Transformers

B.Tech. Project Report

by

Afzal Hussain and Ashish kumar

Faculty Adviser

Dr. Saurabh Trivedi

School of Mathematics

Indian Institute of Technology Goa

Abstract

This report investigates the implementation of the Transformer model architecture for Natural Language Processing (NLP) tasks, with a primary focus on its self-attention mechanism and parallel processing capabilities. The objective is to demonstrate the numerical and conceptual advantages of the Transformer over traditional models like RNNs and CNNs. Key components such as multi-head attention, positional encoding, and feed-forward networks are systematically analyzed, implemented, and validated, highlighting their efficiency and accuracy in handling complex NLP problems.

Contents

8 Advancements

Efficient Transformers (BigBird, Longformer), Compact Models (DistilBERT, ALBERT).

9 Conclusion

Transformers revolutionized NLP and continue to expand AI applications.

Transformers: Revolutionizing NLP and Beyond

NLP

Computers communicate using binary code, which is inherently different from human languages. Natural Language Processing (NLP) focuses on teaching computers to understand and interact with human languages.

Definition:
NLP is a field that combines linguistics, computer science, and artificial intelligence to enable machines to process and analyse large-scale natural language data effectively.

Need of NLP

We need NLP to enhance the accessibility of computers and machines. For instance, if someone is unfamiliar with how to use an ATM, an NLP-based application can provide step-by-step guidance, helping the user navigate through the process efficiently.

NLP has many real-life applications like

1.Voiceassistants- voice assistants like Siri, Alexa, Cortana, and google assistant are applications of natural language processing.

2. Spam filtering and smart reply

3 Recommendation systems- while building recommendation systems, we use NLP. Recommendation systems can be of movies, books, songs, etc.

4 Sentiment analysis- Let's say we own an e-commerce website and we want to know what customers of some particular product feel about that product. We can do sentiment analysis on product reviews to see if customers are liking that product or not.

5 Language translators- NLP is used in language translators.

The NLP Pipeline

Developing an NLP application involves several sequential steps collectively called the NLP pipeline:

1. **Data Acquisition:**
   Gathering data from sources such as:
   - **Public Datasets: Kaggle and academic websites.**
   - **Web Scraping: Extracting data from websites using tools like Beautiful Soup.**
   - **APIs: Many websites offer APIs for structured data retrieval (e.g., RapidAPI).**
2. **Text Preprocessing:**
   Raw data is cleaned and formatted for analysis through the following processes:
   - **Converting text to lowercase.**
   - **Removing punctuations and irrelevant symbols like emojis.**
   - **Eliminating stop words (e.g., "the," "is").**
   - **Tokenization: Breaking text into smaller units, such as words or sentences.**
   - **Stemming: Reducing words to their root forms (e.g., "changing" to "change").**
   - **Part-of-Speech (POS) Tagging: Assigning grammatical roles to words.**
   - **Spell correction.**
3. **Feature Engineering:**
   Converting text into numerical representations is critical for applying machine learning models. Common techniques include:
   - **Bag of Words: Represents text as a matrix of word frequencies in sentences but ignores word order.**
   - **TF-IDF (Term Frequency-Inverse Document Frequency): Weighs words based on their frequency across documents to highlight important terms.**

- **Word2Vec: Maps words into dense vectors that capture their semantic relationships.**

Word2Vec

Google engineers are responsible for creating Word2Vec. The model has already been trained. The Google News dataset is used to train it. Word2Vec creates vectors from words. It conveys the words' semantic meaning. In contrast to a bag of words and TF-IDF, it transforms the words into low-dimensional vectors. It provides us with dense vectors.

Imagine we have the following small vocabulary of five words:
Vocabulary: ["king", "queen", "man", "woman", "monkey"]

After training Word2Vec, each word is represented as a dense vector in a low-dimensional space. Let's assume the embeddings for these words are as follows (in a 5-dimensional space):

Word	Embedding (Vector Representation)
King	[0.8, 0.9, 0.7, 0.6, 0.5]
Queen	[0.7, 0.8, 0.6, 0.5, 0.4]
Man	[0.6, 0.4, 0.9, 0.3, 0.2]
Woman	[0.5, 0.3, 0.8, 0.2, 0.1]
Monkey	[0.2, 0.1, 0.4, 0.1, 0.05]

How Word2Vec Captures Relationships

Example 1: Gender Relationship

One of the core features of Word2Vec is its ability to capture analogical relationships, such as:

King - Man + Woman ≈ Queen

Calculation:

1. Start with the vector for King: [0.8, 0.9, 0.7, 0.6, 0.5]
2. Subtract the vector for Man:

[0.8, 0.9, 0.7, 0.6, 0.5] - [0.6, 0.4, 0.9, 0.3, 0.2] = [0.2, 0.5, -0.2, 0.3, 0.3]

3.Add the vector for Woman:

[0.2, 0.5, -0.2, 0.3, 0.3] + [0.5, 0.3, 0.8, 0.2, 0.1] = [0.7, 0.8, 0.6, 0.5, 0.4]

The resulting vector is [0.7, 0.8, 0.6, 0.5, 0.4], which is the vector for Queen.

Doc2vec

Doc2Vec is an extension of Word2Vec that represents entire documents, paragraphs, or sentences as numerical vectors. While Word2Vec generates embeddings for individual words, Doc2Vec creates a fixed-length vector representation for a piece of text, preserving semantic meaning across varying text lengths.

How it works

Imagine you have the following two documents:

Document 1: "I love reading books."
Document 2: "Books are amazing to read."

Our vocabulary consists of unique words:
["I", "love", "reading", "books", "are", "amazing", "to", "read"]

We also have two document IDs:

- **Document 1: D1**
- **Document 2: D2**

Vocabulary and Document Embedding Sizes:

- **Each word is represented as a 5-dimensional vector.**
- **Each document is represented as a 3-dimensional vector.**

Initial Embeddings:

Word Vectors (randomly initialized):

Word	Embedding (Vector Representation)
I	[0.1, 0.3, 0.5, 0.2, 0.4]
Love	[0.2, 0.1, 0.4, 0.3, 0.6]
Reading	[0.3, 0.2, 0.1, 0.4, 0.5]
Books	[0.5, 0.4, 0.3, 0.6, 0.2]
Are	[0.4, 0.6, 0.5, 0.2, 0.1]
Amazing	[0.3, 0.5, 0.6, 0.1, 0.2]
To	[0.2, 0.4, 0.3, 0.5, 0.1]
Read	[0.5, 0.3, 0.4, 0.2, 0.6]

Document Vectors (randomly initialized):

Document	Embedding (Vector Representation)
D1	[0.7, 0.6, 0.8]
D2	[0.5, 0.9, 0.4]

---

1. PV-DM (Distributed Memory) in Action

Context Window:

Let's say the context window size is 2. For the target word "reading" in Document 1, the context is:
["I", "love"].

Step 1: Combine Context Words and Document Vector

We concatenate the context word vectors and the document vector:

Context Word Vectors:

- **"I" → [0.1, 0.3, 0.5, 0.2, 0.4]**
- **"Love" → [0.2, 0.1, 0.4, 0.3, 0.6]**

Document Vector for D1:

- **[0.7, 0.6, 0.8]**

Concatenated Input:

- **[0.1, 0.3, 0.5, 0.2, 0.4] + [0.2, 0.1, 0.4, 0.3, 0.6] + [0.7, 0.6, 0.8]**
- **Result: [0.1, 0.3, 0.5, 0.2, 0.4, 0.2, 0.1, 0.4, 0.3, 0.6, 0.7, 0.6, 0.8]**

Step 2: Predict the Target Word

This concatenated input is passed through a simple neural network. The output is a probability distribution over the vocabulary, aiming to predict the target word ("reading").

Predicted Target Word Vector: [0.3, 0.2, 0.1, 0.4, 0.5]

Step 3: Update Vectors

Using backpropagation, the model adjusts:

1. **The vectors of context words ("I" and "love")**
2. **The document vector (D1)**
3. **The weights of the neural network**

2. PV-DBOW (Distributed Bag of Words) in Action

Step 1: Randomly Select a Word from the Document

For Document 1, let's randomly select the word "books".

Step 2: Use the Document Vector to Predict the Word

Instead of using context words, PV-DBOW directly uses the document vector (D1) to predict the word "books".

Input: [0.7, 0.6, 0.8]

The neural network processes this input and outputs a probability distribution over the vocabulary.

Predicted Target Word Vector: [0.5, 0.4, 0.3, 0.6, 0.2]

Step 3: Update the Document Vector

Using backpropagation, only the document vector (D1) is updated in this model.

Final Embeddings:

After several iterations of training, the word and document embeddings are adjusted to reflect semantic relationships. For example:

- **Document 1 (D1) and Document 2 (D2) will have similar vectors if their content is semantically related.**
- **Words like "reading" and "books" will have closer vectors since they often appear together.**

Doc2Vec is an essential tool for tasks where understanding the overall meaning of a document is critical, making it a natural extension of Word2Vec.

Models before Transfromers:

1. **RNNs/LSTMs/GRUs**: Sequence modeling, NLP tasks (machine translation, text generation).
2. **CNNs**: Image processing, object recognition, and even sequence tasks like text classification.
3. **FNNs/MLPs**: Early neural network architectures used for classification and regression.
4. **Word2Vec/GloVe**: Word embedding models that learned distributed word representations.
5. **Seq2Seq with Attention**: Encoder-decoder models for sequence-to-sequence tasks (e.g., translation).
6. **CRFs**: Used for sequence labeling tasks, often in NLP.
7. **DBNs and Autoencoders**: Used in unsupervised learning, dimensionality reduction, and generative models.

What is RNNs (Recurrent Neural Networks)

RNNs were one of the first models designed to handle sequence data, like sentences or time-series data, where the order of elements matters. RNNs process data step-by-step, handling one word or item at a time, passing information from one step to the next. This process is useful for understanding sequences, but it comes with significant limitations:

How Do RNNs Work?

1. **Recurrent Structure: RNNs process input sequentially, maintaining a hidden state that captures information about previous elements in the sequence.**

2. **Shared Weights: Each step of the sequence uses the same parameters, making RNNs computationally efficient for sequential tasks.**

At each time step t:

- **Input x_t is combined with the hidden state h_{t-1} (from the previous time step).**
- **The current hidden state h_t is computed using:**

$$H_t=\sigma(W_h h_{t-1}+W_x x_t+b)$$

where W_h, W_x, and b are trainable parameters, and σ is an activation function (e.g., tanh or ReLU).

- **The output y_t is derived as:**

$$y_t=\text{softmax}(W_y h_t+b_y)$$

Example: Sentiment Analysis

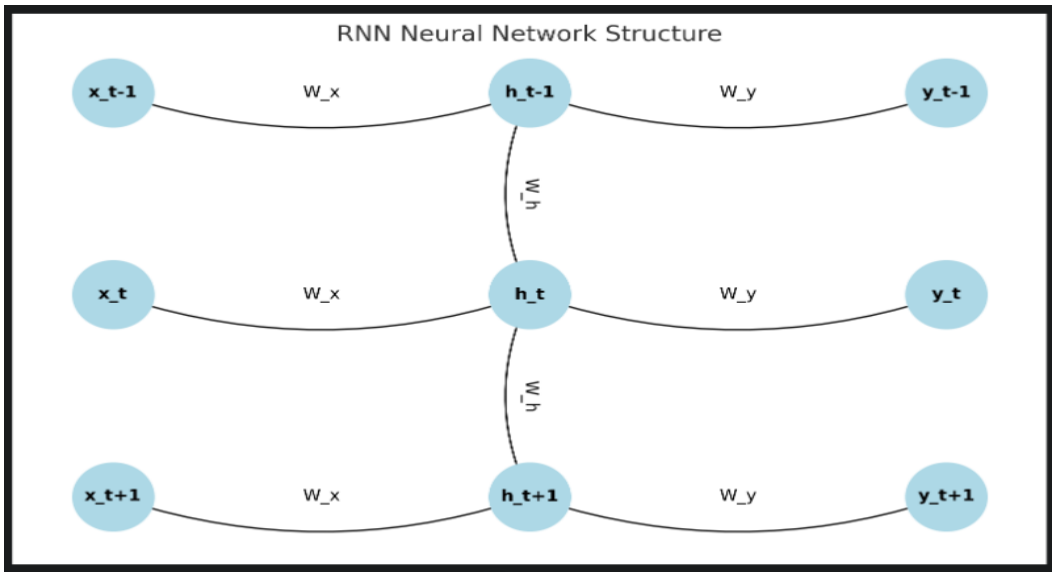
Suppose we want to analyze the sentiment of the sentence: *"The movie was great!"*

1. **Input Sequence: Tokenize the sentence into words: ["The", "movie", "was", "great"].**
2. **Embedding: Map each word into a numerical vector using word embeddings.**
3. **RNN Processing: Process each word one at a time, updating the hidden state.**
    - **Step 1: Input "The" → Hidden State 1**
    - **Step 2: Input "movie" + Hidden State 1 → Hidden State 2**
    - **Step 3: Input "was" + Hidden State 2 → Hidden State 3**
    - **Step 4: Input "great" + Hidden State 3 → Hidden State 4**
4. **Output: Based on the final hidden state, classify the sentiment as "Positive."**

Input Nodes (x_t) represent tokens or features from the input sequence.

Hidden States (h_t) maintain memory and context through recurrent connections.

Output Nodes (y_t) provide predictions for each time step



What is CNN (Convolutional Neural Networks)

CNNs, commonly used for image data, were adapted for sequence tasks to improve upon RNNs. Unlike RNNs, CNNs can process multiple parts of a sequence at once by working on "windows" or "blocks" of data in parallel. This allows them to extract local patterns faster.

How Do CNNs Work?

