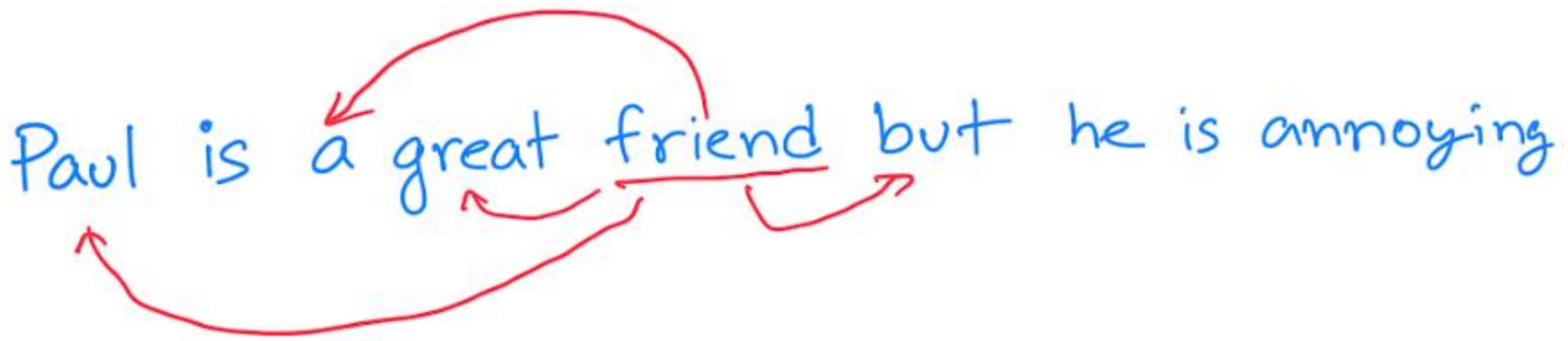
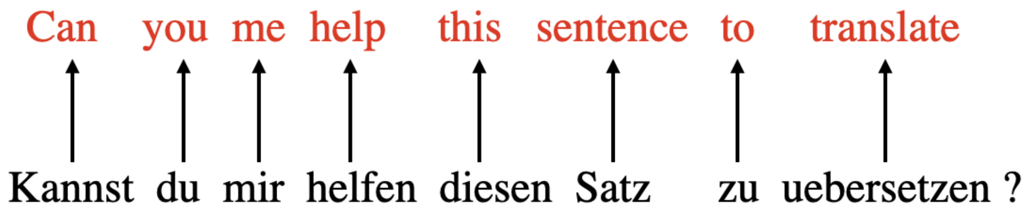
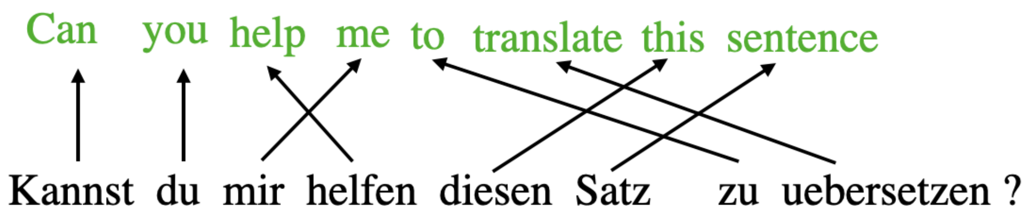
CSE 495 (Natural Language Processing) Lecture 8

