Generative AI and LLMs

What is Attention?

Imagine you're reading a sentence and trying to understand the meaning of a particular word. You might look at other words in the sentence to get context. In NLP, the attention mechanism allows models to do something similar: focus on relevant parts of the input when processing information.

For example, in the sentence "The animal didn't cross the street because it was too tired," understanding what "it" refers to requires attention to "The animal." Attention mechanisms help models make these connections.

Attention is a mechanism that lets a model decide which parts of the input are important when processing or generating each output — rather than treating all inputs equally.

Imagine you're reading a sentence to answer a question like:

"What is the capital of France?"

You'd probably focus your attention on the word "France" in the sentence:

"Paris is the capital of France."

Even though the sentence has many words, your brain gives more weight to the most relevant word — "France" — to find the answer.

In a Transformer model:

When processing a word (say at position t), the model uses attention to look at all the other words in the sentence, and assigns a score to each — representing how important they are for understanding word t.

Step 1: Token: Breaking data into chunks (words into chunks). The process is called tokenization.

Step 2: Each token will carry its own weightage. Based on that weightage, attention will be given to different words.

How it works (briefly):

1. Each token gets transformed into three vectors: Query (Q), Key (K), and Value (V)
2. Attention computes scores between Query and Key → this tells us how much focus (or attention) to place on each token.
3. These scores are then used to weight the Value vectors → giving a weighted sum that becomes the attention output.

For example: in What is the capital of France

Query: Any token that is most important — eg. France

Key: All tokens including query (What is the capital of france)

Thus, model will pay attention to all the tokens as well as itself.

Value: In the context of attention mechanisms, especially within transformer models, "value" refers to **the actual content or information associated with a token that is used to construct the final output**. It's the "what" that's being attended to, while the "query" and "key" determine "how much" to attend to it.

Simple Explanation:

Imagine you're processing the sentence:

"The cat sat on the mat."

To understand the word "sat", the model looks at all other words in the sentence — "The," "cat," "on," "the," "mat" — and decides how much each contributes to understanding "sat."

In self-attention, every word:

Looks at every other word (including itself),

Decides how important each is,

Then combines that information to update its own representation.

Self Attention:

Why "self"?

Because the attention is within the same sequence — each token is attending to itself and its neighbors.

For each token in the input:

1. Create Query, Key, and Value vectors.
2. Use the Query of this token with Keys of all tokens to get attention scores.
3. Apply softmax to turn scores into weights.
4. Multiply these weights with Value vectors to get a new, context-aware representation of that token.

Self-attention lets each token in a sequence gather information from the entire sequence to better understand its context.

The formula for this step is:

$$\text{Attention Scores} = \frac{QK^T}{\sqrt{d_k}}$$

Where:

- Q is the Query matrix.
- K is the Key matrix.
- K^T is the transpose of the Key matrix.
- d_k is the dimension of the key vectors (and query vectors, as they typically have the same dimension in this context). The square root of d_k ($\sqrt{d_k}$) is the scaling factor.

- Thus, you create QKV, then calculate attention scores using Q and K. Then you turn scores to weights using softmax function. Weights * Value to get context aware representation of the token

Multi-Head Attention

What Is Multi-Head Attention?

Multi-head attention is a core concept in Transformer neural networks. It allows the model to focus on different parts of the input sequence simultaneously, capturing various relationships and patterns within the data. Rather than calculating attention just once, multi-head attention splits computations into several "heads," each with separate projections of Query (Q), Key (K), and Value (V) vectors. Each head processes the input differently, and the results are combined at the end.

https://www.geeksforgeeks.org/nlp/multi-head-attention-mechanism/

https://d2l.ai/chapter_attention-mechanisms-and-transformers/multihead-attention.html

Tutorial Notebook :

https://uvadlc-notebooks.readthedocs.io/en/latest/tutorial_notebooks/tutorial6/Transformers_and_MHAttention.html

- Self-attention pays attention to one way of thinking. Multi head attention has multiple heads, and all heads think differently. This is called multiple heads.

Summary in 1-liners:

Attention: Focus on another sequence to extract what matters.

Self-Attention: Focus within the same sequence to understand relationships.

Multi-Head Attention: Use several attention mechanisms in parallel to get richer context.

Transformer Architecture Overview

Core Components of a Transformer

1. Input Embeddings: Raw text is first tokenized into discrete tokens. Each token is mapped to a high-dimensional vector, known as its embedding.

Sources:

https://symbl.ai/developers/blog/a-guide-to-transformer-architecture/

https://www.geeksforgeeks.org/machine-learning/getting-started-with-transformers/

1. Positional Encoding: Since Transformers don't inherently process data sequentially, positional information is injected by adding positional encodings (often sine and cosine functions of varying frequencies) to each embedding, so the model knows the order of the sequence

Sources:

https://www.geeksforgeeks.org/machine-learning/getting-started-with-transformers/

Transformer Layers:

Most Transformers—especially in text-to-text tasks—are built with two major parts: an encoder and a decoder. Each consists of several identical layers (e.g., six encoder + six decoder layers in the original design).

1. **Encoder Layers**

Each encoder layer includes:

- **Multi-Head Self-Attention Mechanism:** Allows every token to "attend" to every other token and decide which are most relevant, from multiple perspectives ("heads").
- **Feedforward Neural Network (Position-wise FFN):** Applies the same dense two-layer perceptron to each token's representation, independently.
- **Residual Connections & LayerNorm:** Skip connections wrap both attention and feedforward sub-layers, stabilized by layer normalization.