1. **Convolutional Layer: Applies filters (kernels) to input data to extract features like edges, textures, or patterns.**

$$\text{Output}(i,j)=\sum_{k,l}\text{Input}(i+k,j+l).\text{Filter}(k,l)$$

2. **Pooling Layer: Reduces the spatial dimensions (e.g., max pooling or average pooling), making the model computationally efficient and robust to small changes in input.**
3. **Fully Connected Layer: Flattens the features and connects them to a classification or regression output.**

Example: Text Classification

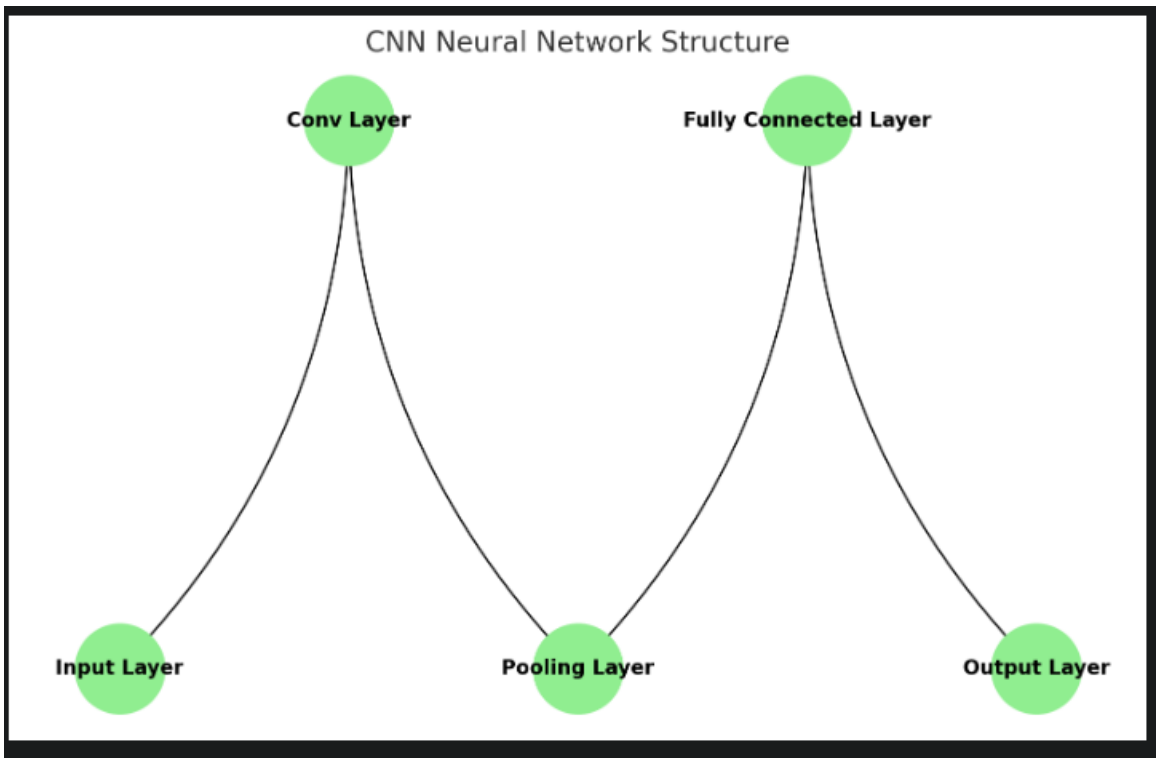
Suppose we want to classify the sentence: *"This is a fantastic product!"*

1. **Input: Convert the sentence into a matrix using word embeddings. Each row corresponds to a word vector.**

Convolution: Apply filters over n-grams (e.g., trigrams) to extract meaningful features.

- **Example: A filter might capture the sentiment in phrases like "fantastic product."**

2. **Pooling: Reduce dimensions by taking the maximum value from each filter output (max pooling).**
3. **Output: Flatten and connect to a softmax classifier for the final label (e.g., "Positive").**



1. **Input Layer: Takes in raw data, such as an image or text embeddings.**
2. **Convolutional Layer: Extracts local patterns using filters.**
3. **Pooling Layer: Reduces dimensionality and retains essential features.**
4. **Fully Connected Layer: Connects learned features to the output.**
5. **Output Layer: Provides the final prediction (e.g., classification).**

What is Transformer

A Transformer is a deep learning model that uses a neural network architecture to convert an input sequence into an output sequence. It plays a key role in natural language processing (NLP) and is widely used in machine learning and artificial intelligence for tasks like translation, text generation, and more.

Comparison RNNs vs CNN vs Transformers

1. Overview and Intuition

Feature	Transformers	RNNs	CNNs
Architecture	Based on self-attention and parallelism.	Sequential, processes input step-by-step.	Processes input with convolutional filters.
Core Idea	Understand relationships between all input tokens using attention.	Use previous context to process the current token.	Capture spatial/local features in input.
Primary Use Case	NLP (translation, summarization, etc.), Vision.	Sequential data like time-series, text.	Images, videos, and spatial data.

2.Input Sentence:

"The movie was incredible, and the visuals were stunning."

RNN (e.g., LSTM or GRU)

Approach:

1. **Process the input sequentially:**
    - **"The" → "movie" → "was" → "incredible" → ...**
2. **Maintains a "hidden state" to remember context as it reads through tokens:**
    - **After processing "movie", it remembers it when reaching "incredible".**

3. **Outputs the final sentiment based on the accumulated context in the hidden states.**

Weaknesses:

- **Long-range dependencies are harder to capture effectively (e.g., "stunning" might lose context with "visuals" due to distance).**
- **Sequential nature makes it slower compared to Transformers.**

CNN (e.g., TextCNN)

Approach:

1. **Apply convolutional filters over n-grams (e.g., "The movie", "movie was", "was incredible", etc.).**
2. **Each filter captures patterns, such as sentiment-related phrases ("was incredible", "were stunning").**
3. **Outputs a fixed-size representation for classification.**

Strengths:

- **Efficient for local feature extraction (e.g., "was incredible").**
- **Cannot capture long-range dependencies (e.g., connection between "stunning" and "visuals" is missed).**

Transformers (e.g., BERT)

Approach:

1. **Break the input into tokens ("The", "movie", "was", etc.).**
2. **Each token is embedded into a vector.**
3. **Self-attention assigns importance to each token by relating it to all others. For example:**
    - **"incredible" relates strongly to "movie".**
    - **"stunning" relates strongly to "visuals".**
4. **Outputs are contextualized embeddings representing relationships across the sentence.**

5. **A classification layer predicts the sentiment (e.g., Positive).**

Strengths:

- **Captures long-range dependencies efficiently (e.g., "stunning" relates to "visuals" far apart in text).**
- **Processes all tokens in parallel, speeding up computation.**

3. Example Task: Image Classification

Input Image:

A photo of a cat sitting on a sofa.

Transformers (e.g., Vision Transformer - ViT)

Approach:

1. **Divide the image into patches (e.g., 16x16 pixels).**
2. **Flatten and embed each patch into a vector.**
3. **Apply self-attention to relate all patches:**
    - **Identify the "cat" region and distinguish it from the "sofa."**
4. **Classify the image based on the relationships between patches.**

Strengths:

- **Captures global patterns in the image (e.g., "cat on sofa").**
- **Outperforms CNNs for large datasets due to its attention mechanism.**

CNN

Approach:

1. **Apply convolutional filters to capture local patterns (e.g., fur texture, cat ears).**
2. **Gradually combine features through pooling layers, identifying larger patterns (e.g., "cat face").**
3. **Classify based on extracted features.**

Strengths:

- **Highly efficient for spatial data.**
- **Struggles with global patterns compared to Transformers.**

RNN (Not Typical for Images)

RNNs are not commonly used for image classification because they process sequential data rather than spatial data. However, they could process an image row-by-row (inefficient and not effective).

4. Comparison Table

Feature	Transformers	RNNs	CNNs
Best For	**Long-range dependencies, large datasets.**	**Sequential data (text, time-series).**	**Spatial data (images, videos).**
Training Speed	**Fast (parallel processing).**	**Slow (sequential processing).**	**Fast (highly parallelizable).**
Handling Long Dependencies	**Excellent.**	**Struggles with very long sequences.**	**Poor (local features only).**
Data Type	**Sequential or spatial (after patching).**	**Sequential.**	**Spatial.**
Computational Cost	**High (requires more memory).**	**Moderate.**	**Moderate to low.**

5. Conclusion

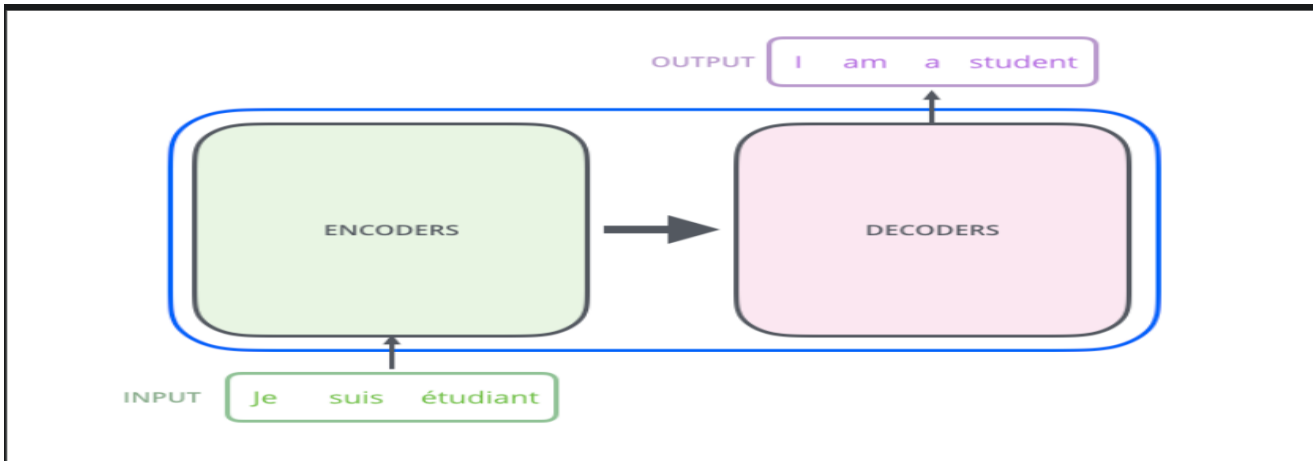
Model	Strengths	Limitations
Transformers	Best for global relationships and long sequences (NLP, vision, large datasets).	High computational cost, requires a lot of data.
RNNs	Best for time-series and sequential tasks where order matters.	Struggles with long dependencies, slower.
CNNs	Best for images and local patterns.	Struggles with long-range relationships.

Result: Transformer outperforms previous Models.

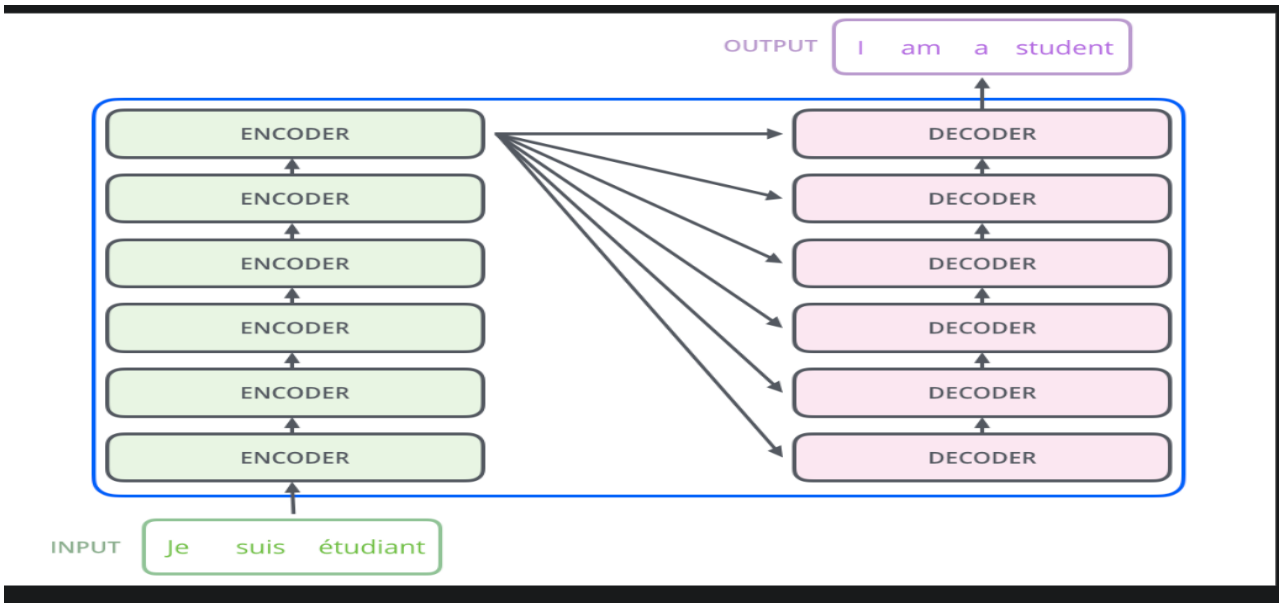
Transformer's Architecture

Encoder: Converts the input sequence into continuous representations using self-attention and point-wise feed-forward layers.

Decoder: Generates the output sequence, utilizing the encoder's output and its own self-attention mechanism



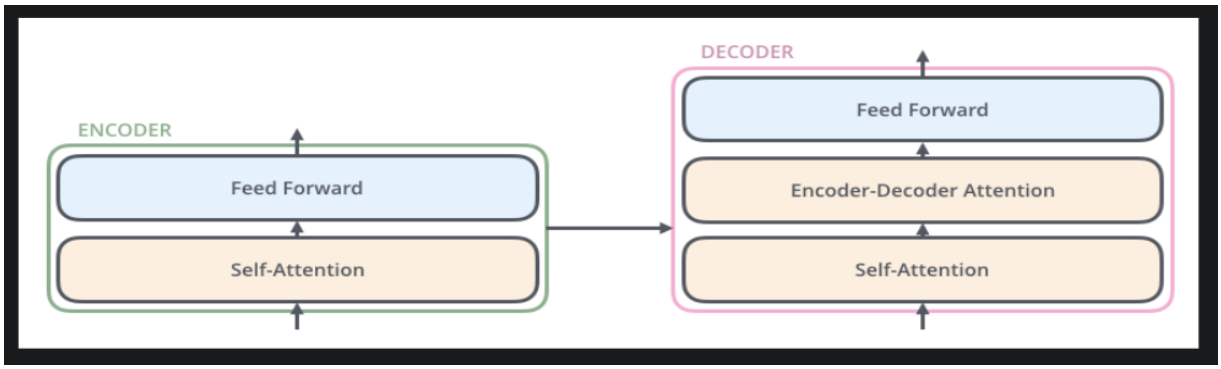
The encoding component is a stack of encoders. The decoding component is a stack of decoders of the same number.



