

Transformer Neural Networks

transformer is a type of neural network, and neural networks only have numbers for input values

1. Concept of Word Embeddings

Word embeddings are a form of representation that allows words with similar meaning to have a similar representation. They are essentially vectors of real numbers where each word in the language is mapped to a vector in a predefined vector space. Each vector captures the semantic properties of words that are semantically similar in a pre-defined vector space. Each vector captures the semantic properties of words; words that are semantically similar are placed closer together in this space.

2. Tokenization

Before a text can be processed by a transformer, it needs to be tokenized. Before text can be fed into a transformer it must be converted from raw strings of characters into tokens. This process is known as tokenization.

- word level tokenization

- subword tokenization (very commonly used)

- character (text) tokenization

Once a text is tokenized, each token is converted into a numerical form known as a token ID. The mapping from tokens to token IDs is defined by the vocabulary that the model was trained on. The vocabulary contains a unique ID for each token.

- A vocabulary is a list of all unique tokens that the model will recognize. Each token in this list is assigned a unique index or ID. The vocabulary is typically created during the training of the model on large corpora of text data, where the tokenization method is applied to the text, and frequency statistics are often used to decide which tokens are included in the vocabulary.

- Vocabulary size: the size of the vocabulary is a critical parameter. A large vocabulary can capture more information but requires more computational resources; a smaller vocabulary is more computationally efficient but may lose information if many words are mapped to an "unknown" token.

Mapping tokens to token IDs

- With a vocabulary established, each token in a text input is replaced by its corresponding ID based on the vocabulary list. This process is straight forward: the tokenizer scans through the input text, tokenizes it, and then substitutes each token with its index in the vocabulary. If a token does not exist in the vocabulary, it is replaced by the ID for a special token.

3. Token IDs to high dimensional vectors (embedding layer)

- Once each piece of text has been converted to token IDs, these IDs are then input into the embedding layer of the transformer model. This embedding layer contains a look-up table where each unique ID corresponds to a vector in high dimensional space.

- These vectors are not merely arbitrary; they are learned and contain semantic meaning. This transformation allows the model to process text in a meaningful way.

- If the transformer is trained from scratch, these vectors are initially random, but are adjusted during training to capture useful properties and relationships.

4. Positional Encoding

- Once each token is represented by its corresponding vector from the embedding layer, positional encodings are added to these vectors. Positional encodings provide the model with information about the order or position of tokens within the sequence. This is crucial because the self-attention mechanism in transformers does not process sequential data in order (unlike RNNs).

- we add a set of numbers that correspond to word order to the embedding values for each word.

- the transformer model uses sinusoidal functions to compute positional encodings, which are then added to the embeddings of the tokens. Specifically, for each dimension of the positional encoding vector:

- For even indices $2i$:

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

The dot product of a vector can be used to define how well two tokens align with each other.

- For odd indices $2i+1$:

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

$$\begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} \cdot \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} = v_1 \cdot w_1 + v_2 \cdot w_2 + \dots + v_n \cdot w_n = v \cdot w$$

If the value is positive, the vectors point in similar directions.

If the value is 0, the vectors are perpendicular.

If the dot product is negative, the vectors point in different directions.

What each variable means:

- pos is the position of the token (vector w semantic meaning) in the sequence.

- A network can only process a fixed number of vectors at a time known as its context size.

- i is the index within the positional encoding vector.

- In GPT-3, the context size was 5,120 (number of tokens being processed at once).

- d_{model} is the dimension of the embedding.

- In GPT-3, the context size was 5,120 (number of tokens being processed at once).

By adding positional encodings to the embedding of tokens, the transformer model can understand not only what each token represents (through its embedding) but also where it fits in the overall sequence (through its positional encoding). This is crucial for effectively handling tasks that depend on the sequence order, such as translating sentences or answering questions based on a given text.

5. Self-attention: what you need to add to your generic embeddings so it actually represents the true proper meaning of the word

- What is self-attention (high-level overview)?

- Self-attention sometimes referred to as intra-attention, is a mechanism that allows each token in the input sentence to interact w every other token to determine its influence on the sequence. This process helps the model to dynamically focus on different parts of the sequence as needed without being restricted by their position in the sequence.