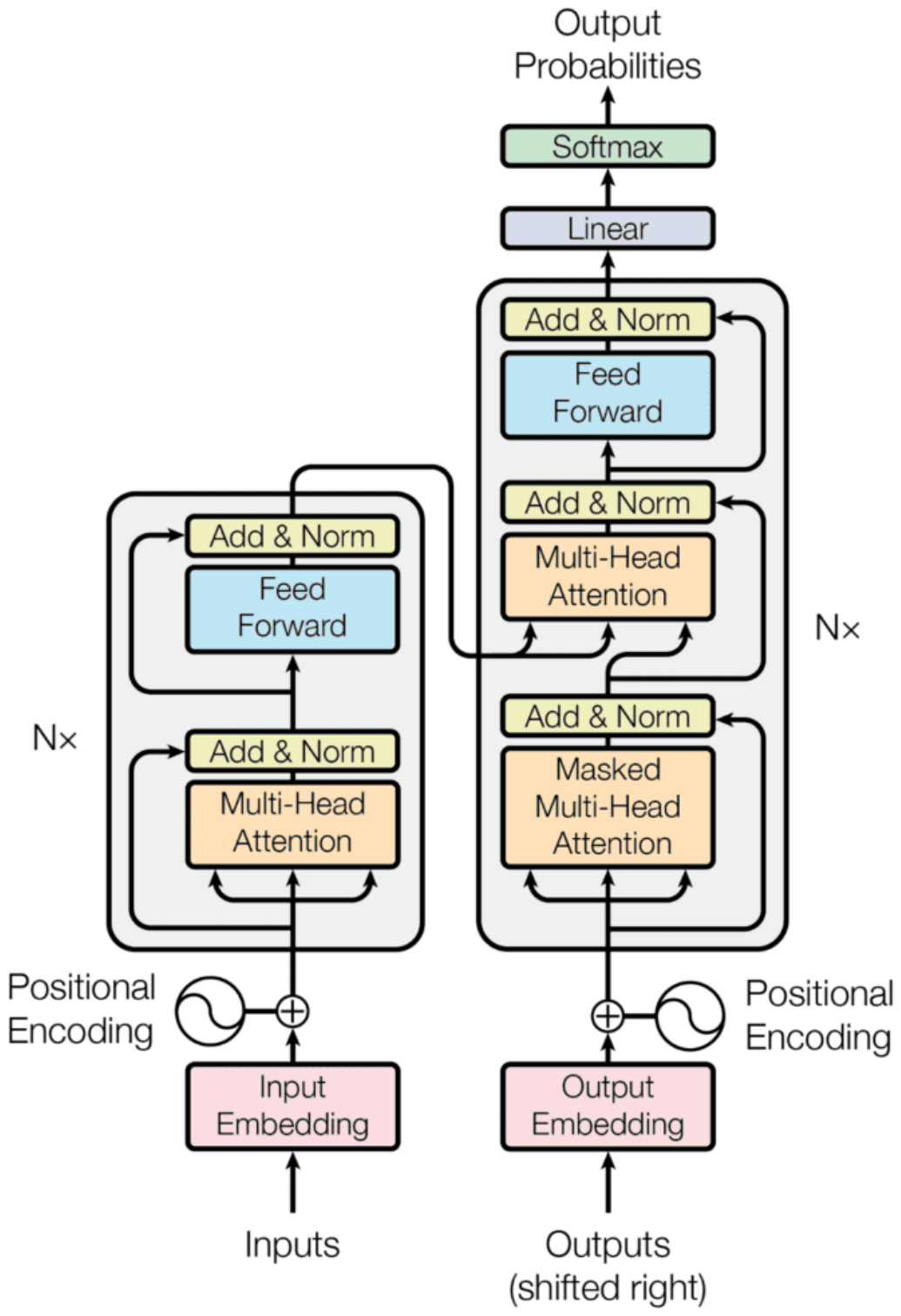
2. **Decoder Layers**

Each decoder layer includes:

- **Masked Multi-Head Self-Attention:** Ensures that future tokens cannot be seen, preserving autoregressive properties for sequence generation.
- **Encoder-Decoder Attention:** Lets each position in the decoder attend across all encoder outputs, learning correspondences between source and target sequences.
- **Feedforward Neural Network**
- **Residual Connections & LayerNorm**

3. **Output Layers**

- **Linear Projection:** The decoder output is linearly transformed into logits, mapping to vocabulary size.
- **Softmax Layer:** Converts logits into probabilities for predicting the next token



Masked Attention - Decoder Architecture

Masked attention modifies the standard attention mechanism by **blocking certain positions** in the input sequence from being attended to. This is done using a **mask**, usually a matrix of 0s and `-∞`s (or very negative numbers), which is added to the attention scores before applying softmax.

Autoregressive Masking (Causal Masking):

- Used in **GPT** and similar models.
- Prevents a token from seeing *future* tokens.
- Example: When predicting the 3rd word in a sentence, the model should not look at the 4th word and beyond.
- Implemented with a **triangular mask**.

Activation Function:

Converts scores or weights into probability values.