The encoder's inputs first flow through a self-attention layer – a layer that helps the encoder look at other words in the input sentence as it encodes a specific word.

The outputs of the self-attention layer are fed to a feed-forward neural network. The exact same feed-forward network is independently applied to each position.

The decoder has both those layers, but between them is an attention layer that helps the decoder focus on relevant parts of the input sentence



Self-Attention in the Encoder and Decoder

- **Encoder:**
  - **Uses self-attention to consider all positions in the input sequence simultaneously, weighing the importance of each word based on the others.**
  - **Fully connected layers process this representation further.**
- **Decoder:**
  - **Also uses self-attention but with two types:**
    - **Self-attention over the generated output so far.**
    - **Attention over the encoder's output to integrate context from the input sequence.**
  - **This auto-regressive nature ensures that the decoder generates one symbol at a time.**

The Encoder Stack

- **Structure: The Transformer's encoder consists of 6 identical layers.**
  - **Sub-layers:**
    - **Multi-Head Self-Attention: Allows the model to attend to different parts of the input sequence simultaneously, capturing relationships between words in the input.**
    - **Feed-Forward Network: A fully connected, position-wise feed-forward network that processes each position independently after the attention mechanism.**
- **Residual Connections:**
- **Residual connections are added around each sub-layer to help with the flow of gradients and prevent vanishing gradient problems.**
- **The output of each sub-layer is processed as: LayerNorm(x + Sublayer(x)), where Sublayer(x) is either the self-attention or the feed-forward network.**
- **Layer Normalization: Applied after each residual connection for stabilization.**
- **Dimensionality: Each layer (including embeddings) has an output dimension of 512.**

The Decoder Stack

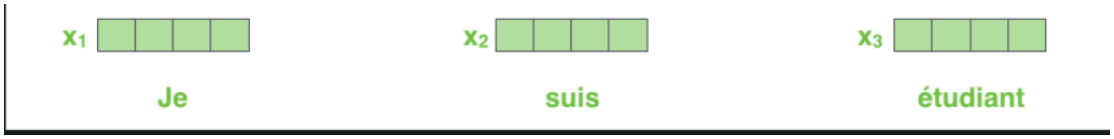
- **Structure: Similar to the encoder, the decoder also consists of 6 identical layers, but with an additional sub-layer.**
    - **Sub-layers:**
        - **Masked Multi-Head Self-Attention: Prevents positions from attending to future positions, ensuring the model only uses past (or known) outputs to make predictions at the current position.**
        - **Multi-Head Attention over the Encoder's Output: This sub-layer allows the decoder to attend to the encoder's output, integrating the context from the input sequence.**
        - **Feed-Forward Network: A fully connected layer similar to the encoder's feed-forward sub-layer.**
- **Residual Connections and Layer Normalization: As with the encoder, each sub-layer in the decoder is wrapped with residual connections and layer normalization.**
- **Positional Embedding: The decoder also uses positional encoding to retain the information about the order of the words in the output sequence.**

Masking in the Decoder

- **Purpose of Masking: Masking ensures that during training, each output prediction is based only on previous outputs (auto-regressive property).**
- **The mask blocks attention to future positions, which prevents the model from "cheating" by looking ahead at the next word.**
- **The output embeddings are also shifted by one position to align predictions with the correct output.**

Embedding

Turn each input words into a vector using an embedding algorithm.

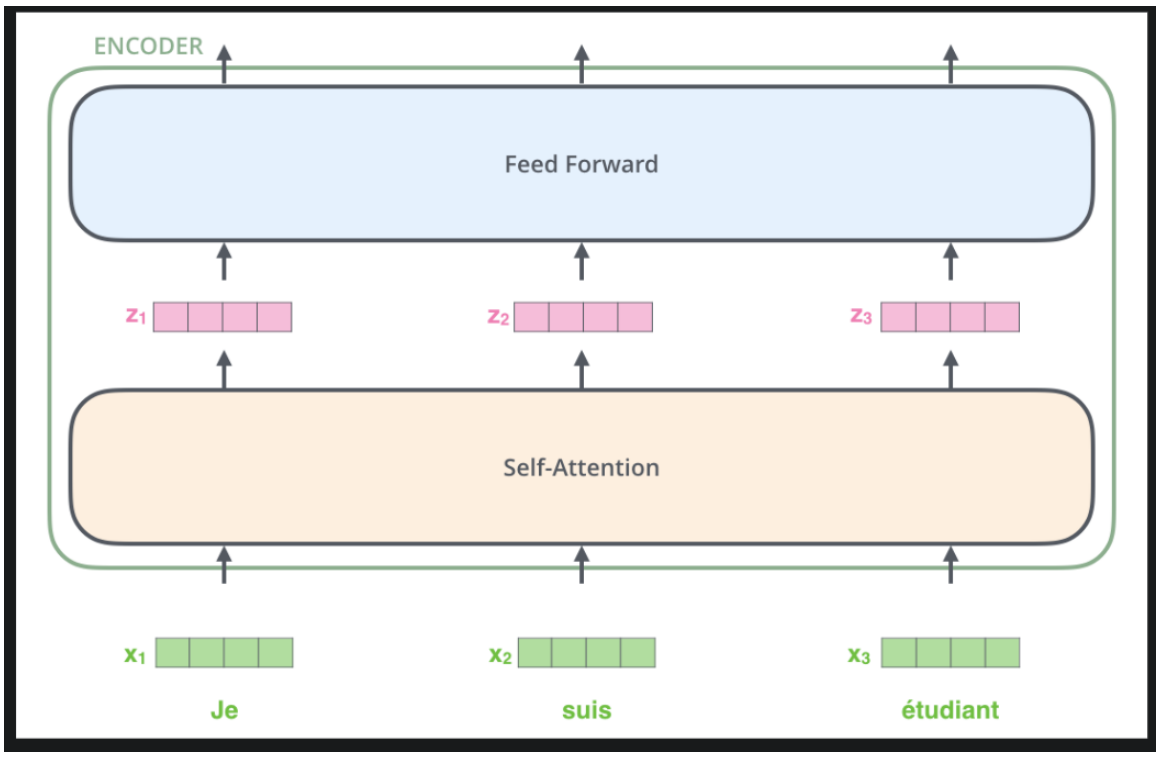


Embedding happens only in the first encoder. This step converts words into numerical vectors (word embeddings).

All other encoders work the same way, taking a list of vectors as input. Each vector is of size 512.

In the first encoder, the input vectors are word embeddings. In subsequent encoders, the input is the output from the encoder below.

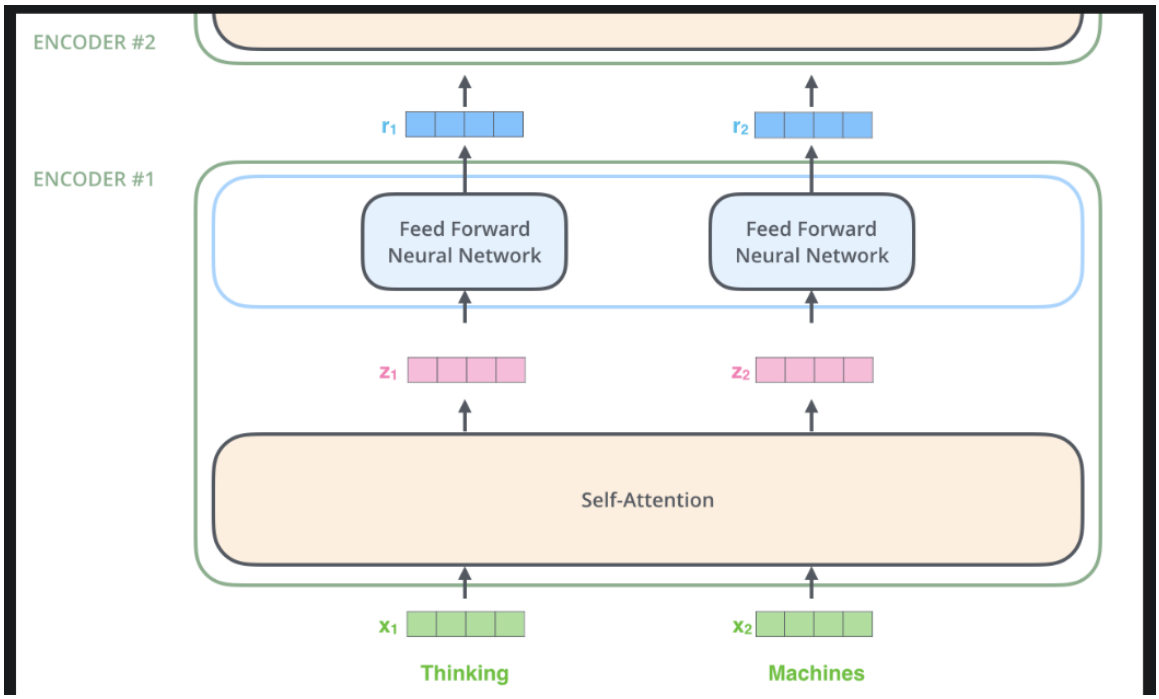
The length of the input list is adjustable and usually corresponds to the length of the longest sentence in the training data. Once the words in the input sequence are embedded, each word passes through the two layers of every encoder.



In the Transformer, each word follows its own path through the encoder, with dependencies between paths in the self-attention layer. However, in the feed-forward layer, there are no dependencies, so the paths can be processed in parallel.

Encoding

An encoder takes a list of vectors as input, processes them through a self-attention layer, then a feed-forward neural network, and finally sends the output to the next encoder.

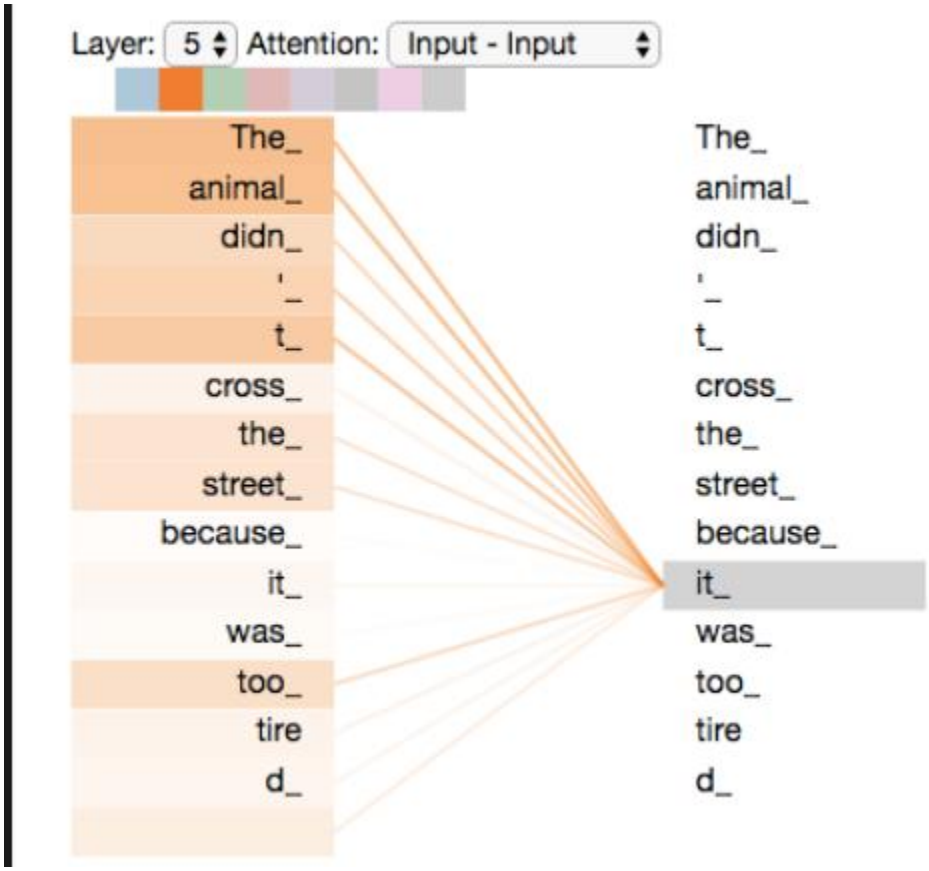


Self-Attention

Say the following sentence is an input sentence we want to translate:

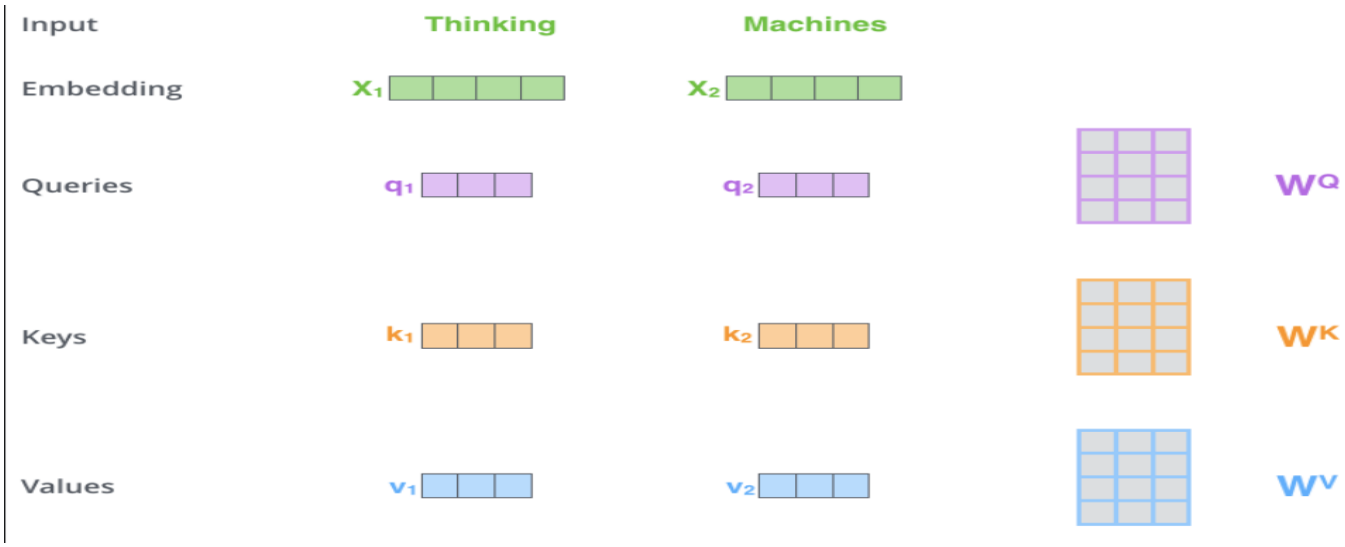
" The animal didn't cross the street because it was too tired"

When the model is processing the word "it", self-attention allows it to associate "it" with "animal".