- How self-attention is connected to the previous steps?

- The input to the self-attention layer consists of the output from the embedding and positional encoding steps. Each token representation now contains information about the tokens meaning (from embeddings) and its position in the sequence (from positional encodings).

- Matrix Representation: In practice, self-attention is computed using matrices. The input matrix X represents all embedded tokens in the sequence, adjusted for their positions. For each token, three different vectors are derived through linear transformations: Query (Q), Key (K), and Value (V). These transformations are performed using trainable weights, typically different for each attention head. In multi-head

attention setups.

- In the vector X

- each row in X corresponds to the embedded vector of a token. The first row of X contains the embedding (plus positional encoding) of the first token; the second row contains that of the second token, and so on.

Detailed computation steps

- Each token's embedded vector is transformed into 3 different vectors: Query (Q), Key (K), and Value (V). These values are used to determine the attention scores.

$$Q = XW^Q$$

$$K = XW^K$$

$$V = XW^V$$

Here, W^Q , W^K , and W^V are parameter matrices that are learned during training.

Wing down into 3 different vectors

- In the self-attention mechanism of a transformer, each input vector is transformed into the 3 vectors above; below is what they signify:

1. Query (Q): A query is a representation of a token used to score how well it matches every other token. Essentially when considering a specific token, its query vector is used to seek out which tokens (keys) are most relevant to it.
2. Key (K): A key is associated with a token to be matched against queries. When a query is compared to this key, the resulting score determines the impact of the corresponding token's value on the output.
3. Values (V): A value is a representation of a token that is used to compute the final output. Once tokens are scored based on their query-key matches, the values are weighted by these scores and summed up to produce the output for the next layer.

Attention scores determine how much focus to place on other parts of the input sequence when processing a specific part of that sequence.

$$\text{Attention Score} = \frac{\vec{Q} \cdot \vec{K}^T}{\sqrt{d_K}}$$

The scores are scaled down by the square root of the dimension of the key vectors (d_K) to stabilize the gradients during training.

Forward Pass

- For each input token, the model calculates the query, key and value vectors using the current values of W^Q , W^K , and W^V .
- Self-attention scores are computed based on the dot product of query and keys, scaled by the square root of the dimension of the key vector.
- These scores are normalized using a softmax function to create a distribution of weights.

Example

• a flying blue creature named the verdant forest

$$\begin{array}{ccccccccc} \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ E_1 & E_2 & E_3 & E_4 & E_5 & E_6 & E_7 & E_8 \end{array} \quad E_i \rightarrow \text{word embedding + positional encoding for word } i$$

$$\begin{array}{ccccccccc} \rightarrow 1 & \rightarrow 1 \\ E_1 & E_2 & E_3 & E_4 & E_5 & E_6 & E_7 & E_8 \end{array}$$

the goal is to create a new set of embeddings where each word considers the words around it to determine its meaning (Query Vectors)

To compute this take a weight vector W_Q and multiplying it by the embedding vectors

the entries of this matrix are parameters of the model (meaning true behavior is learned from data).

$$W_Q \cdot E_i = Q_i \text{ or query vector}$$

do this computation until you get a query vector for every token embedding

$$\begin{matrix} \vec{E}_i \\ 512 \end{matrix} \times \begin{matrix} W_Q \\ 512 \end{matrix} = \begin{matrix} \vec{Q}_i \\ 64 \end{matrix}$$

• a flying blue creature named the verdant forest

$$\begin{array}{ccccccccc} \downarrow & \downarrow \\ E_1 & E_2 & E_3 & E_4 & E_5 & E_6 & E_7 & E_8 \end{array} \quad \xrightarrow{\text{conceptually this is checking if there are any words that change the words meaning}}$$
$$\begin{array}{ccccccccc} \downarrow & \downarrow \\ W_1 & W_2 & W_3 & W_4 & W_5 & W_6 & W_7 & W_8 \end{array}$$
$$\begin{array}{ccccccccc} \downarrow & \downarrow \\ Q_1 & Q_2 & Q_3 & Q_4 & Q_5 & Q_6 & Q_7 & Q_8 \end{array}$$