Handson:

1. Take a sentence
2. Break it into tokens using custom function

   def tokenize(text: str, vocab_size: int) → torch.Tensor:
   """Dummy text tokenizer."""
   #break into chunks
   words = text.split(" ")
   #give integer value to each chunk
   return torch.randint(0, vocab_size, [len(words)])

   #give a hyperparameter called vocab_size ⟶ this means that total number of unique tokens that model can recognize and process is 20000.
   VOCAB_SIZE = 20000

   tokenized_sentence = tokenize(sentence, VOCAB_SIZE)
   n_tokens = len(tokenized_sentence)
   tokenized_sentence
3. Generate Embeddings and convert input tokens into vectors
   EMBEDDING_DIM = 32

embedding_layer = nn.Embedding(VOCAB_SIZE, EMBEDDING_DIM)
embedded_tokens = embedding_layer(tokenized_sentence)
embedded_tokens.shape

Session 2: Embeddings, Causal Maskings

Let's break down the key points:

"the weights are scaled by the embedding dimension":

1. In the context of self-attention (a core component of transformers), the attention mechanism calculates "attention scores" (or "logits") between a query and all keys. These scores determine how much focus each part of the input sequence should get.
2. Before applying the softmax function, these attention scores are typically divided by the square root of the embedding dimension
3. Why? This scaling is crucial to prevent the dot products (which are used to calculate the attention scores) from becoming very large as the embedding dimension increases. When dot products become very large, they can push the softmax function into regions where its gradients are extremely small (saturate), which can lead to vanishing gradients and hinder the training process. Dividing by square root of d, helps to keep the variance of the dot products consistent, regardless of the embedding dimension size. This concept is detailed in the original "Attention Is All You Need" paper by Vaswani et al.

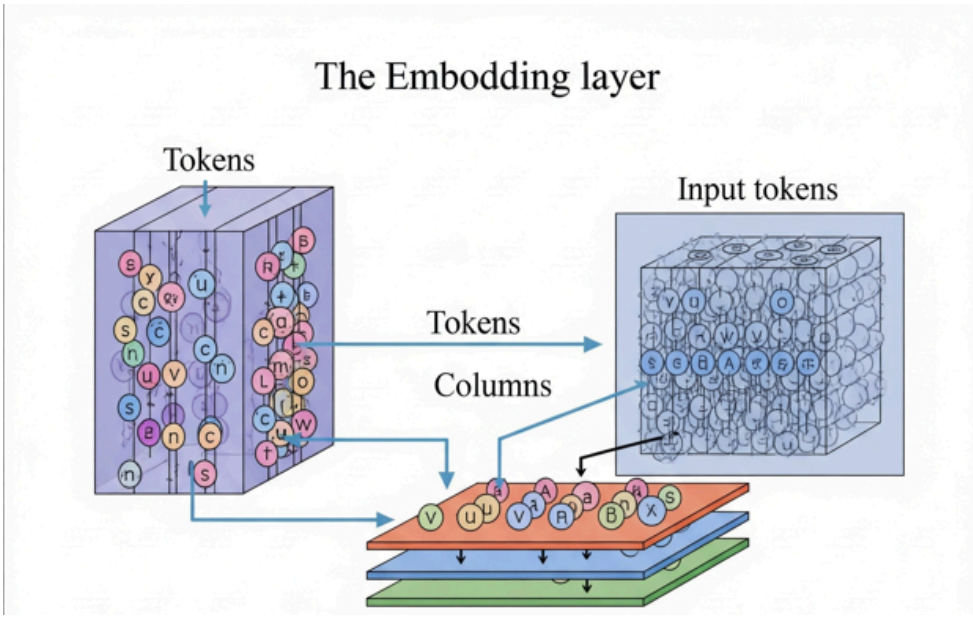
"and subsequently renormalised to sum to one across rows using the softmax function.":

1. After scaling, the attention scores are passed through a softmax function.
2. Why? The softmax function converts the raw attention scores into a probability distribution. This means that for each query, the attention weights assigned to all keys will be positive and sum up to 1. These normalized weights then determine how much each key contributes to the weighted sum that forms the attention output. This effectively creates a convex combination of the value vectors, where the weights indicate the "importance" of each value.

"Steps like these make models easier to train by normalising the magnitude of gradients used within algorithms like stochastic gradient descent.":

1. Normalization of Gradients: The scaling by square root of d and the use of softmax both contribute to a more stable training process.
2. Scaling: As mentioned, it prevents large attention scores from leading to saturated softmax outputs, which would result in near-zero gradients during backpropagation. By keeping the gradients in a reasonable range, the optimization algorithm (like Stochastic Gradient Descent - SGD) can make meaningful updates to the model's weights.
3. Softmax: By producing probabilities that sum to one, softmax ensures that the output of the attention mechanism is well-bounded. This helps in controlling the magnitude of activations and, consequently, the gradients flowing back through the network.
4. Easier Training: When gradients are well-behaved (not too large, not too small), optimizers can more effectively navigate the loss landscape, leading to faster convergence and better overall model performance. Issues like vanishing or exploding gradients can significantly impede or even halt the training process.

In summary, the described steps are fundamental to the stability and effectiveness of attention mechanisms in transformers. They address potential numerical issues that arise from large dot products, ensuring that the gradients during training remain in a healthy range, thus facilitating the training of these complex models with algorithms like SGD.



Causal Masking:

In a standard Transformer encoder's self-attention, each word can attend to all other words in the input sequence, both before and after it. This is great for understanding context in a static sentence.