Use Self-Attention to Neural Networks: Attention is all you need

Multiple types of context

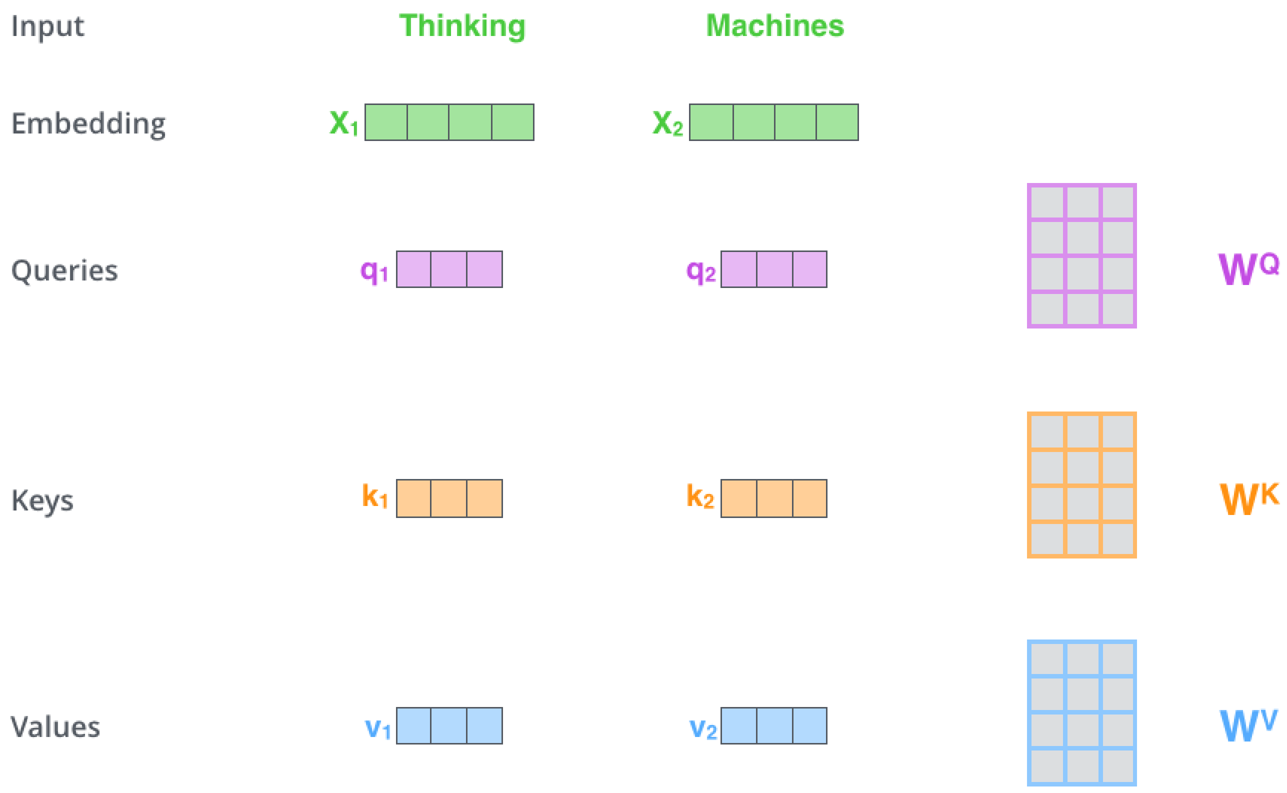



Can you me help this sentence to translate
Kannst du mir helfen diesen Satz zu uebersetzen ?


Can you help me to translate this sentence
Kannst du mir helfen diesen Satz zu uebersetzen ?