• $W_K = \text{key matrix}$

$$\begin{array}{l} \rightarrow E_1 \rightarrow K_1, K_1 \cdot Q_1, K_1 \cdot Q_2, K_1 \cdot Q_3, K_1 \cdot Q_4, K_1 \cdot Q_5, K_1 \cdot Q_6, K_1 \cdot Q_7, K_1 \cdot Q_8 \\ \text{fluffy} \rightarrow E_2 \rightarrow K_2, K_2 \cdot Q_1, K_2 \cdot Q_2, K_2 \cdot Q_3, K_2 \cdot Q_4, K_2 \cdot Q_5, K_2 \cdot Q_6, K_2 \cdot Q_7, K_2 \cdot Q_8 \\ \text{blue} \rightarrow E_3 \rightarrow K_3, K_3 \cdot Q_1, K_3 \cdot Q_2, K_3 \cdot Q_3, K_3 \cdot Q_4, K_3 \cdot Q_5, K_3 \cdot Q_6, K_3 \cdot Q_7, K_3 \cdot Q_8 \\ \text{creature} \rightarrow E_4 \rightarrow K_4, K_4 \cdot Q_1, K_4 \cdot Q_2, K_4 \cdot Q_3, K_4 \cdot Q_4, K_4 \cdot Q_5, K_4 \cdot Q_6, K_4 \cdot Q_7, K_4 \cdot Q_8 \\ \text{named} \rightarrow E_5 \rightarrow K_5, K_5 \cdot Q_1, K_5 \cdot Q_2, K_5 \cdot Q_3, K_5 \cdot Q_4, K_5 \cdot Q_5, K_5 \cdot Q_6, K_5 \cdot Q_7, K_5 \cdot Q_8 \end{array}$$

= dot product values highlighted with green are high values

All these dot products are being computed parallel

• conceptually this is answering

* If the dot product of a key and query vector is really high then the specific key embeddings attend to the query embeddings

to the query whether or

not a word is changing

its meaning

* As of right now the grid of values ranges from -oo to oo giving us a score for how relevant each word is to updating the meaning of every other word

* Next we will take a weighted sum along each column, weighted by relevance (so we want each column to add up to one, as it were a probability distribution)

a fluffy blue creature roamed the verdant forest

$E_1 \downarrow E_2 \downarrow E_3 \downarrow E_4 \downarrow E_5 \downarrow E_6 \downarrow E_7 \downarrow E_8 \downarrow$

$W_1 \downarrow W_2 \downarrow W_3 \downarrow W_4 \downarrow W_5 \downarrow W_6 \downarrow W_7 \downarrow W_8 \downarrow$

$Q_1 \downarrow Q_2 \downarrow Q_3 \downarrow Q_4 \downarrow Q_5 \downarrow Q_6 \downarrow Q_7 \downarrow Q_8 \downarrow$

$\cdot W_K = \text{key matrix}$

$a \rightarrow E_1 \rightarrow K_1$

$\text{fluffy} \rightarrow E_2 \rightarrow K_2$

$\text{blue} \rightarrow E_3 \rightarrow K_3$

$\text{creature} \rightarrow E_4 \rightarrow K_4$

$\text{roamed} \rightarrow E_5 \rightarrow K_5$

\vdots

$\rightarrow E_6 \rightarrow K_6$

$\rightarrow E_7 \rightarrow K_7$

$\rightarrow E_8 \rightarrow K_8$

normalized columns
called an attention pattern

$K_1 \cdot Q_1$

$K_2 \cdot Q_1$

$K_3 \cdot Q_1$

$K_4 \cdot Q_1$

$K_5 \cdot Q_1$

$K_6 \cdot Q_1$

$K_7 \cdot Q_1$

$K_8 \cdot Q_1$

apply a softmax function to each column

* to normalize each column, apply a softmax function to each column so it adds up to one

* at this point its safe to think about each column as giving weights according to how relevant the word on the left is to the corresponding value at the top

softmax function - how it works

* what is the goal of a softmax function?

- softmax is a way to change a valid list of numbers into a valid probability distribution such that the largest values end up having a higher probability (closer to 1) while, the smallest values have a smaller probability (closer to 0)

state your output vector (doesn't add up to 1) ex.