However, imagine you're training a model to write a story, one word at a time. If, when predicting the third word, the model could "see" the actual fourth, fifth, or sixth words (which haven't been generated yet), it would be "cheating." It wouldn't be learning to predict genuinely based only on what has come before. This is a problem known as **data leakage or look-ahead bias.**

This isn't a problem if all we're doing is creating embeddings (or sequences) based on whole passages of text. It does pose a problem, however, if we're trying to develop a model that can generate new sequences given an initial sequence (or prompt).

- For tasks like text classification or summarization where you have the entire input available upfront, looking at future words is fine and even beneficial.
- But for tasks like predicting the next word in a sequence or machine translation decoding (generating the target sentence word by word), the model must only rely on information up to the current point. It must operate in an **autoregressive manner.**

The Solution: Causal Masking

"This problem is solved by using causal masking."
Causal masking (also known as **look-ahead masking or masked self-attention)** is a technique applied specifically in the self-attention mechanism of the Transformer's decoder (and decoder-only models like GPT).

Causal masking matrices can be constructed to flag which attention weights should be set to zero so that causal relationships between embeddings aren't broken.

- During the calculation of attention scores, a causal mask is applied. This mask is typically an upper triangular matrix (including the diagonal) filled with very large negative numbers (like negative infinity) in the positions corresponding to "future" tokens.
- When softmax is applied to these masked scores, the terms become effectively zero. This forces the attention weights for future tokens to be zero, meaning the current token cannot attend to any subsequent tokens in the sequence.

For example, when calculating the representation for the first word, it can only attend to itself. When calculating for the second word, it can attend to the first word and itself, but not the third, fourth, or fifth words, and so on.

How to Apply Causal Masking?

- Create a causal mask matrix that masks the future data

Apply Mask to the attention score: This causal_mask is precisely the mechanism to prevent the "cheating" we talked about in sequence generation.

In the matrix above, True at (row, col) means that the token at row (as a query) is forbidden from attending to the token at col (as a key).

Since True appears only for respective col and row (i.e., tokens that come after the current query token), this mask ensures that:

When the attention mechanism calculates the context for token 0, it can only use information from token 0 itself (not 1, 2, 3).
When calculating for token 1, it can use information from token 0 and 1, but not 2 or 3.
And so on.

During the attention calculation, this boolean mask is typically used to set the attention scores for the "forbidden" (future) connections to a very large negative number. After softmax, these large negative numbers effectively become zero, meaning the model assigns no attention to future tokens. This enforces the autoregressive property, allowing the model to learn to predict sequences step-by-step based only on past information.

Application of Scaling and normalization: By using softmax activation function

Day 1 and Day 2: https://github.com/sajagmathur/Generative-AI---Attentions-and-Transformers/tree/main/Day1

Day 3: Parameterized Self Attention

Parametrised Self-Attention

In Standard Self-Attention:

Each input token is projected into Query (Q), Key (K), and Value (V) vectors using learned linear transformations:

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

Then attention is computed as:

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

The matrices W^Q, W^K, W^V are learnable and are trained during the model's learning process.

Parametrised Self-Attention refers to a variation of the self-attention mechanism where additional learnable parameters are introduced into the attention computation. This allows the model to adaptively control how it distributes attention across input tokens.

What Makes Parametrised Self-Attention Different?

While standard self-attention already includes parameters (via the Q, K, V projections), the term **"parametrised self-attention" typically refers to further enhancements where:**

- Additional parameters are introduced into the attention score calculation or the softmax operation.
- Learnable biases, temperature scaling factors, or gating mechanisms are included.
- Multi-head attention components may be further parameterised with learned transformations or scaling terms.

Queries, Keys and Values

In this setup, the values contain the information that we wish to access via a query that is made on a set of keys (that map to the values), such that the context-aware embeddings can now be computed as,

$$
\vec{z_{i}} = \sum_{j=1}^{N}{a_{ij} \times \vec{v_{j}}}
$$

Where, $a_{ij} = q_{i}^{T} \cdot k_{i}$ - i.e., the attention weights now represent the distance between the query and keys.

Very often we only have a single sequence to work with, so the model will have to learn how to infer the queries, keys and values from this. We can enable this level of plasticity by defining three $N \times N$ weight matrices, $\textbf{U}*{q}$, $*textbf{U}*{k}$ and $\textbf{U}_{v}$.

