentence or any sequence of words or tokens.

Self-Attention Mechanism: Each word in the input sequence is converted into a set of vectors through the self-attention mechanism. These vectors are key components in the Transformer's functioning:

- Key Vectors (K): Encapsulate the information about each word in relation to others, indicating which words are significant for a particular word in the sequence.
- Value Vectors (V): Hold the content-related information of the words, representing the actual content of the input.

Parallelization: Unlike RNNs, this entire process is parallelized. This means that each word is processed in relation to every other word simultaneously, which makes Transformers highly efficient.

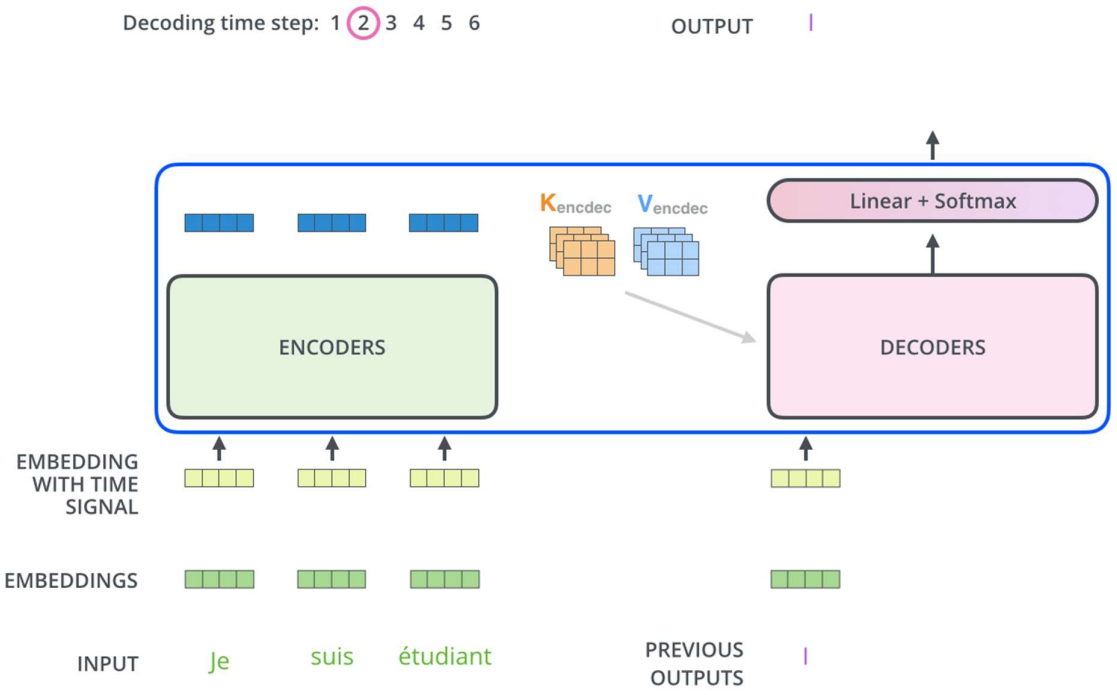
Output of Encoder: The final output of the Encoder is a set of attention vectors containing the key (K) and value (V) vectors for the entire sequence. These vectors are then passed on to each Decoder block.

2. Role of Encoder-Decoder Attention

- **Focus Mechanism**: The attention vectors (K and V) generated by the Encoder are crucial for the Decoder. They are used in the **Encoder-Decoder Attention layer** of the Decoder. This layer allows the Decoder to "focus" on specific parts of the input sequence. It helps the Decoder determine which parts of the input are relevant when generating each word in the output sequence.

3. Decoding Stage

Press enter or click to view image in full size



- Sequential Output Generation: After the Encoder finishes processing, the Decoding phase begins. The Decoder generates the output sequence one element at a time.
- Attention in Decoding:
— Each step in the Decoder uses both self-attention (to consider the sequence generated so far) and Encoder-Decoder attention (to focus on relevant parts of the input).
- Process Repetition: The Decoder continues this process until it reaches a special end-of-sequence token (e.g., `<eos>`), indicating that the output sequence is complete.
- Output Feeding:
— After each word is generated, it is fed into the Decoder's next step as input. For example, if the generated word is "I," it will be used in the next step to help predict the following word, and so on.
- Final Output: The final output is a complete sentence or sequence in the target language (in this example, an English translation).

Example Process Flow:

1. **Encoding**:

The input sequence (e.g., a sentence in a source language) is processed by the Encoder.

The Encoder generates key (K) and value (V) vectors for each word in the input sequence.

2. Decoding:

The Decoder begins generating the output sequence one word at a time.