$K_1 \cdot Q_1 = x_1$, and raise e to the power of each of the values in the vector
 $K_2 \cdot Q_1$, which makes all values positive:

$K_3 \cdot Q_1$,
 $K_4 \cdot Q_1$,
 $K_5 \cdot Q_1$,
 $K_6 \cdot Q_1$,
 $K_7 \cdot Q_1$,
 $K_8 \cdot Q_1$

then take the sum of all e^{x_i} values and divide each e^{x_i} value by the sum

$e^{x_1} / \sum_{n=0}^{N-1} e^{x_n}$

which normalizes the vector which adds up to 1

softmax with Temperature

* the temperature T multiplies the softmax function by dividing the logits by the temperature value before applying the softmax:

$$a(z; T) = \frac{e^{z/T}}{\sum_j e^{z/T}}$$

* the temperature plays a role in the "sharpness" of the probability distribution:

* high temperature ($T > 1$): the softmax

output becomes smoother and more uniform.

higher temperatures make the model

output probabilities closer to each

other, which can be useful when you

want the model to be less confident

and explore more diverse options

* In the paper "Attention is all you need" all of the above details are represented in the following equation:

$$\text{Attention}(Q, K, V) = \text{softmax}(QK^T)V / \sqrt{d_K}$$

* Q represents a full matrix of query vector embeddings

* K represents a full matrix of key vector embeddings

- QK^T \rightarrow represents the dot product between the key and query matrices

- $\sqrt{d_K}$ \rightarrow divides QK^T by $\sqrt{d_K}$ for numerical stability

* d_K dimension in the key vector space

- then the softmax that's wrapped around QK^T normalizes the dot product between the key and query matrices

* During the training process when you run this model on a given test example and tokens are being tested:

- it makes sense to train the model with a sentence from the ground up

example: if you have the sentence - "a fluffy blue creature roamed the verdant forest"

the fluffy \rightarrow [99]

the fluffy blue \rightarrow [99]

the fluffy blue creature \rightarrow [99]

the fluffy blue creature roamed \rightarrow [99]

the fluffy blue creature roamed the \rightarrow [99]

the fluffy blue creature roamed the verdant \rightarrow [99]

the fluffy blue creature roamed the verdant forest \rightarrow [99]

* training process becomes a lot more efficient if you simultaneously have the model predict every possible next token following each initial subsequent of tokens in this passage

* while words effectively act as a single training example would now act as many training examples

* the bigger the size the longer the dot product

MASKING

* GPT models are autoregressive language models. This means that the prediction of a word at a specific position should only depend on the words before it, and not on any future words.

* masking is used within the self-attention layer to prevent positions from attending to subsequent positions. This means that for a given word, the attention mechanism can only consider previous words in the sequence

* How masking is created: A mask is created where the mask at position i, j in the matrix is set to 0 if $i > j$ and -∞ if $i < j$. The mask is applied before the softmax step in the attention mechanism

* During the softmax calculation on the attention scores, the -∞ will effectively become zero after applying the softmax function, which means that those positions will not contribute to the weighted sum in the attention output

* this is essential because w/o it the model might learn to rely on future inputs that will not be available during text generation, leading to poor performance and non-sensical outputs

a fluffy blue creature roamed the verdant forest

$E_1 \downarrow E_2 \downarrow E_3 \downarrow E_4 \downarrow E_5 \downarrow E_6 \downarrow E_7 \downarrow E_8 \downarrow$

$W_1 \downarrow W_2 \downarrow W_3 \downarrow W_4 \downarrow W_5 \downarrow W_6 \downarrow W_7 \downarrow W_8 \downarrow$

$Q_1 \downarrow Q_2 \downarrow Q_3 \downarrow Q_4 \downarrow Q_5 \downarrow Q_6 \downarrow Q_7 \downarrow Q_8 \downarrow$