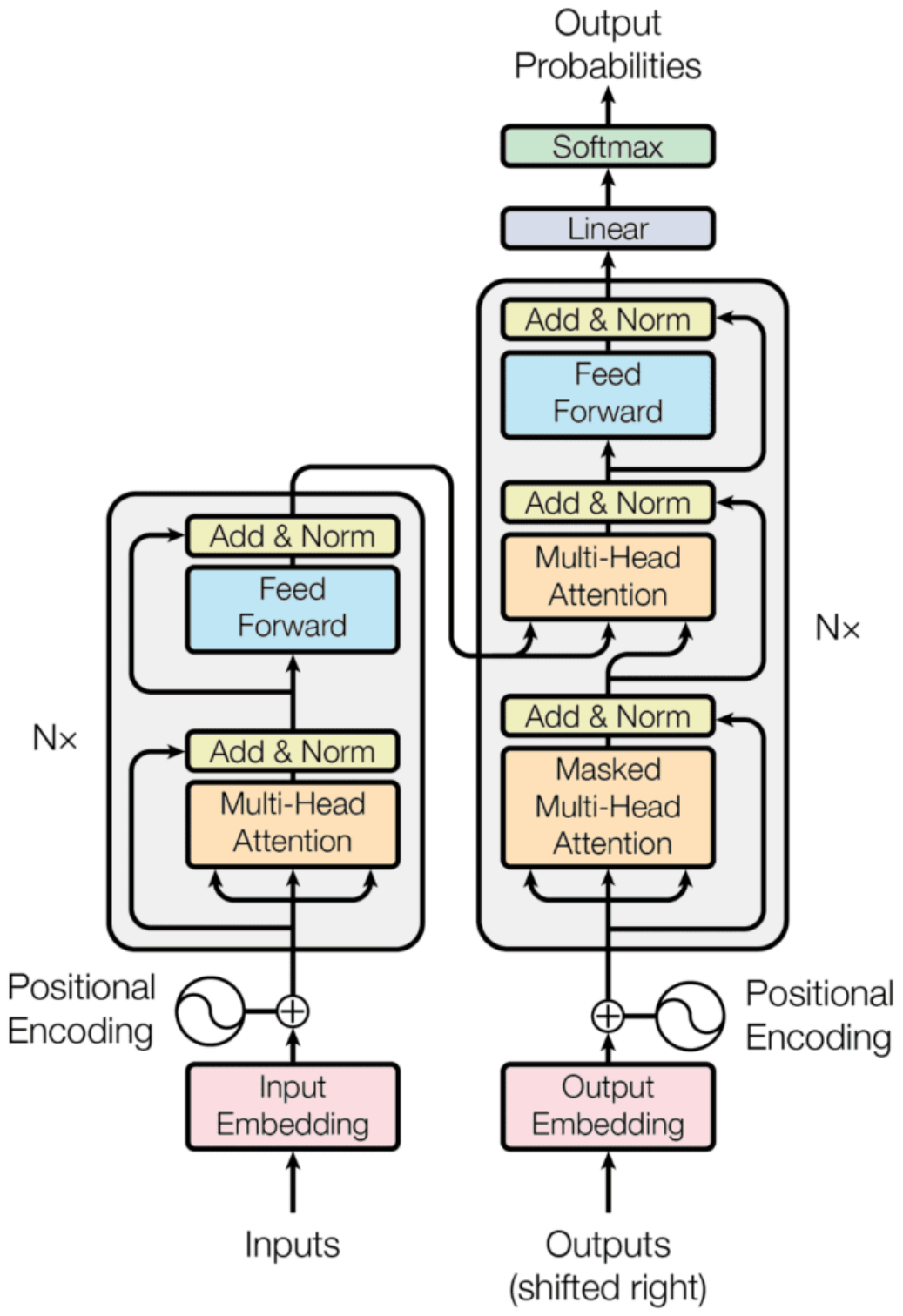
For model to be able to do its work better, you add a parameter to accelerate focus and make the model more dynamic. So, when we shift from single head to multihead attention, we apply learning parameters that give some context to data and give different things for different tokens.

Multi-Head Attention

In what follows we demonstrate how use the parametrised attention mechanism sketched out above to develop the multi-head attention block that forms the foundation of all transformer architectures. Our aim here is purely didactic - the functions defined below won't yield anything you can train (refer to the full codebase in the `modelling` directory for this), but they do demonstrate how these algorithm are composed.

We start by encapsulating the parametrised attention mechanism within a single function.

Till now, we have done the following:



How we designed **Encoder**:

- **Generated Embeddings**
- Fed it to **Multi Head Attentions**
- **Normalized**
- Fed to Decoder
- **Masking is a part of decoder**

Query comes from **Decoder, Keys and Values come** from Encoder

- Final output goes to Softmax layer and output helps us generate new word!

Complete Day 3 Handson:

- Guided project on transformer architecture: https://uvadlc-notebooks.readthedocs.io/en/latest/tutorial_notebooks/tutorial6/Transformers_and_MHAttention.html
- https://uvadlc-notebooks.readthedocs.io/en/latest/tutorial_notebooks/tutorial6/Transformers_and_MHAttention.html

Session 4:

1. Pos Encoding
2. Pre-trained model
3. Transformer Playground
4. Post assessment

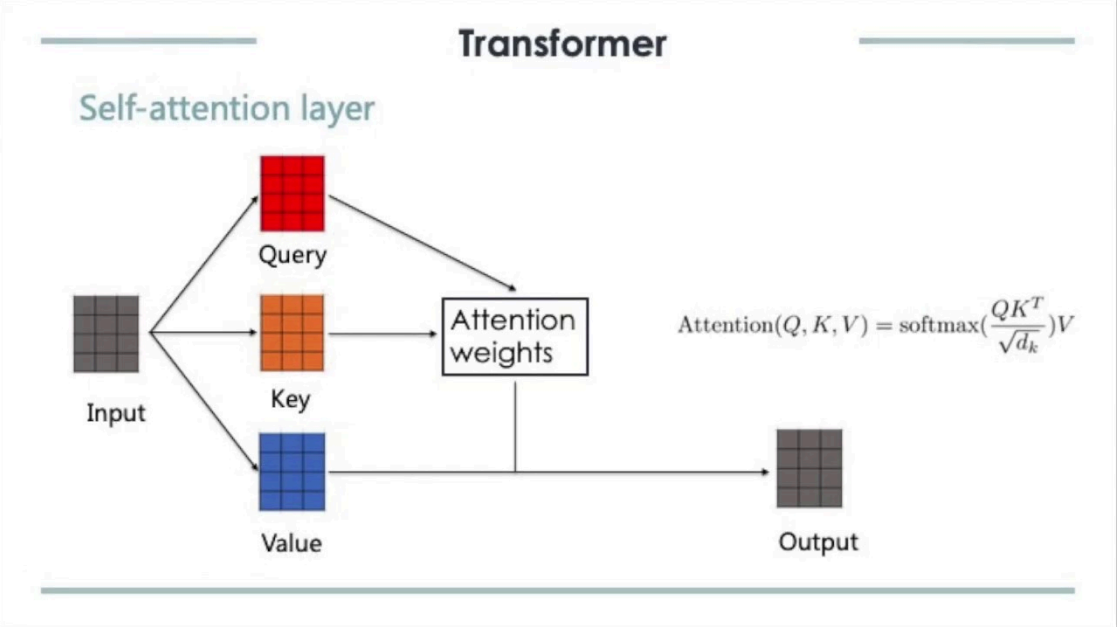
Transformers" in the context of artificial intelligence are a type of computer model designed to understand and generate human language. They're really good at tasks like translating languages, answering questions, and generating text.

- Transformers rely on a mechanism called "self-attention" to weigh the importance of different words in a sentence when processing language data. This mechanism allows them to capture long-range dependencies and relationships between words more effectively than previous models. As a result, Transformers have achieved state-of-the-art performance in many NLP tasks, including language translation, text summarization, question answering, and sentiment analysis.
- Self-attention is a technique used in NLP that helps models to understand relationships between words or entities in a sentence, no matter where they appear. It is a important part of transformers model which is used in tasks like translation and text generation.

Understanding Attention in NLP

- The goal of self attention mechanism is to improve performance of traditional models such as encoder-decoder models used in RNNs (Recurrent Neural Networks).

- In traditional encoder-decoder models input sequence is compressed into a single fixed-length vector which is then used to generate the output.
- This works well for short sequences but struggles with long ones because important information can be lost when compressed into a single vector.



Self-Attention Mechanism Explained

The **self-attention mechanism** is a key innovation behind models like **Transformers** (e.g., BERT, GPT). It allows models to weigh the importance of different words in a sequence **relative to each other**.

---

1. What is Self-Attention?

- Self-Attention lets a model **attend** to all positions of a sequence to compute a representation of that sequence.
- It's used to model **dependencies** between tokens, even if they are far apart.
- It allows a model to dynamically focus on **relevant parts** of the input for each token.

---

2. Core Idea

Given an input sequence of token embeddings:

[X = [x_1, x_2, ..., x_n]]

We transform these into **Query (Q)**, **Key (K)**, and **Value (V)** vectors:

[Q = XW^Q,\quad K = XW^K,\quad V = XW^V]

Where (W^Q, W^K, W^V) are learnable weight matrices.

---

3. Scaled Dot-Product Attention

For each token:

1. Compute attention scores using **dot product** between its Query and all Keys: [\text{AttentionScore}(i,j) = Q_i \cdot K_j^T]
2. Scale the scores: [\text{ScaledScore} = \frac{QK^T}{\sqrt{d_k}}]
3. Apply **softmax** to get attention weights: [\text{AttentionWeights} = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} \right)]
4. Multiply by **Values**: [\text{Output} = \text{AttentionWeights} \cdot V]

---

4. Multi-Head Attention

Instead of performing a single attention function, we run multiple in parallel (called "heads"):

[\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O]

Each head learns **different representations**, helping the model attend to information from **different subspaces**.

---

5. Role in Transformers

- Self-Attention is used in **Encoder** and **Decoder** blocks.
- Captures **contextual information** for each token efficiently.
- Enables **parallelization** (vs. RNNs).

Key Advantages

- **Long-range dependency modeling**
- **Parallel computation**
- **Dynamic contextualization** of words
- Scales better than recurrent models

. Summary

Component	Role
Query (Q)	Current word's focus
Key (K)	Candidate words to attend to
Value (V)	Contains information to combine
Softmax	Normalizes attention scores
Multi-head	Allows model to attend to different perspectives

The key components of Transformer models include:

-

1. **Self-Attention Mechanism**: This is the core component of Transformers. Self-attention allows the model to weigh the importance of different words in a sentence when processing language data. It enables capturing contextual relationships between words in a sequence, facilitating better understanding of the input.

- 1. **Multi-Head Attention**: In Transformers, self-attention is typically used in multiple "heads" or parallel attention mechanisms. Each head allows the model to focus on different parts of the input, enabling it to capture different types of relationships simultaneously.

- 1. **Positional Encoding**: Since Transformer models do not inherently understand the sequential order of input tokens like recurrent neural networks (RNNs), positional encoding is added to the input embeddings to provide information about the position of each token in the sequence.

- 1. **Feedforward Neural Networks**: Transformers include feedforward neural networks as part of their architecture. These networks are applied independently to each token's representation after self-attention and positional encoding, allowing the model to capture non-linear relationships between features.

- 1. **Encoder and Decoder Layers**: Transformer architectures often consist of encoder and decoder layers. The encoder processes the input sequence, while the decoder generates the output sequence in tasks like sequence-to-sequence translation. Each layer in the encoder and decoder typically includes self-attention and feedforward neural network sub-layers.