To compute self-attention:

1. **For each input word embedding, we generate three vectors: Query, Key, and Value.**
2. **These vectors are created by multiplying the embedding with three separate trained matrices.**
3. **Each of these vectors has a smaller dimension of 64 (compared to the embedding and encoder vector size of 512).**
4. **The smaller size is a deliberate design choice to keep the computational cost of multi-headed attention manageable.**



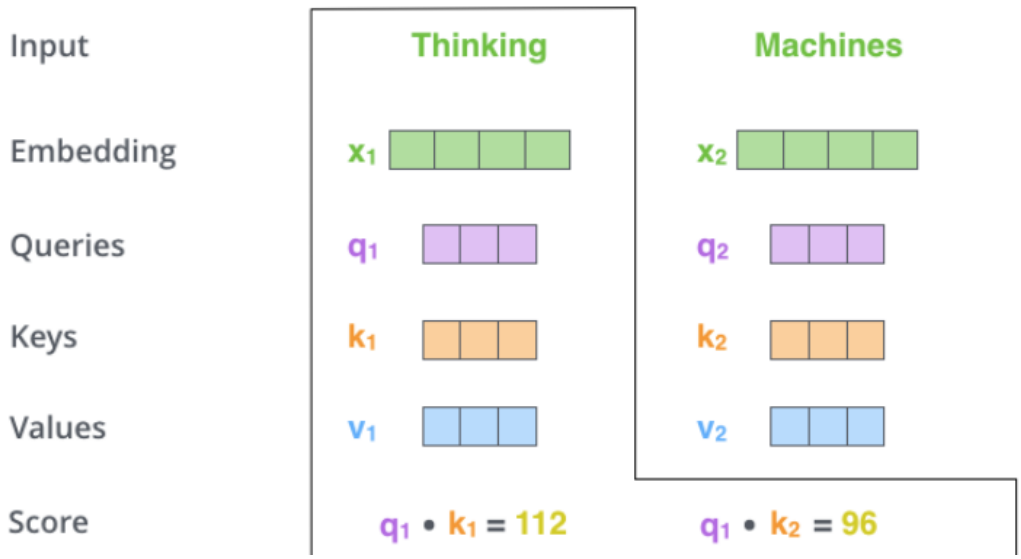
- **The model takes queries (Q), keys (K), and values (V) as inputs.**
- **Dot Product: The query is compared to all keys by taking their dot product (measures similarity).**
- **Scaling: Each dot product is divided by $\sqrt{d_k}$, where d_k is the dimensionality of the key vectors.**
- **Softmax: A softmax function is applied to these scaled values, converting them into probabilities (attention weights).**
- **Weighted Sum: The attention weights are used to compute a weighted sum of the values (V), which is the output.**

The Query, Key, and Value vectors are essential for calculating and understanding attention. Each vector plays a specific role in determining how much importance one word should give to another in a sentence.

Calculating Attention Scores

- **To compute attention for a word (e.g., *"Thinking"*), we compare it with every other word in the sentence.**
- **The attention score indicates how much focus the word *"Thinking"* should place on each other word.**

- **This score is calculated by taking the dot product of the query vector for the word (*q1*) and the key vector for another word (*k1*, *k2*, etc.).**



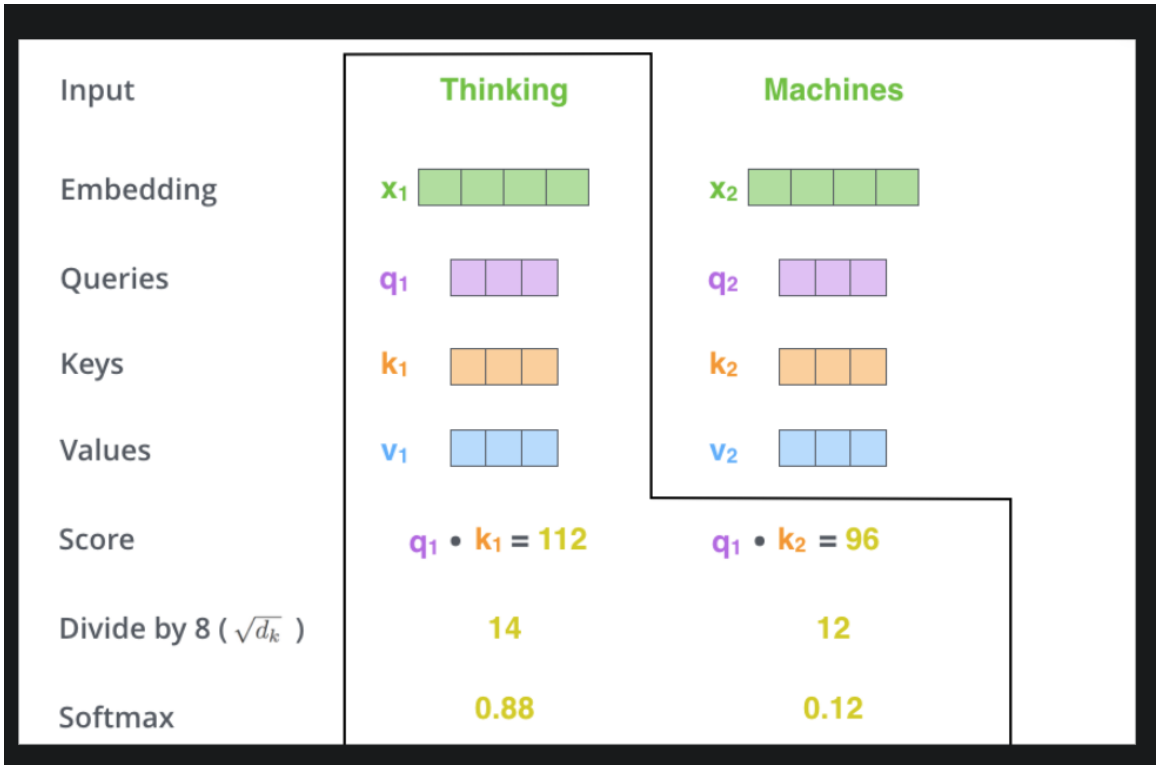
The next steps are:

Scale the Scores:

- **Divide the attention scores by 8 (the square root of the dimension of the key vector, which is 64).**
- **This scaling helps to ensure more stable gradients during training.**

Apply Softmax:

- **Pass the scaled scores through a softmax function.**
- **This normalizes the scores so that all values are positive and their sum equals 1.**
- **This step ensures that the model assigns meaningful attention weights to different words in the sentence.**



The next steps are:

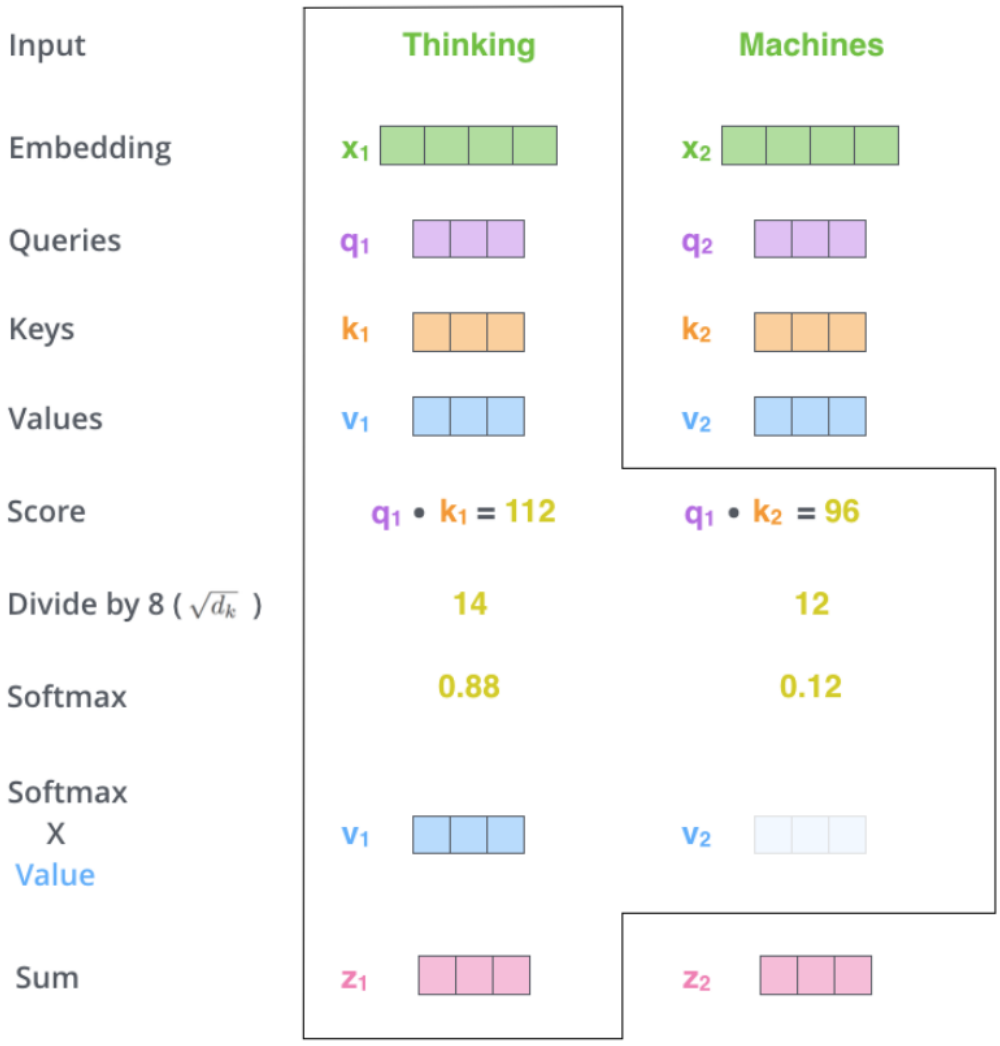
Weight the Value Vectors:

- **Multiply each value vector by its corresponding softmax score.**
- **This step emphasizes relevant words (with higher scores) and reduces the influence of less relevant ones (with smaller scores like 0.001).**

Sum the Weighted Value Vectors:

- **Add the weighted value vectors together.**
- **This produces the output of the self-attention layer for the current word (e.g., the first word in the sequence).**

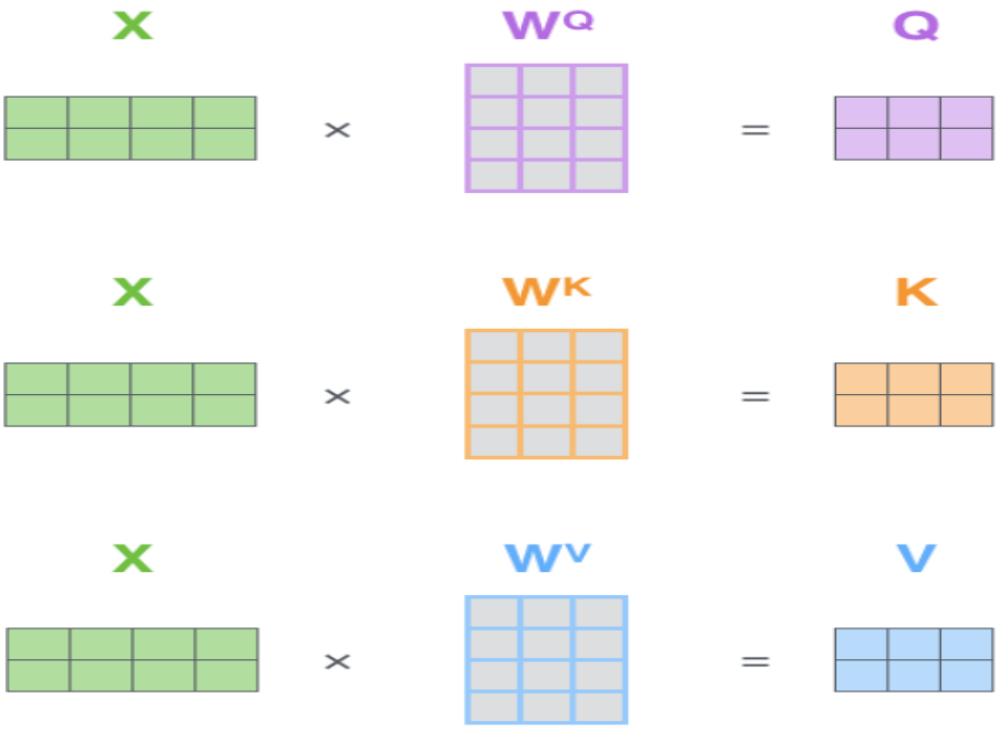
This process ensures that the self-attention output captures the most important context for each word.