$\cdot W_K = \text{key matrix}$

$a \rightarrow E_1 \rightarrow K_1$

$\text{fluffy} \rightarrow E_2 \rightarrow K_2$

$\text{blue} \rightarrow E_3 \rightarrow K_3$

$\text{creature} \rightarrow E_4 \rightarrow K_4$

$\text{roamed} \rightarrow E_5 \rightarrow K_5$

\vdots

$\rightarrow E_6 \rightarrow K_6$

$\rightarrow E_7 \rightarrow K_7$

$\rightarrow E_8 \rightarrow K_8$

the size of the attention grid is equal to the square of the context size

this is why context size can be a huge bottleneck for large language models
and scaling it up is a non-trivial task

$\cdot W_K = \text{key matrix}$

$a \rightarrow E_1 \rightarrow K_1$

$\text{fluffy} \rightarrow E_2 \rightarrow K_2$

$\text{blue} \rightarrow E_3 \rightarrow K_3$

$\text{creature} \rightarrow E_4 \rightarrow K_4$

$\text{roamed} \rightarrow E_5 \rightarrow K_5$

\vdots

$\rightarrow E_6 \rightarrow K_6$

$\rightarrow E_7 \rightarrow K_7$

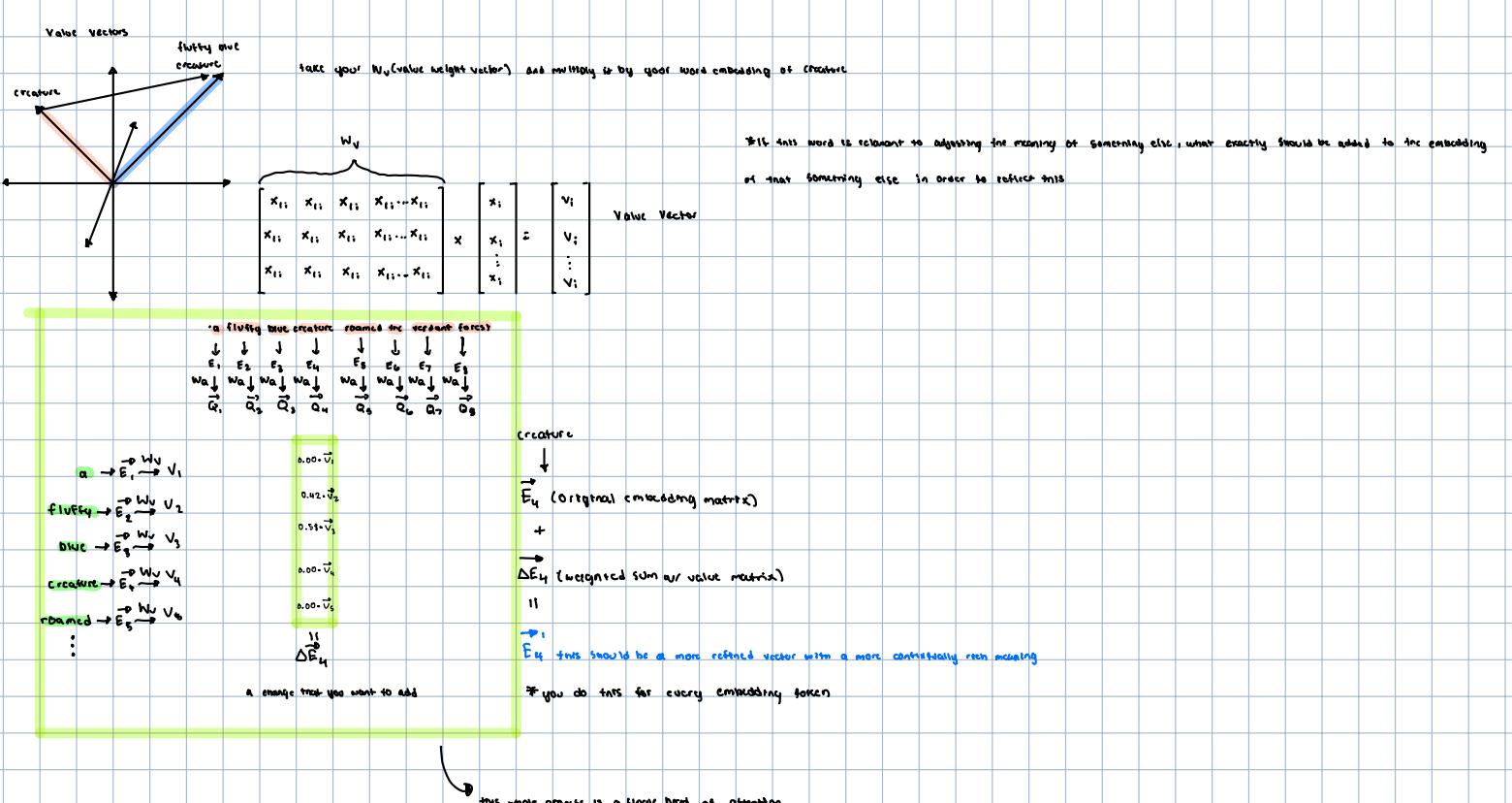
$\rightarrow E_8 \rightarrow K_8$

* if you never want later words to influence earlier words, otherwise they could give away the answer for what comes next