- 1. **Residual Connections and Layer Normalization**: To facilitate training deep networks, Transformers use residual connections around each sub-layer followed by layer normalization. These techniques help alleviate the vanishing gradient problem and improve the flow of information through the network.

- 1. **Masking**: In tasks like language translation, where the entire input sequence is available during training, masking is applied to prevent the model from attending to future tokens when predicting the output sequence.

These components work together to enable Transformers to achieve state-of-the-art performance in various natural language processing tasks.

Positional encoding:

Positional encoding is a crucial concept in the Transformer architecture, enabling the model to capture the order of words in a sequence since Transformers lack inherent sequential information. Let's break down the concept using a simple example with the sentence "Monika likes coffee".

1. Understanding Positional Encoding In a Transformer model, each word is embedded into a high-dimensional space using an embedding matrix. However, the position of each word in the sequence is not captured by these embeddings. Positional encoding addresses this by adding a unique positional information to each word embedding.
2. Positional Encoding Formula The most common method for positional encoding in Transformers involves using sine and cosine functions of different frequencies. For a position $p\ o\ s$ pos and a dimension i i of the encoding, the positional encoding is defined as:

$$\mathrm{PE}(pos, 2i) = \sin\left(\frac{pos}{10000^{2i/d}}\right)$$
$$\mathrm{PE}(pos, 2i+1) = \cos\left(\frac{pos}{10000^{2i/d}}\right)$$

where:

- pos is the position in the sequence,
- i is the dimension,
- d is the total number of dimensions.

$$\mathrm{PE}(pos, 2i) = \sin\left(\frac{pos}{10000^{2i/d}}\right)$$
$$\mathrm{PE}(pos, 2i+1) = \cos\left(\frac{pos}{10000^{2i/d}}\right)$$

where:

- pos is the position in the sequence,
- i is the dimension,
- d is the total number of dimensions.

3. Example: "Monika likes coffee"

- Let's consider a simple example where we encode a small part of the sequence using positional encoding. Assume the sentence "MOnika likes coffee" has four positions (0, 1, 2, 3), and we use a small dimension d=4 for simplicity.
-

For position $pos = 0$:

$$\text{PE}(0,0) = \sin\left(\frac{0}{10000^{0/4}}\right) = \sin(0) = 0$$
$$\text{PE}(0,1) = \cos\left(\frac{0}{10000^{1/4}}\right) = \cos(0) = 1$$
$$\text{PE}(0,2) = \sin\left(\frac{0}{10000^{2/4}}\right) = \sin(0) = 0$$
$$\text{PE}(0,3) = \cos\left(\frac{0}{10000^{3/4}}\right) = \cos(0) = 1$$

For position $pos = 1$:

$$\text{PE}(1,0) = \sin\left(\frac{1}{10000^{0/4}}\right) = \sin(1)$$
$$\text{PE}(1,1) = \cos\left(\frac{1}{10000^{1/4}}\right)$$
$$\text{PE}(1,2) = \sin\left(\frac{1}{10000^{2/4}}\right)$$
$$\text{PE}(1,3) = \cos\left(\frac{1}{10000^{3/4}}\right)$$

For position $pos = 2$:

$$\text{PE}(2,0) = \sin\left(\frac{2}{10000^{0/4}}\right) = \sin(2)$$
$$\text{PE}(2,1) = \cos\left(\frac{2}{10000^{1/4}}\right)$$
$$\text{PE}(2,2) = \sin\left(\frac{2}{10000^{2/4}}\right)$$
$$\text{PE}(2,3) = \cos\left(\frac{2}{10000^{3/4}}\right)$$

For position $pos = 3$:

$$\text{PE}(3,0) = \sin\left(\frac{3}{10000^{0/4}}\right) = \sin(3)$$
$$\text{PE}(3,1) = \cos\left(\frac{3}{10000^{1/4}}\right)$$
$$\text{PE}(3,2) = \sin\left(\frac{3}{10000^{2/4}}\right)$$
$$\text{PE}(3,3) = \cos\left(\frac{3}{10000^{3/4}}\right)$$

For position $pos = 0$:

$$\text{PE}(0,0) = \sin\left(\frac{0}{10000^{0/4}}\right) = \sin(0) = 0$$
$$\text{PE}(0,1) = \cos\left(\frac{0}{10000^{1/4}}\right) = \cos(0) = 1$$
$$\text{PE}(0,2) = \sin\left(\frac{0}{10000^{2/4}}\right) = \sin(0) = 0$$
$$\text{PE}(0,3) = \cos\left(\frac{0}{10000^{3/4}}\right) = \cos(0) = 1$$

For position $pos = 1$:

$$\text{PE}(1,0) = \sin\left(\frac{1}{10000^{0/4}}\right) = \sin(1)$$
$$\text{PE}(1,1) = \cos\left(\frac{1}{10000^{1/4}}\right)$$
$$\text{PE}(1,2) = \sin\left(\frac{1}{10000^{2/4}}\right)$$
$$\text{PE}(1,3) = \cos\left(\frac{1}{10000^{3/4}}\right)$$

For position $pos = 2$:

$$\text{PE}(2,0) = \sin\left(\frac{2}{10000^{0/4}}\right) = \sin(2)$$
$$\text{PE}(2,1) = \cos\left(\frac{2}{10000^{1/4}}\right)$$
$$\text{PE}(2,2) = \sin\left(\frac{2}{10000^{2/4}}\right)$$
$$\text{PE}(2,3) = \cos\left(\frac{2}{10000^{3/4}}\right)$$