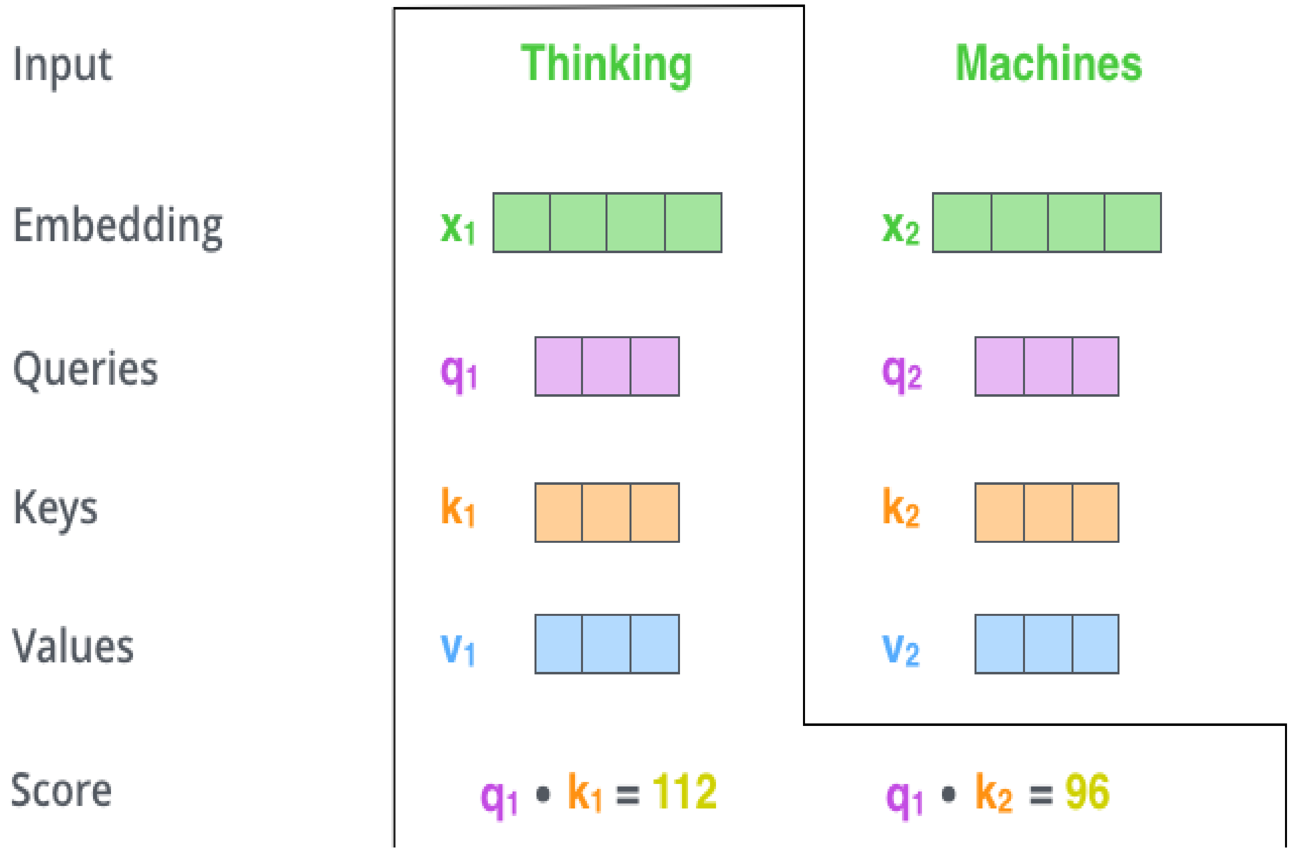
Self Attention: Projection

- The first step in self-attention is to create vectors from the embedding of each word.
- From each word, create a Query vector, a Key vector, and a Value vector.
  - These vectors are created by multiplying the embedding by three matrices that we trained previously.
- These new vectors are smaller in dimension than the embedding vector.
  - Their dimensionality is 64, while the embedding and encoder input/output vectors have dimensionality of 512.