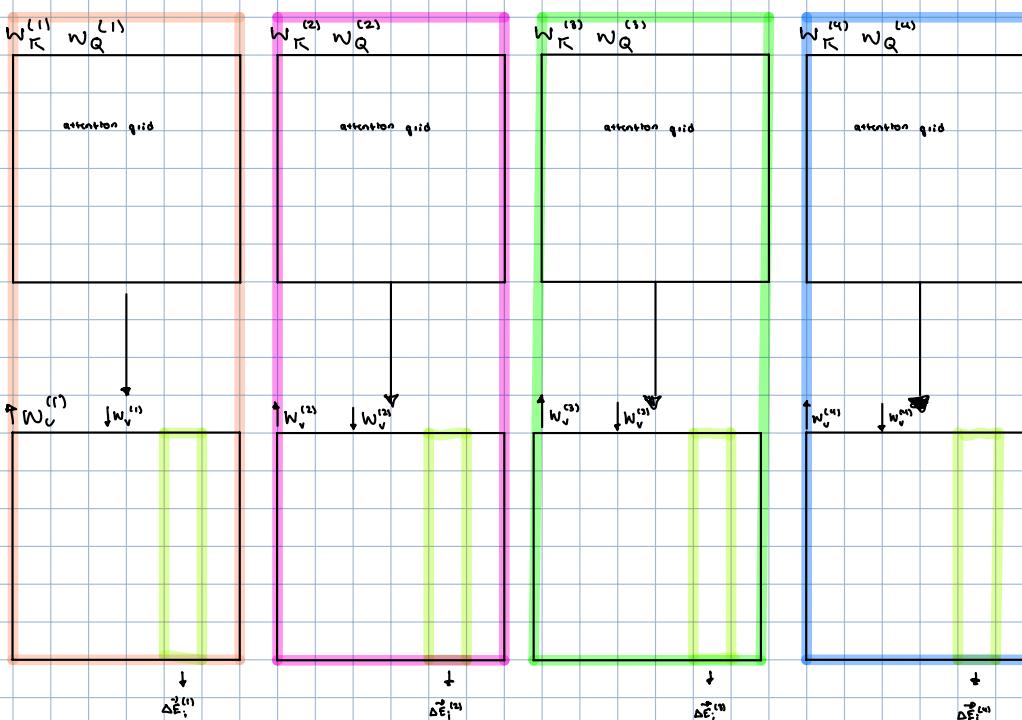
* that means the spots in \square represent later tokens influencing earlier ones, should be forced to be 0

* its not possible to force the red boxes to be equal to 0 because then the normalized columns will not add up to 1 anymore

* the most common way to fix this issue is before applying the softmax function, you set all the entries in the \square to be negative infinity -∞, now after applying the softmax function, all the values in the \square that are set to -∞ will get turned to 0, while keeping the column normalized



this whole process is a single head of attention



every head proposes a change to the original vector embedding that will be added to the original embedding

$$\vec{E}_i = \vec{E}_i^{(1)} + \vec{E}_i^{(2)} + \vec{E}_i^{(3)} + \vec{E}_i^{(4)} + \dots$$

(context many attention heads)

If each color is an attention head, and by running many heads in parallel you can learn many distinct ways, your giving the model the capacity to learn many distinct ways that context changes meaning