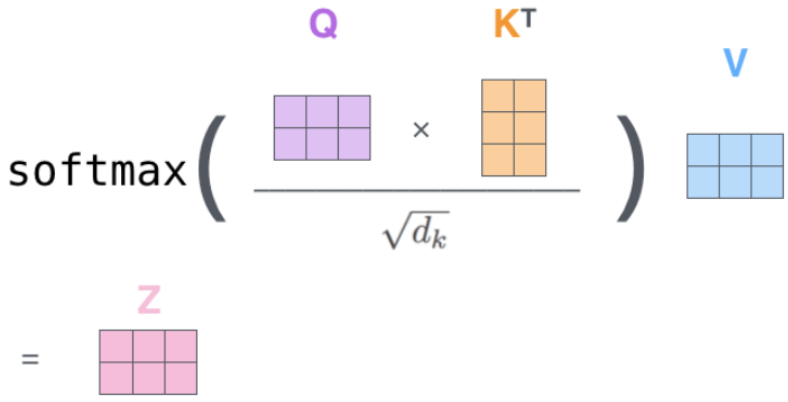


Now the resulting vector is ready to be sent to the feed-forward neural network.

Matrix Calculation of Self-Attention

The first step is to calculate the Query, Key, and Value matrices. We do this by packing the embeddings into a matrix X and multiplying it by the trained weight matrices W^Q, W^k, and W^v.





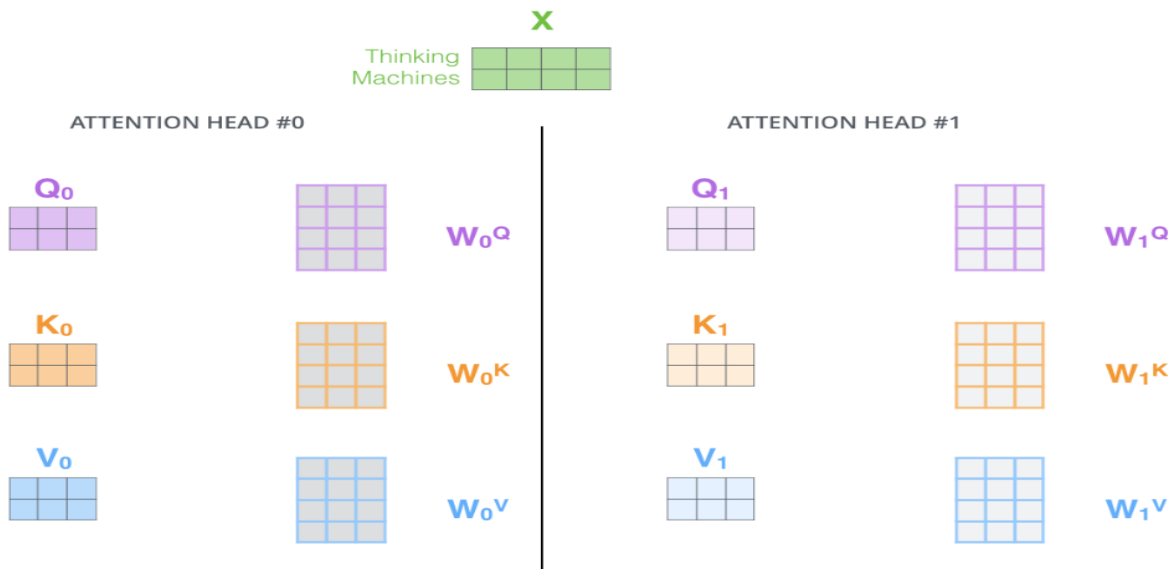
Multi heads

Multi-headed attention allows the model to focus on different parts of a sentence simultaneously, helping it understand relationships from multiple perspectives.

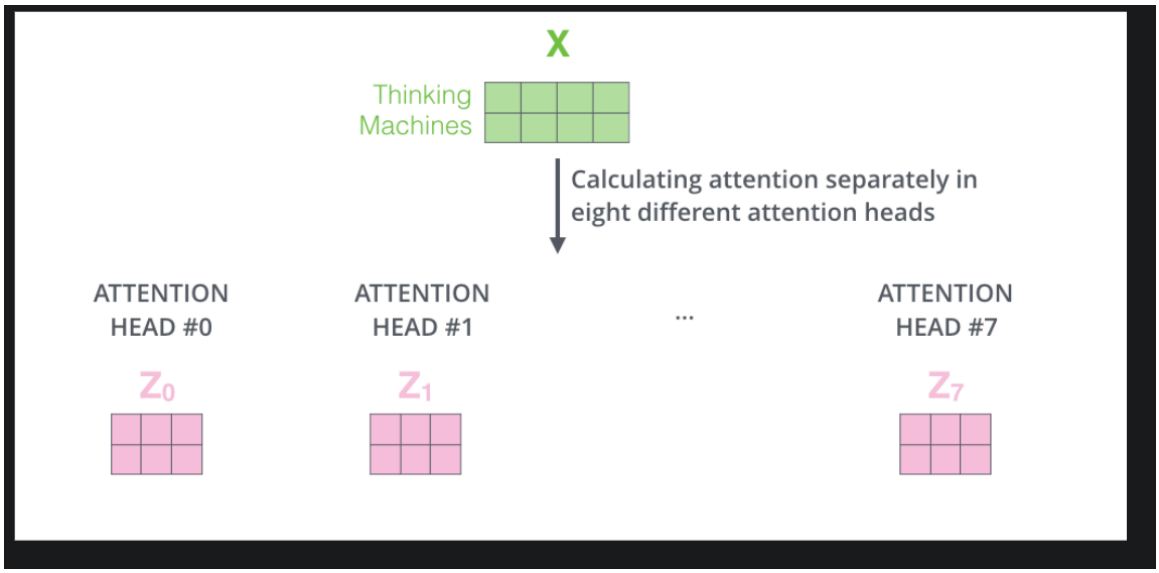
For example, in the sentence "The animal didn't cross the street because it was too tired," multi-headed attention helps the model determine that "it" refers to "the animal" instead of "the street".

- **Multiple Attention Heads:**
  Each attention head has its own set of Query, Key, and Value matrices.
  In the Transformer, there are eight sets of these matrices for each encoder and decoder.
    - **This means that the model can learn multiple types of relationships and focus on different aspects of the input data.**
- **Unique Representations:**
  After training, each attention head projects the input embeddings into a unique representation, enabling the model to capture various features and interactions in the data.
    - **This gives the model a more comprehensive understanding of the input sentence.**

In summary, multi-headed attention enriches the model's ability to focus on different parts of the input simultaneously, improving the context-awareness of the output.

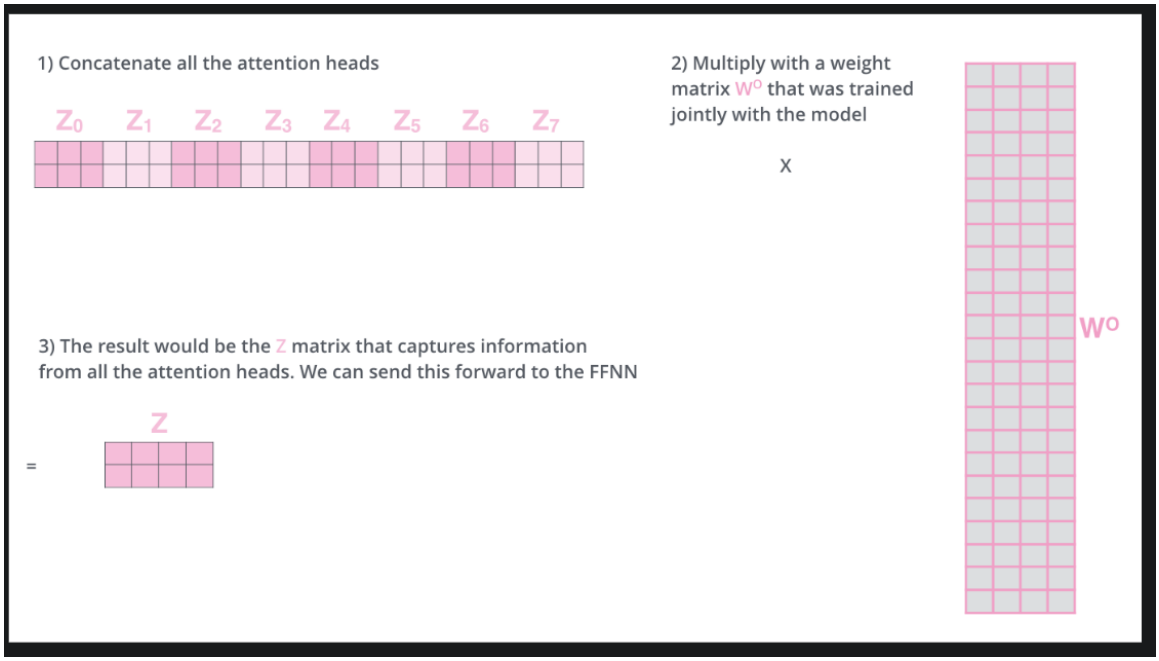


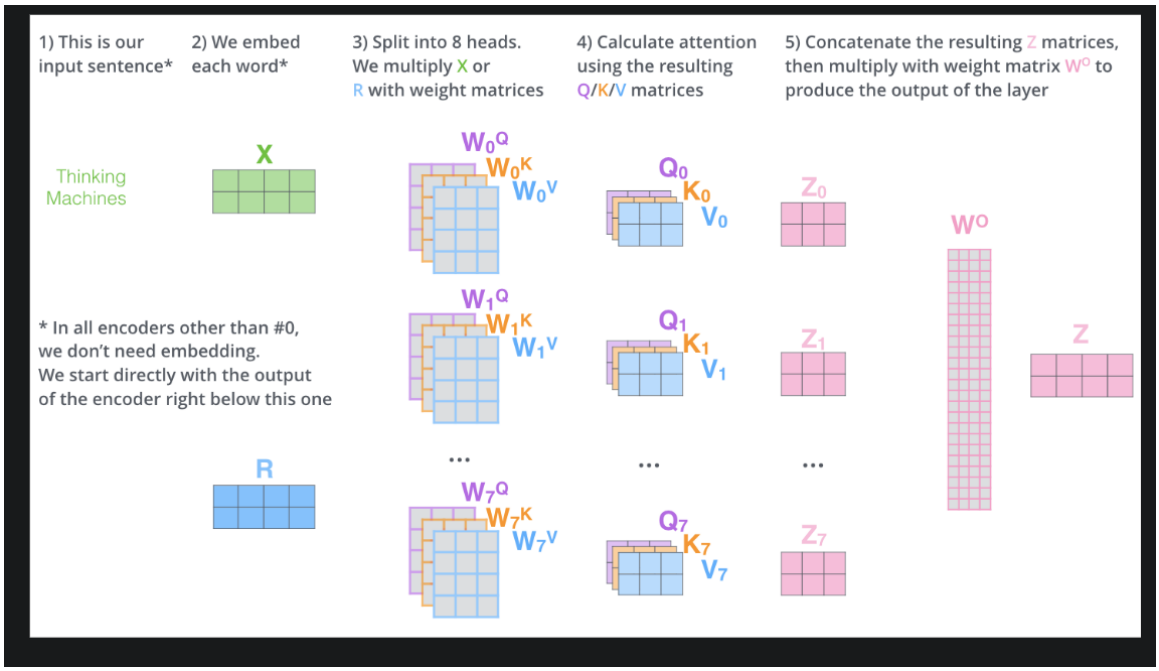
If we perform the self-attention calculation eight times using different weight matrices each time, we get eight different Z matrices.



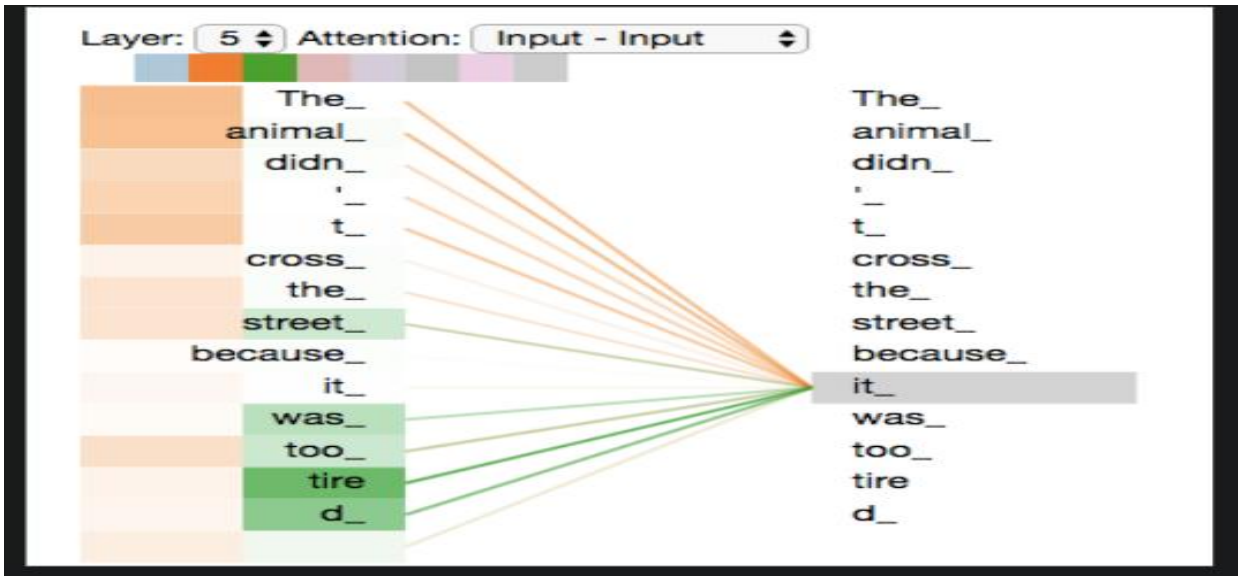
This creates a challenge since the feed-forward layer expects a single matrix (one vector per word), not eight. To solve this, we concatenate the eight matrices and then multiply the result by an additional weight matrix, W^O, to combine them into a single matrix.