In each step, the Decoder uses self-attention to consider the words generated so far and Encoder-Decoder attention to focus on relevant parts of the input.

This process repeats until the Decoder outputs the special end-of-sequence token (<eos}>).

Summary:

- **Encoder**: Converts the input sequence into key and value vectors using self-attention, enabling the model to understand the relationships between words in the sequence.

- **Decoder**: Uses these vectors to generate the output sequence one element at a time, utilizing both self-attention and encoder-decoder attention to produce coherent and contextually relevant output.

This dynamic workflow allows the Transformer to efficiently and accurately translate or generate sequences by leveraging parallel processing and sophisticated attention mechanisms.

Characteristics of Transformers

Transformers have significantly advanced the field of natural language processing by introducing a novel architecture that relies on self-attention mechanisms. Below are the key advantages and disadvantages of Transformers.

Advantages

1. **Lower Computational Complexity per Layer Compared to RNNs**

- Transformers have a per-layer computational complexity of $O(n^2 \cdot d)$, where n is the sequence length and d is the dimension.
- Recurrent Neural Networks (RNNs) have a complexity of $O(n \cdot d^2)$ due to their sequential nature.
- This makes Transformers more efficient for processing sequences, especially when the model dimension d is large.

2. Parallel Computation

- Transformers process all positions in the sequence simultaneously, allowing for efficient parallel computation.
- In contrast, RNNs process sequences sequentially, which limits parallelization and increases computation time.

3. Shorter Path Lengths for Long-Distance Dependencies

- **Convolutional Neural Networks (CNNs)** require increasing the number of layers to expand the receptive field and capture long-range dependencies.
- **RNNs** need to process the sequence step by step from position 1 to n, which can lead to long path lengths and gradient vanishing issues.
- **Self-Attention** allows any two positions in the sequence to interact directly in a single computation step.
- The path length for Transformers is 1, enabling more efficient modeling of long-term dependencies compared to RNNs and CNNs.

4. Better Handling of Long-Term Dependencies

- Self-Attention mechanisms can capture relationships between distant positions without degradation.
- This overcomes the limitations of RNNs, which struggle with long sequences due to vanishing or exploding gradients.

5. Interpretability

- The attention weights in Transformers provide insights into how the model makes decisions.
- Visualization of attention distributions shows that Transformers learn syntactic and semantic relationships.
- In the appendix of the original Transformer paper, examples illustrate how the model attends to relevant words in the input when generating translations.

Disadvantages

1. **Difficulty with Certain Tasks Easily Handled by RNNs**

- Transformers may struggle with tasks like string copying or simple algorithmic operations that RNNs handle more naturally.
- When the sequence length during inference is longer than during training, Transformers can face issues because positional embeddings for unseen positions are not learned.
- For example, if a Transformer is trained on sequences up to length 100 and then tested on length 200, it may not perform well due to missing positional embeddings beyond position 100.

2. Not Computationally Universal (Not Turing Complete) in Standard Form

- Theoretical analysis suggests that standard Transformers are not **computationally universal** (Turing complete) without recursion or unbounded resources.
- RNNs, with their recurrent connections and potential for unbounded computation, are Turing complete under certain conditions.
- This limitation means that Transformers may not inherently perform certain types of computations required for complex reasoning or algorithmic tasks without modifications.
- **Note:** Extensions like the Universal Transformer introduce recurrence into the Transformer architecture to address this limitation.

Addressing Computational Constraints

- **Windowed Self-Attention**
- When dealing with very long sequences where the sequence length N exceeds the model dimension D, the computational cost of self-attention becomes high ($O(N^2)$).
- Limiting self-attention to a local window reduces computation to $O(N \cdot w)$, where w is the window size.
- This approach balances the model's ability to capture local dependencies while managing computational resources.
- **Efficient Attention Mechanisms**
- Various methods like **sparse attention**, **low-rank approximations**, and **kernel-based methods** have been proposed to reduce the computational overhead of self-attention.
- These techniques aim to make Transformers more scalable to longer sequences without significantly sacrificing performance.

Conclusion

Transformers have transformed NLP by enabling efficient and effective handling of sequences through self-attention mechanisms. Their ability to process sequences in parallel and model global relationships allows them to capture long-term dependencies more effectively than traditional RNNs or CNNs.

However, Transformers also have limitations:

- They may struggle with tasks that require computational universality or handling sequences longer than those seen during training.
- Modifications and extensions to the standard Transformer architecture, such as adding recurrence or adaptive computation, can help address these limitations.

Understanding both the strengths and weaknesses of Transformers is essential for applying them effectively in various natural language processing tasks.