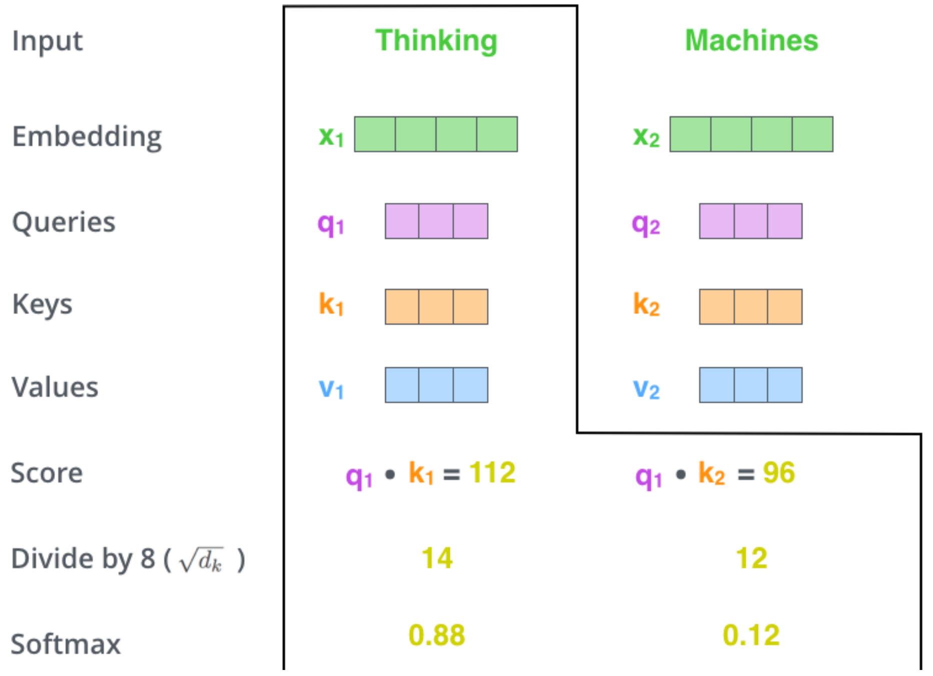
Self Attention: The (concept) weight

- The second step in self-attention is to calculate a score/weight.
- Say we're calculating the self-attention for the first word in this example, "Thinking".
    - We need to score each word of the input sentence against this word.
    - The score determines how much focus to place on other parts of the input sentence as we encode a word at a certain position.
- The score is calculated by taking the dot product of the query vector with the key vector of the respective word.
- So if we're processing the self-attention for the word in position #1, the first score would be the dot product of q1 and k1. The second score would be the dot product of q1 and k2.



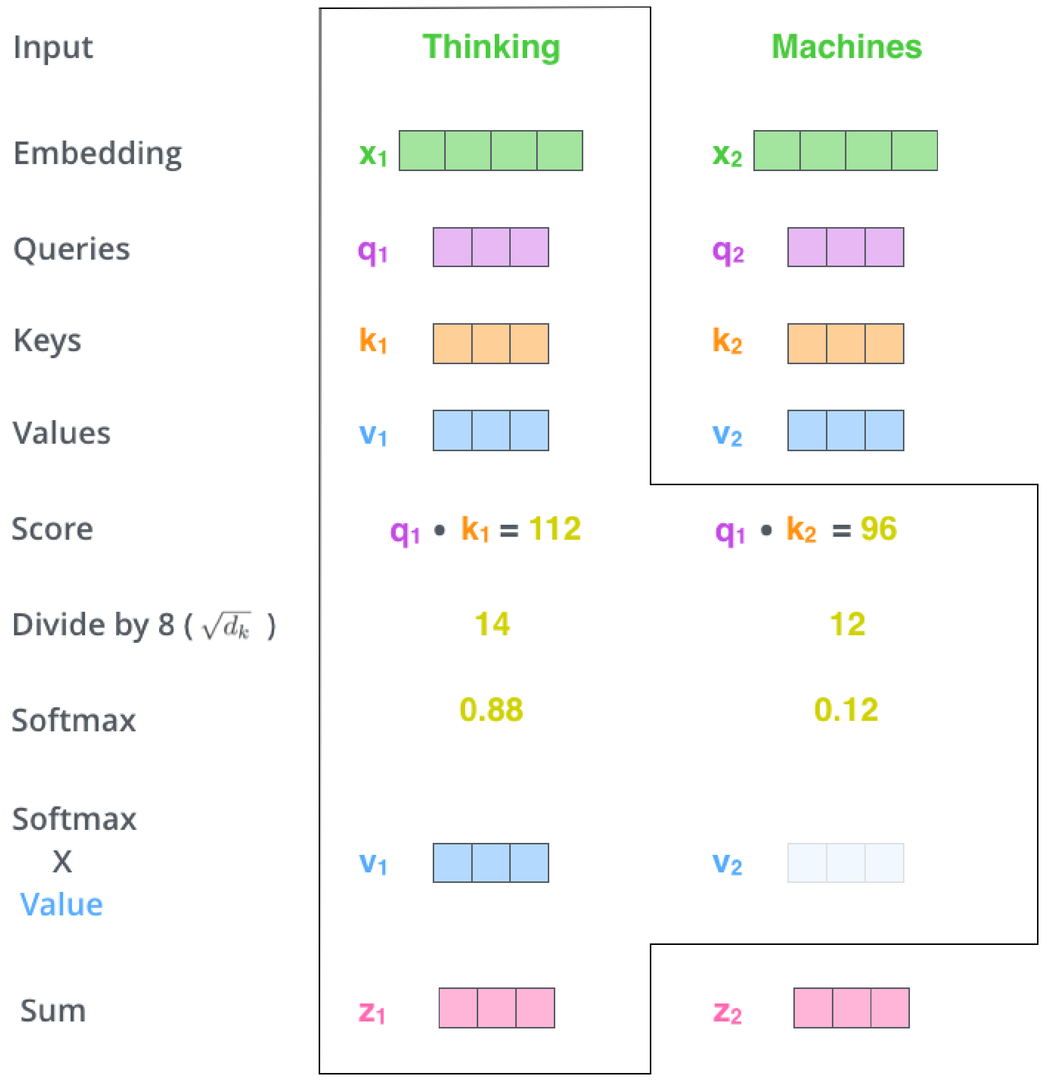
Self Attention: Normalizing concept weight

- The third and fourth steps are to divide the scores by the square root of the dimension of the key vectors used.
- This step ensures more stable gradients.
- Then pass the result through a softmax operation.
- Softmax normalizes the scores so they're all positive and add up to 1.