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

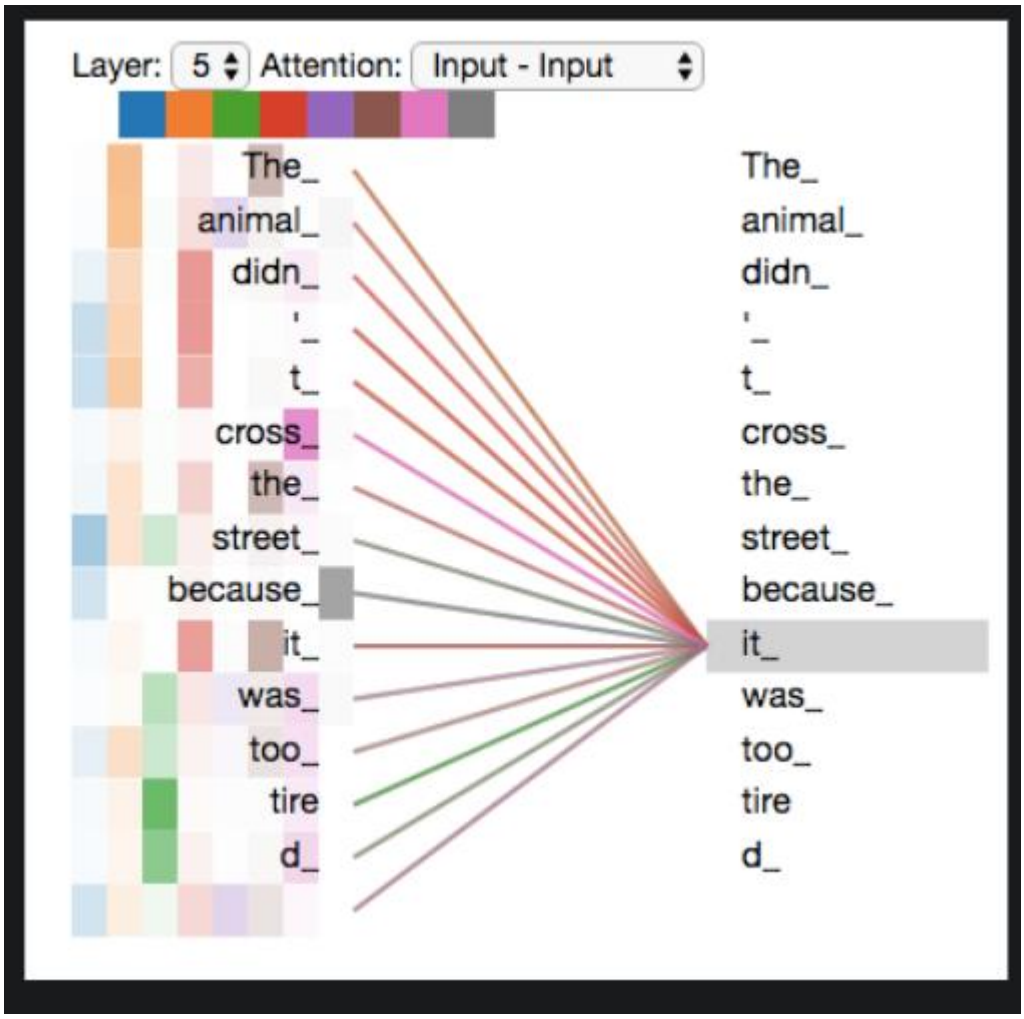




The word "it" in our example sentence



If we add all the attention heads to the picture, things can be harder to interpret:



Benefits of Multi-Head Attention

- **Joint Attention:** Each head focuses on different parts of the input, capturing varied relationships in the data.
- **Multiple Perspectives:** Multiple heads allow the model to simultaneously capture different aspects of the input, enhancing its understanding.
- **Efficiency:** Despite using multiple heads, each with smaller dimensions (e.g., 64), the computational cost remains similar to a single attention head.

Position-Wise Feed-Forward Networks

In the Transformer architecture, the **Position-wise Feed-Forward Networks (FFNs)** serve as a non-linear transformation applied after the attention sub-layers within both the encoder and decoder layers. Each FFN is applied independently to each token in the sequence and uses identical parameters across the sequence, but different parameters across different layers.

FFN Equation

The equation given for the feed-forward network is as follows:

$$FFN(sz) = \max(0, 2W1 + b1)W2 + b2$$

This can be interpreted as:

First Linear Transformation: The input xxx (of dimensionality d_{model}=512 is passed through a fully connected layer with weight matrix W_1 and bias b_1. This transforms the input from d_{model} to a higher dimensional space d_s=2048

$$h=xW_1 + b_1$$

ReLU Activation: A ReLU (Rectified Linear Unit) activation function is applied to the result of the first transformation. ReLU sets all negative values to zero, ensuring non-linearity in the transformation.

$$h_{ReLU}=\max(0,h)$$

2.Second Linear Transformation: The activated output is then linearly transformed again using a second weight matrix W2 and bias b2, reducing the dimensionality back to d_{model}=512.

$$FFN(x)=h_{ReLU}W_2+b_2$$

Embeddings and softmax layers

1.**Token Embeddings**

Transformers use **learned embeddings** to convert both input and output tokens into continuous vector representations of fixed dimensionality, d_{model}

- **Input Token Embeddings**: Each token in the input sequence is mapped to a d_{model}-dimensional vector using a learned embedding matrix.

- **Output Token Embeddings**: Similarly, each token in the output sequence (generated by the decoder) is mapped to d_{model}-dimensional vectors.

2.Shared Weight Matrix

- The Transformer model **shares the same weight matrix** for the input embeddings, output embeddings, and pre-softmax linear transformation. This means:
- The same matrix is used for encoding input tokens into embeddings and decoding embeddings back into tokens.
- This weight sharing not only reduces the number of parameters but also improves model generalization.

3. Scaling in the Embedding Layer

To stabilize training, the embedding vectors are scaled by a factor of $\sqrt{d_{model}}$

This is crucial because embeddings with larger dimensions tend to have higher magnitudes, which can affect the stability of the model's output.

Thus, for an embedding matrix W_e of size (vocab_size, d_model), the embedding for each token is calculated as:

$$\text{embedding}=\frac{We}{\sqrt{d_{model}}}$$

4. Softmax Output Layer

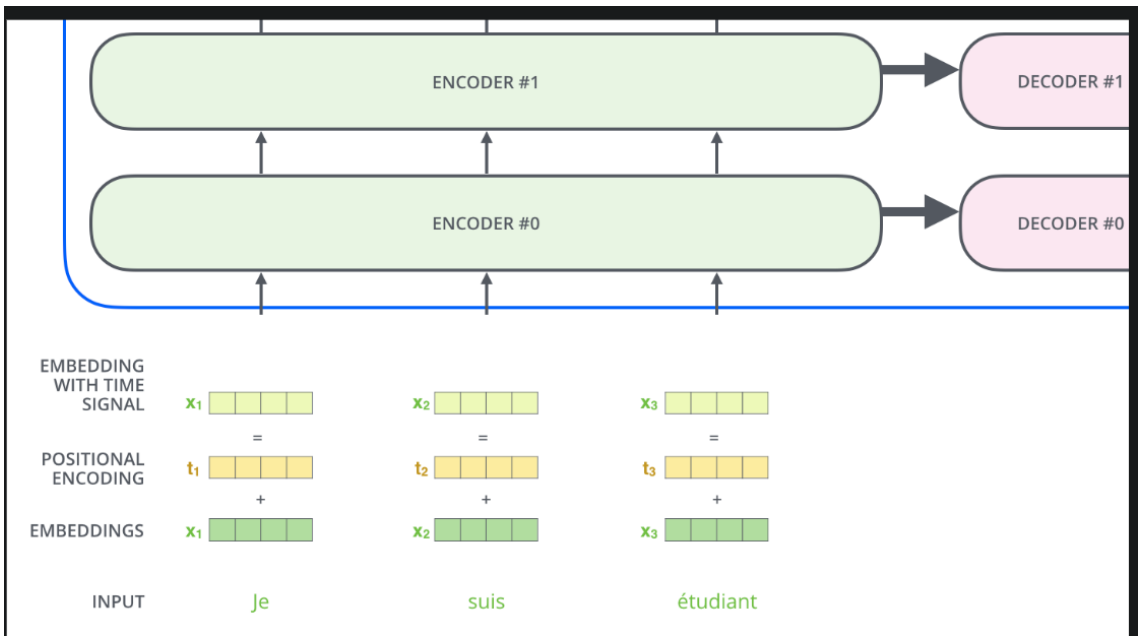
- The **output layer** of the decoder uses a linear transformation followed by a softmax function to predict the probability distribution over the vocabulary for the next token.
- **Linear Transformation**: The decoder's output vectors are multiplied by the shared embedding matrix (or weight matrix) and transformed into a vector of vocabulary size.
- **Softmax**: The softmax function is applied to this vector, producing probabilities for each token in the vocabulary, indicating the likelihood of each token being the next in the sequence.

Positional encoding

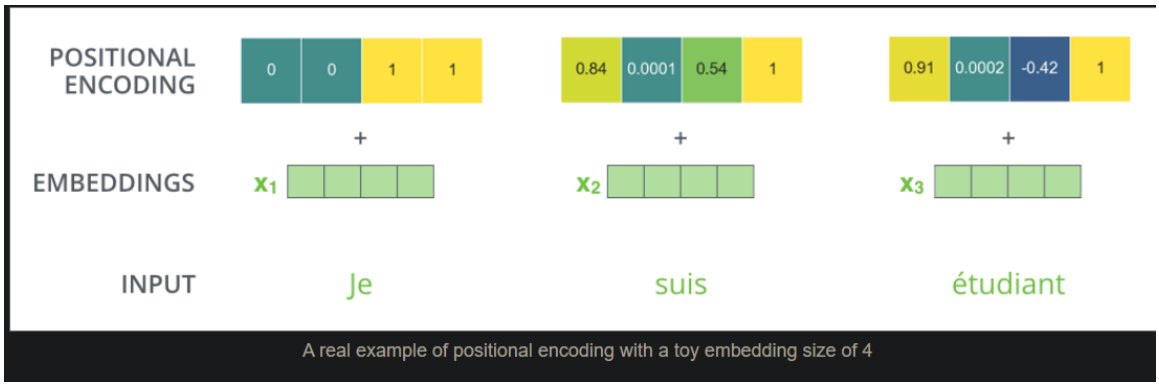
> In sequence models, understanding the order of tokens is essential. While recurrent models (like LSTMs) inherently handle order through sequential processing, Transformers use parallel processing without any recurrence. Thus, positional encodings are added to the embeddings at the input of the encoder and decoder to provide this sequence order information.

Representing The Order of The Sequence Using Positional Encoding

One thing missing from the model so far is a way to track the order of words in the input. To solve this, the Transformer adds a positional vector to each word embedding. These vectors follow a pattern that helps the model learn each word's position or distance between words. This added information helps keep meaningful distances between embedding vectors during their transformation into Q/K/V vectors and throughout the attention calculations.



If we assumed the embedding has a dimensionality of 4, the actual positional encodings would look like this:



A real example of positional encoding with a toy embedding size of 4

Characteristics of Positional Encoding

1. Dimension Match with Embeddings: Positional encodings have the same dimensionality as the input embeddings (d_{model}), so they can be directly added to them.
2. Fixed vs. Learned Positional Encodings: Positional encodings can be either:
    1. Fixed (as in the Transformer paper) with sine and cosine functions.
    2. Learned (trainable parameters), though the authors found that learned encodings yielded similar results to fixed encodings.

Sine and Cosine Positional Encoding Formula

$PE_{(pos,2i)}=\sin(\frac{pos}{100002i\ /dmodel})$

$PE_{(pos,2i+1)}=\sin(\frac{pos}{100002i\ /dmodel})$

Here:

pos: Position in the sequence (e.g., 1, 2, 3,...). i: Index of the dimension within the embedding vector.

d_{model}: The model's embedding dimension (e.g., 512).

The sinusoidal functions with varying frequencies create unique position values for each token's position in the sequence.

Even indices in the embedding dimension use sine, while odd indices use cosine.

Benefits of sinusoidal encoding

1.**Capturing word order** with unique positional vectors.

2. **Generalizing to different sequence lengths** due to its periodic nature.

3.**Being computationally efficient** as it doesn't require training.

4.**Encoding relative positions**, making it useful for tasks focused on word relationships.

4.**Ensuring smooth transitions** between positions, capturing subtle positional details.

Why Self-Attention

- **Computational Complexity**: Measures how efficiently each layer type processes information.
- **Parallelizability**: Assesses the ability to run computations concurrently.
- **Path Length for Long-Range Dependencies**: Shorter path lengths between distant positions make it easier to learn dependencies, a challenge in many sequence tasks.

Optimizer and Learning Rate Schedule

1. **Adam Optimizer**:
    1. The **Adam optimizer** was selected for its ability to handle sparse gradients and large parameter spaces efficiently. The chosen parameters are:
        1. $\beta 1=0.9$ and $\beta 2=0.98$
        2. $\epsilon=10^{-9}$
    2. These settings provide stability during training, with a focus on adapting the learning rate to reduce oscillations.
2. **Learning Rate Schedule**:
    1. The learning rate follows a **two-phase schedule**:
        1. **Warmup Phase**: During the first warmup_steps (set to **4000 steps**), the learning rate **increases linearly**. This gradual increase helps prevent large weight updates early in training, facilitating stable learning.

2. **Decay Phase**: After the warmup phase, the learning rate **decays according to the inverse square root of the step number**. This slow decay helps refine the model, avoiding drastic changes in later stages of training

- **Formula**:

$$lrate = d_{\text{model}}^{-0.5} \cdot \min(step\_num^{-0.5}, step\_num \cdot warmup\_steps^{-1.5})$$

- Here, $d^{-0.5}_{model}$ serves as a scaling factor for the base learning rate, while step_num and warmup_steps control the learning rate adjustment.