If this diagram shows the architecture of how the model is being trained

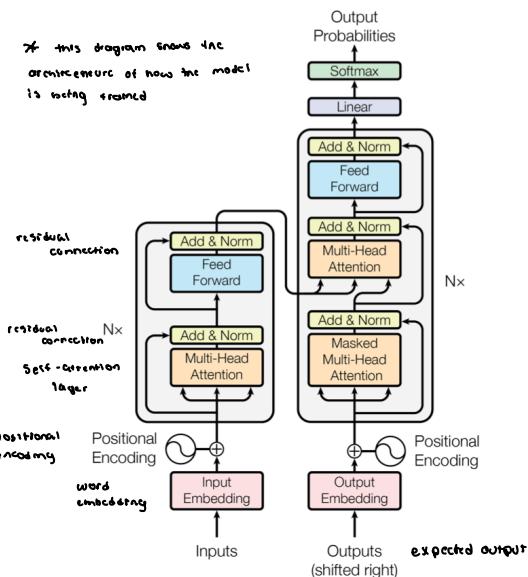
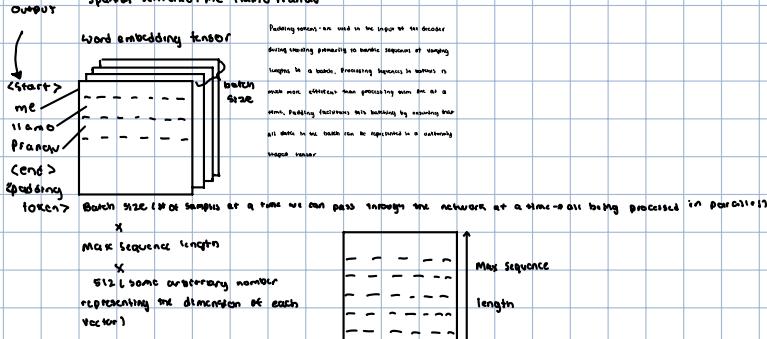


Figure 1: The Transformer - model architecture.

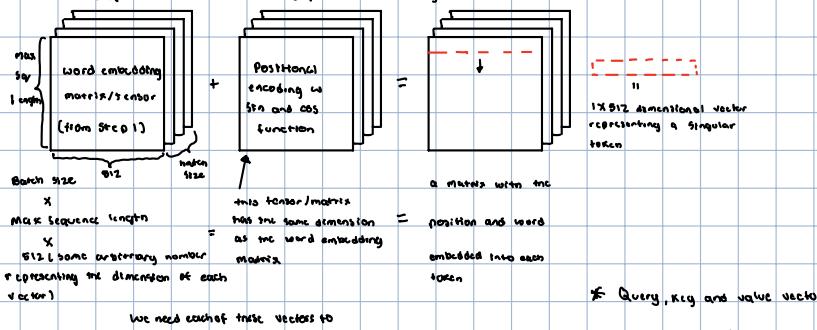
lets say we are translating a sentence from english to spanish

- *English sentence: My name is Pronav \rightarrow after this sentence is passed into the encoder, each word will be turned into a numerical vector that has the contextual meaning of the word, the positional encodings of the words, and the relationships with other words encoded into each token vector.
- *Spanish sentence: Me llamo Pronav