For position $pos = 3$:

$$\text{PE}(3,0) = \sin\left(\frac{3}{10000^{0/4}}\right) = \sin(3)$$
$$\text{PE}(3,1) = \cos\left(\frac{3}{10000^{1/4}}\right)$$
$$\text{PE}(3,2) = \sin\left(\frac{3}{10000^{2/4}}\right)$$
$$\text{PE}(3,3) = \cos\left(\frac{3}{10000^{3/4}}\right)$$

Resulting Positional Encodings

Combining these, we get the positional encodings for each position

- Position 0: $[0, 1, 0, 1]$
- Position 1: $[\sin(1), \cos(1), \sin(1/100), \cos(1/100)]$
- Position 2: $[\sin(2), \cos(2), \sin(2/100), \cos(2/100)]$
- Position 3: $[\sin(3), \cos(3), \sin(3/100), \cos(3/100)]$

- Position 0: $[0, 1, 0, 1]$
- Position 1: $[\sin(1), \cos(1), \sin(1/100), \cos(1/100)]$
- Position 2: $[\sin(2), \cos(2), \sin(2/100), \cos(2/100)]$
- Position 3: $[\sin(3), \cos(3), \sin(3/100), \cos(3/100)]$

If the word embeddings for "monika", "likes", "coffee" are vectors, the positional encoding vectors would be added to these embeddings. This addition ensures that the model is aware of the position of each word in the sentence.

- In practice, these operations are done over higher-dimensional spaces and with many more positions, but the fundamental idea remains the same. The positional encodings help the Transformer model understand the order and position of words within a sequence

Now we will use a pre trained data to generate content:

Simplified Python code demonstrating how self-attention might be applied in a neural machine translation scenario using the transformers library

- This code uses the BERT model from the transformers library to tokenize the input text, compute its hidden states, and extract the self-attention weights. These weights indicate how much each token attends to every other token in each layer of the model. However, note that BERT is not specifically trained for machine translation, so this is just an illustration of self-attention in a language model context.

Steps

1. Tokenization: The input text "Monika likes coffee." is tokenized into its constituent tokens using the BERT tokenizer. Each token is represented by an integer ID. Let's denote the tokenized input as X
   X.
2. Model Computation: The tokenized input X
3. X is fed into the BERT model, which consists of multiple layers of self-attention and feedforward neural networks. The BERT model processes the input tokens and produces hidden states for each

token. Let's denote the hidden states as H

4. Self-Attention: During each layer of the BERT model, self-attention is applied to the input tokens. The self-attention mechanism computes attention scores between each token and every other token in the sequence. These attention scores are calculated using the formula:
5. Self-Attention Weights: The self-attention weights represent the importance of each token attending to every other token in the sequence. These weights are computed for each layer of the model. In the code, the mean of the attention weights across the sequence dimension is calculated for each layer and printed out.

When you use pre trained models: You can fine tune the model, but you don't retrain the model. You just use the pretrained model. You use the exact name of the model to call the model.

- You take pretrained models

```
import torch
from transformers import BertModel, BertTokenizer, BertConfig
```

Load pre-trained BERT model and tokenizer

```
machine_T = 'bert-base-multilingual-cased'
tokenizer = BertTokenizer.from_pretrained(machine_T )
model = BertModel.from_pretrained(machine_T )
```

Input text

```
input_text = input("enter your text : ")
```

Tokenize input text - Adjust and apply weights

```
input_ids = tokenizer.encode(input_text, add_special_tokens=True, return_tensors="pt")
```

Get BERT model's output

```
outputs = model(input_ids)
```

Configurations and Input Data

##Check configurations of model

Check if the model supports attention weights

```
config = BertConfig.from_pretrained(machine_T )
if config.output_attentions:
# Extract hidden states
hidden_states = outputs.last_hidden_state
```

```
# Self-attention
self_attention_weights = outputs.attentions

# Print self-attention weights
print("Self-attention weights:")
for layer, attn_weights in enumerate(self_attention_weights):
    print(f"Layer {layer+1}: {attn_weights.mean(dim=1)}")
```

Decoding input text

decoded_output = tokenizer.decode(input_ids[0], skip_special_tokens=True)
print("Decoded output from positional encoder:", decoded_output)

Get the model's final output

final_output = outputs[0]

Print the final output

print("Final output:", final_output)

print("Shape of the final Output",final_output.shape)

Example of Language Translation using Marin Model

Import relevant libraries

import torch
from transformers import MarianMTModel, MarianTokenizer

Load pre-trained MarianMT model and tokenizer for English to Hindi translation

model_name = 'Helsinki-NLP/opus-mt-en-hi'
tokenizer = MarianTokenizer.from_pretrained(model_name)
hindi_tran = MarianMTModel.from_pretrained(model_name)

Input text

input_text = input("Enter text for transalation = ")
print("Input=",input_text)

Tokenize input text

input_ids = tokenizer(input_text, return_tensors="pt")

Perform translation

```
translated_output = hindi_tran.generate(**input_ids)
```

Decode the translated output

```
translated_text = tokenizer.decode(translated_output[0], skip_special_tokens=True)
```

Print the translated output

```
print("Translated Output (Hindi):")
print(translated_text)
```

Multiple models tried — You are done!