Regularization Techniques

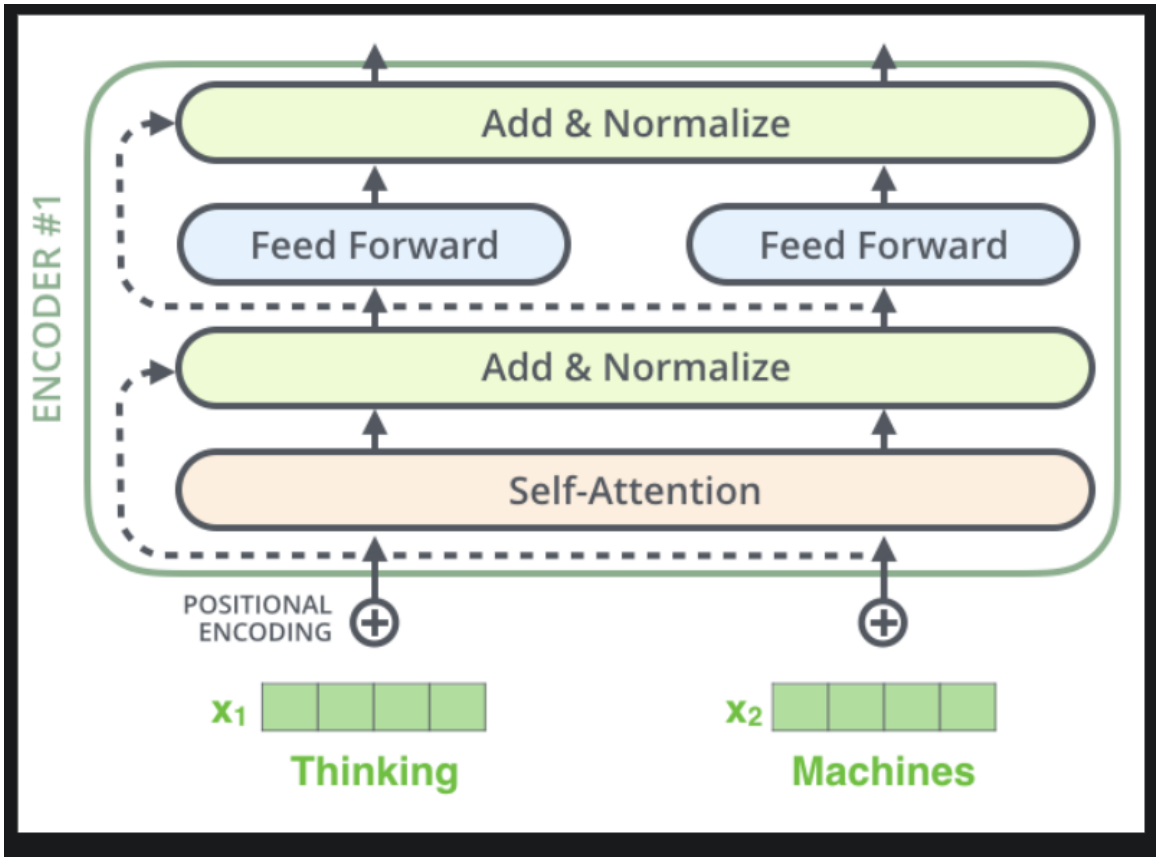
- To enhance generalization and reduce overfitting, three main regularization techniques are used during training in the Transformer model:

1. **Residual Dropout**:
    1. Dropout is applied to each **sub-layer's output** before it is added to the input and normalized. This strategy helps reduce the risk of overfitting by randomly omitting certain units during each training step.
    2. Additionally, dropout is also applied to the **sums of embeddings and positional encodings** in both the encoder and decoder stacks, ensuring more robust encoding and decoding of sequence information.
    3. For the **base model**, a dropout rate of p=0.1 is used.
2. **Label Smoothing**:
    1. **Label smoothing** of value ϵls=0.1 is applied. This technique adjusts the model's predictions slightly away from certainty by penalizing it for overconfidence in its outputs.
    2. Though this increases perplexity, label smoothing improves model **accuracy** and **BLEU score** by making the model more adaptable to variations in data, enhancing the robustness of its translations.

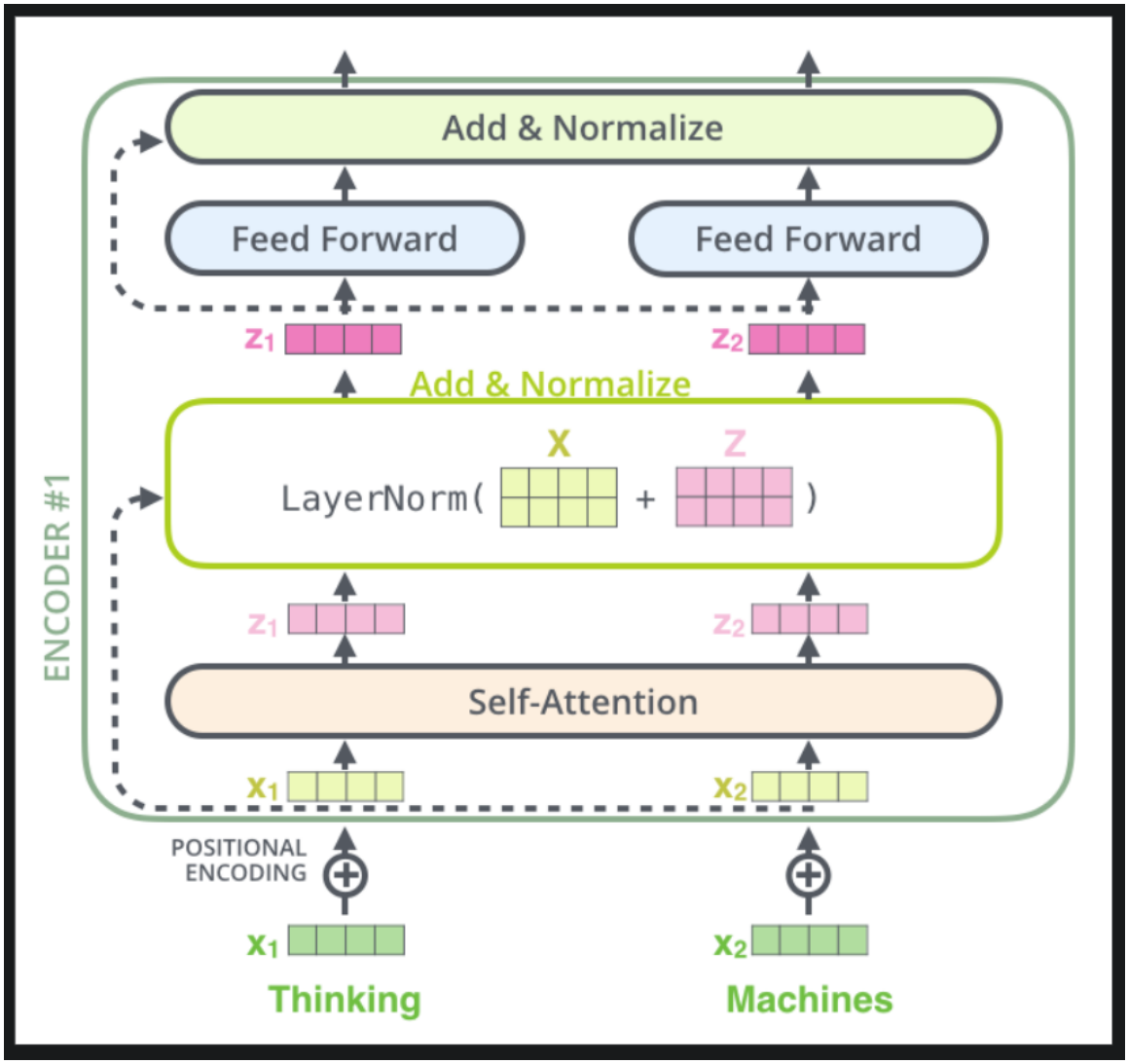
Sublayers in encoding and decoding

Encoder side

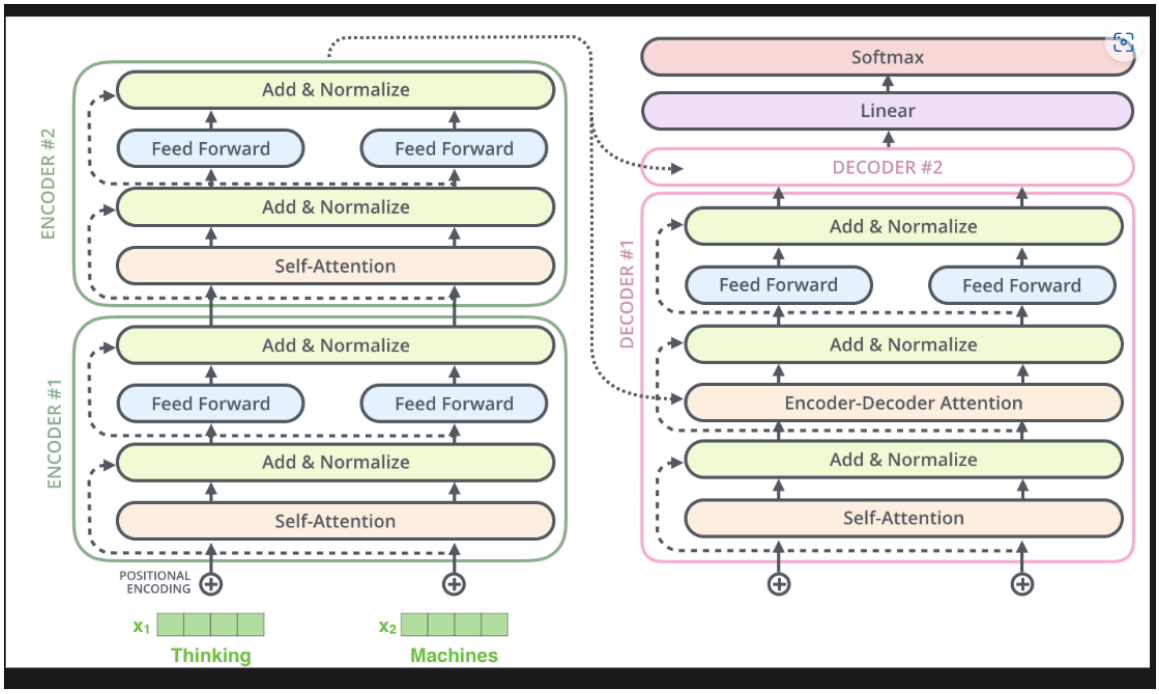
the encoder architecture is that each sub-layer (self-attention and feed-forward network) has a residual connection around it, followed by a layer normalization step.



To see the vectors and the layer-normalization operation in self-attention, it would look like this:

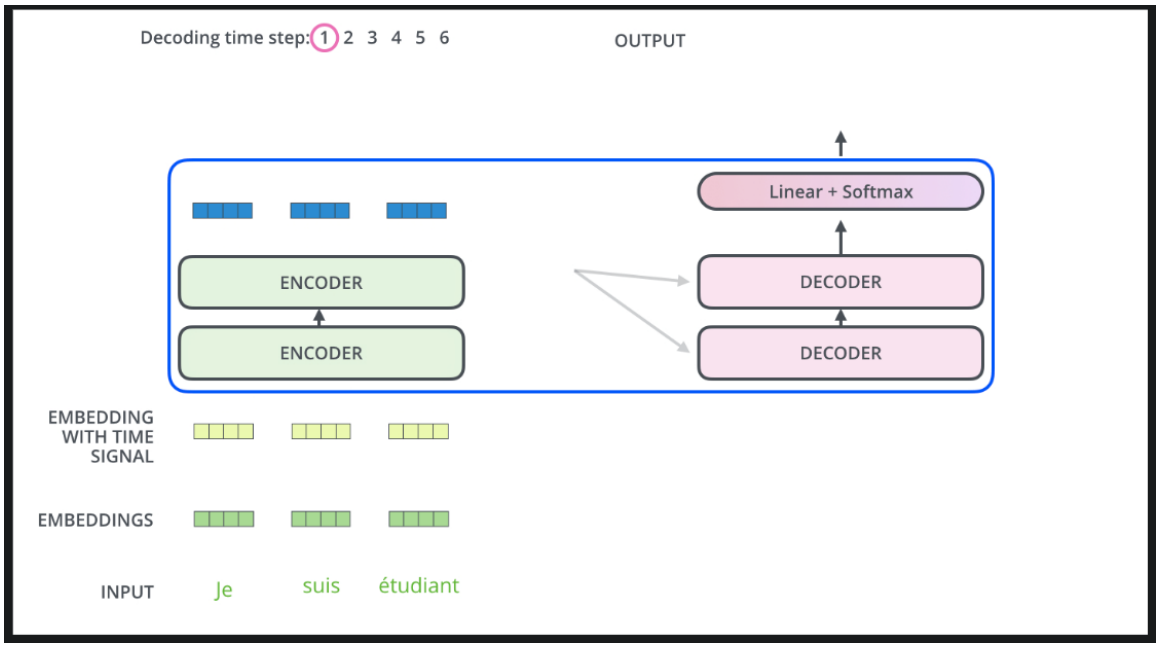


For a Transformer with 2 stacked encoders and decoders, the architecture would look like this:

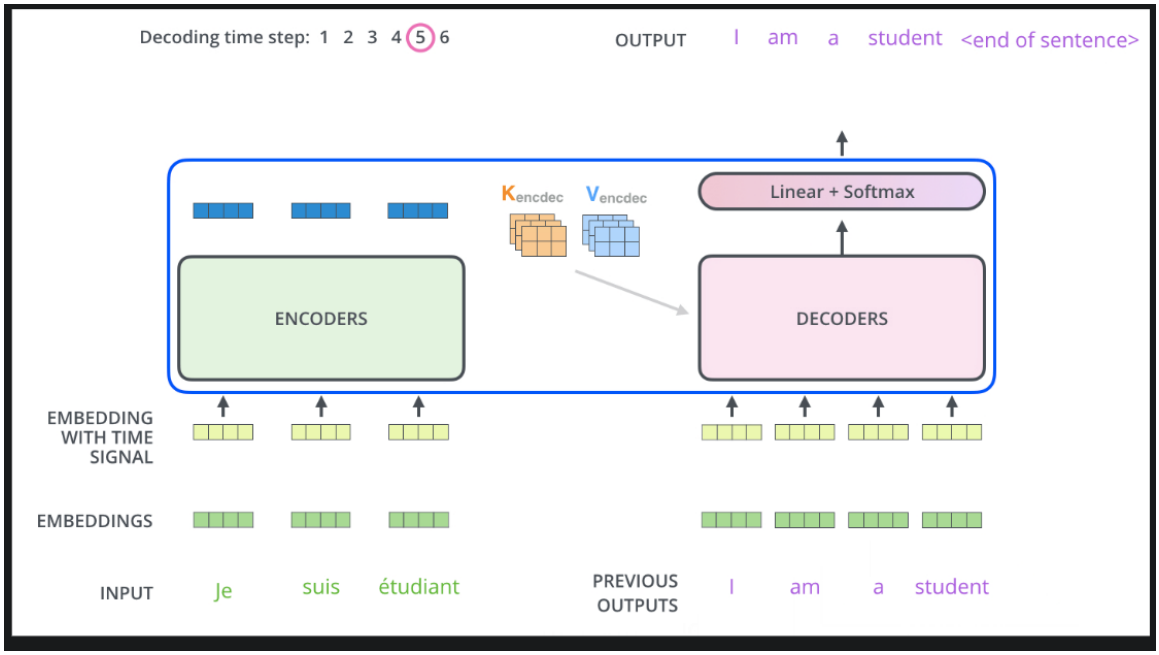


The Decoder Side

The encoder processes the input sequence and produces attention vectors (K and V). These are passed to each decoder's encoder-decoder attention layer, helping the decoder focus on the relevant parts of the input while generating the output.