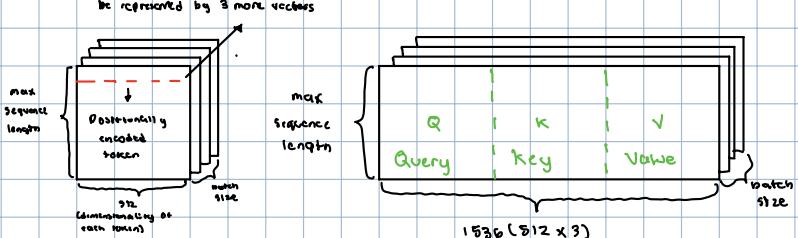
Expected Output



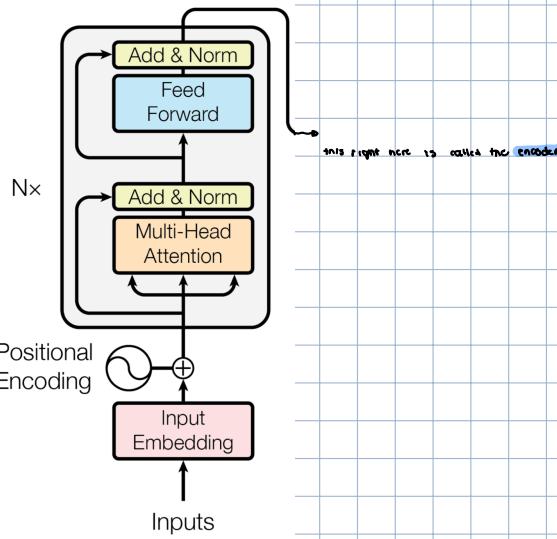
Next Step in Decoder Process: add positional encoding



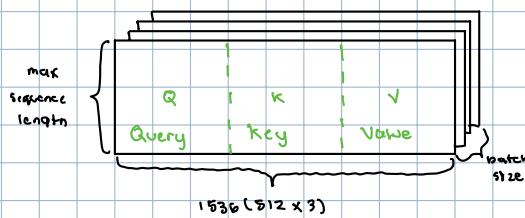
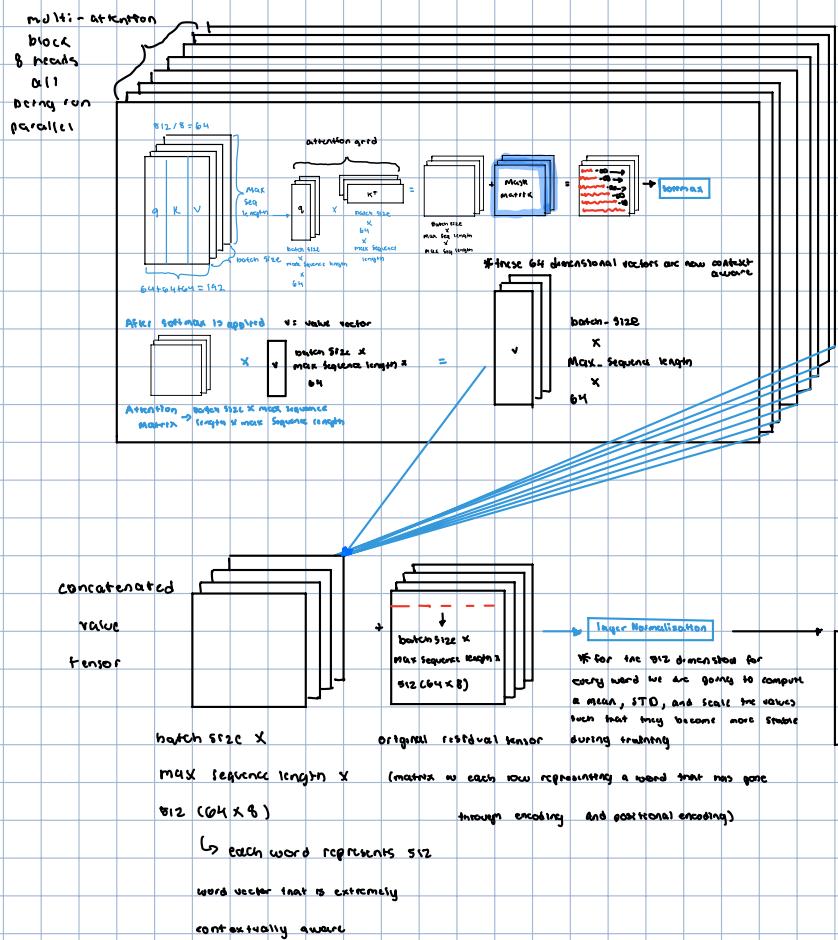
* Query, Key and Value vector can be intuitively interpreted as the same as before



The 4 features labeled next to the encoder allow the transformer to encode words into numbers, encode the positions of the words, encode the relationships among the words, and relatively easily and quickly train in parallel.



this right here is called the encoder



* you want to break each query, key, and value matrix into 8 different vectors (so that means you should divide on the vertical dimension)



~ this is equal to the query vector of the first head

Multimodal Cross-Attention