The process continues until a special symbol is reached, signaling the end of the Transformer decoder's output. At each step, the output is passed to the next decoder, which processes it similarly to the encoder. Just like the encoder, we embed the decoder inputs and add positional encoding to show the position of each word.



The self-attention layers in the decoder work differently from those in the encoder:

- **Self-Attention in the Decoder:**
  In the decoder, the self-attention layer is limited to attending only to earlier positions in the sequence. This is done by **masking future positions** (setting them to negative infinity) before applying the softmax function in the attention calculation. This ensures that the model can only focus on previously generated words, not future ones.

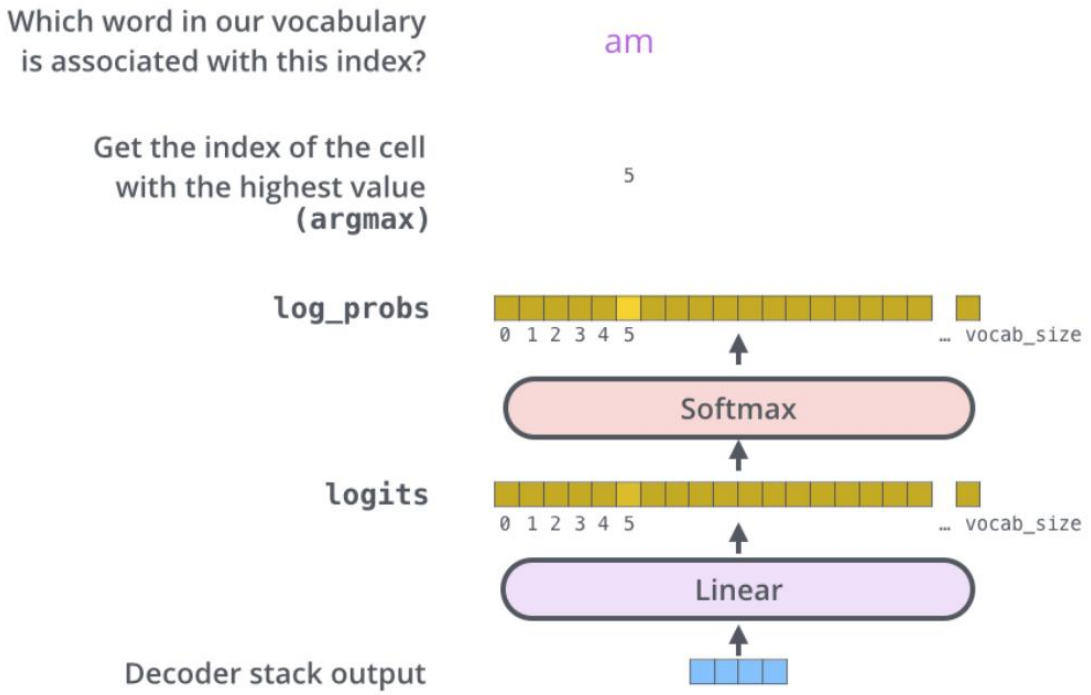
- **Encoder-Decoder Attention Layer:**
  This layer works similarly to multi-headed self-attention, but with a key difference:

  - It **creates the Queries matrix** from the previous decoder layer's output.
  - It **uses the Keys and Values matrices** from the encoder's output. This allows the decoder to integrate information from the encoder while generating the output.

The Final Linear and Softmax Layer

The decoder stack outputs a vector of floats, which is then processed by the final layers to generate a word:

1. **Linear Layer:**
    - This is a fully connected layer that projects the decoder output vector into a larger vector called the **logits vector**.
    - If the model knows 10,000 unique words, the logits vector would have 10,000 values, each representing the score for a specific word in the output vocabulary.
2. **Softmax Layer:**
    - The Softmax layer converts these logits (scores) into **probabilities**, ensuring that all values are positive and add up to 1.0.
    - The word with the highest probability is selected as the output for the current time step.



History and Evolution of Transformers

- **Background:**
  The Transformer model, introduced by Vaswani et al. in 2017, revolutionized Natural Language Processing (NLP) by replacing traditional recurrent and convolutional neural networks (RNNs and CNNs). Before the introduction of Transformers, Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) networks were the dominant models used in NLP. However, these models struggled with processing long-range dependencies effectively, limiting their ability to capture context in long sentences or paragraphs.

- **Shift to Attention Mechanisms:**
  The key breakthrough of the Transformer model was the use of self-attention, which enabled the model to consider different parts of the input sequence simultaneously, rather than sequentially. This allowed for faster processing and better capture of dependencies over long distances in the input text. This innovation led to the development of models like:

  - **BERT (Bidirectional Encoder Representations from Transformers): Focuses on masked language modeling, capturing bidirectional context for improved understanding of sentence meaning.**

  - **GPT (Generative Pre-trained Transformer): Utilizes an autoregressive approach for generating coherent and contextually relevant text.**

- **Milestones:**
  Each generation of Transformer-based models has pushed the boundaries of what is possible in NLP:

  - **BERT: Introduced bidirectional context understanding, allowing for more accurate comprehension of text.**

  - **GPT-3: Scaled up to a staggering 175 billion parameters, demonstrating unprecedented capabilities in language generation, such as producing coherent and contextually appropriate text across a wide range of applications.**

Large Language Models (LLMs)

A **Large Language Model (LLM)** is a type of deep learning model designed to understand, generate, and manipulate human language. These models are based on advanced neural network architectures, like the Transformer, and are trained on massive datasets containing text from diverse sources (e.g., books, articles, websites). They have billions or even trillions of parameters—adjustable weights that the model learns during training to better understand the patterns and nuances of language.

Large Language Models (LLMs), such as GPT, BERT, and T5, are built on the Transformer architecture. LLMs leverage the Transformer's ability to capture complex patterns and long-range dependencies within text.

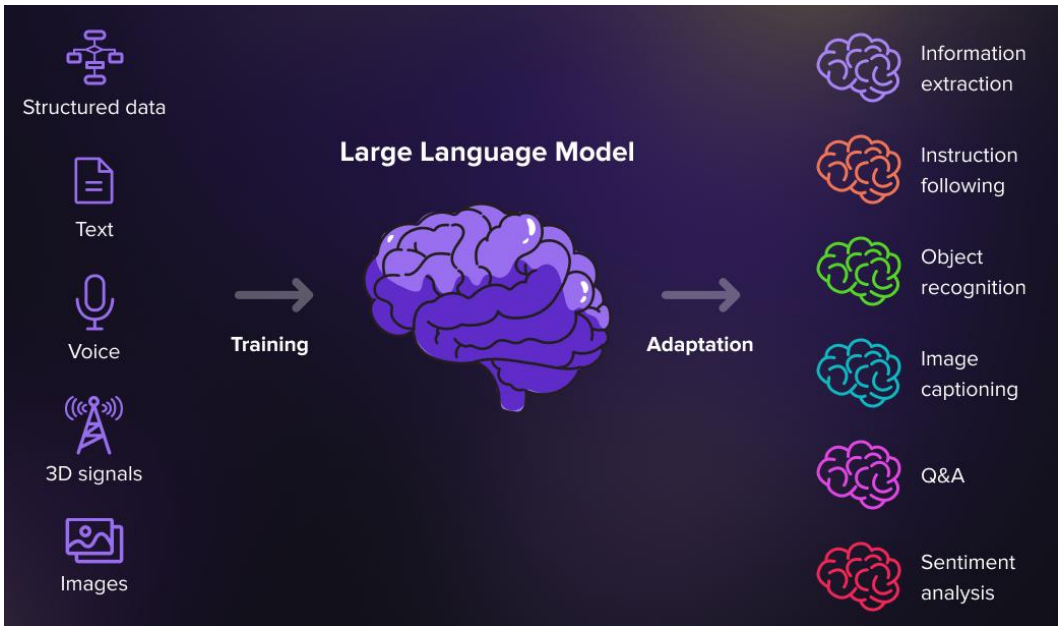
1. **Scalability**: The Transformer's self-attention mechanism allows LLMs to handle large datasets and billions of parameters by processing tokens in parallel.

2. **Contextual Understanding**: Self-attention within the Transformer lets LLMs learn relationships across sentences and paragraphs.

3. **Multi-Head Attention for Diverse Representations**: Multi-head attention in the Transformer gives LLMs multiple perspectives on the data, allowing each attention head to focus on different aspects of language, such as syntax, semantics, or long-range dependencies.

4. **Efficient Training and Inference**: The Transformer's design allows LLMs to train on extensive datasets efficiently and perform inference quickly. With parallel processing and reduced sequential dependencies,

5. **Pre-training and Fine-tuning**: LLMs are typically pre-trained on massive text corpora, learning general language patterns, and then fine-tuned for specific tasks.



Applications of transformers

1.**Image Processing**: In order to analyse and comprehend visual input, transformers have been modified for use in computer vision tasks.

2.Multimodal Tasks (Combining Text, Image, and Other Data):

Transformers have enabled the development of multimodal models that can process and integrate data from different sources, such as text, images, and even audio. These multimodal models are particularly powerful in applications where a single model must understand and relate information from multiple types of data.

Eg. **Image Captioning**, Visual Question Answering (VQA) ,Iext-to-Image and Image-to-Text Models.

3.Scientific and Technical Applications:

Transformers are also finding applications in scientific and technical domains where they handle complex, sequential, or structured data.

- **Genomics**: Transformers can analyze DNA sequences, which are essentially long strings of text, to help identify genes, understand genetic variations, and predict protein structures.
- **Chemistry and Drug Discovery**: Transformers are used to model molecular structures and predict chemical properties. For example, they can assist in predicting molecular interactions, which aids in the process of drug discovery.
- **Time-Series Analysis**: Transformers are being applied to time-series data (e.g., stock prices, weather data, and sensor readings), where they can capture both short-term and long-term patterns. Their self-attention mechanism allows them to identify important dependencies across different times, which is useful in forecasting and anomaly detection.

4.Audio and Speech Processing:
Transformers are also applied to audio and speech tasks, which require handling sequential audio data. Tasks include speech recognition (transcribing spoken language to text), speech synthesis (generating realistic-sounding speech), and even audio classification.

- **Speech Recognition**: Models like Speech Transformers adapt the Transformer architecture to transcribe spoken language accurately by learning the relationships between sounds over time.

- **Music Generation**: Transformers have been applied in generating music by treating notes and rhythms as sequences, enabling the creation of compositions that follow musical patterns.

Real-World Applications of Transformers and LLMs

- **Healthcare: Transformers are used to analyze medical texts, patient records, and research papers. LLMs help in summarizing patient history or even generating insights for clinical decision-making.**

- **Finance: Used in tasks like market prediction, sentiment analysis of news articles, and automated customer support.**

- **Customer Support (Chatbots): Transformers enable chatbots to understand and respond to customer inquiries, providing human-like responses and improving customer satisfaction.**

- **Language Translation: LLMs, particularly those based on the Transformer, achieve near-human accuracy in translating between languages.**

- **Creative Writing and Content Generation: GPT models generate text for articles, stories, and scripts, assisting writers and content creators.**

Challenges and Limitations of Transformers and LLMs

- **Computational Requirements: The Transformer's parallelization increases computational demand. Training large models like GPT-3 requires extensive GPU resources, making it costly.**
- **Data Dependency: Transformers need vast amounts of data to generalize well. This dependence often limits their usability in domains with limited data.**
- **Ethical and Social Concerns: Transformers, especially LLMs, can inherit biases from training data, which may lead to biased or harmful outputs. This highlights the need for careful dataset curation and ethical review processes.**
- **Model Interpretability: Transformer models are often considered "black boxes" as understanding how they make specific predictions is challenging.**

Future Directions and Advancements

- **Efficient Transformers: New architectures like BigBird and Longformer reduce computational cost and handle longer input sequences, making them more efficient and scalable.**
- **Reducing Model Size and Energy Efficiency: Efforts are underway to create smaller, energy-efficient models (e.g., DistilBERT, ALBERT) that perform similarly to larger models but with fewer resources.**
- **Improving Ethical Use: Ongoing research aims to reduce biases in LLMs and enhance model interpretability. Ethical AI labs are working on developing frameworks and guidelines for the responsible deployment of these models.**

Conclusion

The Transformer model's strengths in handling long sequences, efficient parallel processing, and flexibility for various tasks have made it a dominant model architecture not only in NLP but across many fields. By adapting its architecture to work with different types of data, researchers have extended its impact to applications in computer vision, multimodal tasks, scientific research, and audio processing. This adaptability highlights the versatility of Transformers and their potential to advance diverse areas in artificial intelligence and beyond.