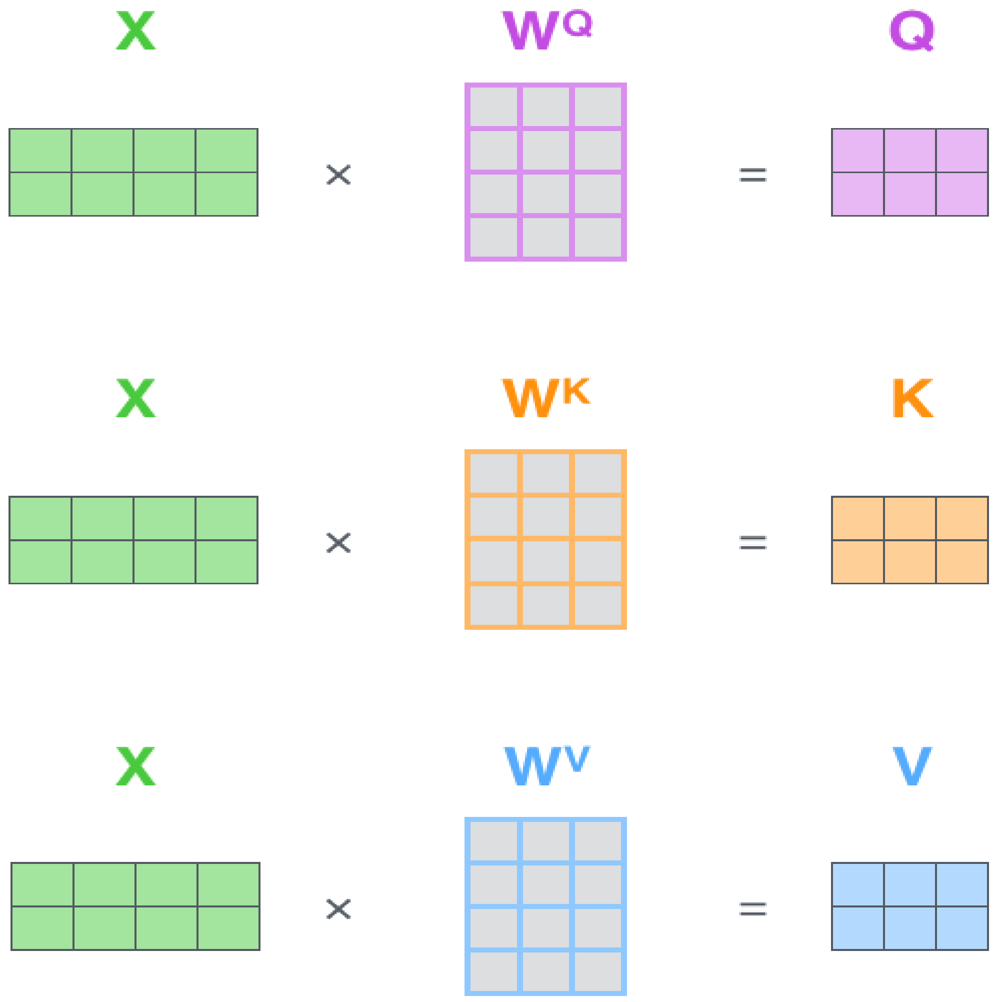
Self Attention: New Embedding

- The fifth step is to multiply each value vector by the softmax score (in preparation to sum them up).
- The intuition here is to keep intact the values of the word(s) we want to focus on, and drown-out irrelevant words (by multiplying them by tiny numbers like 0.001, for example).
- The sixth step is to sum up the weighted value vectors. This produces the output of the self-attention layer at this position (for the first word).

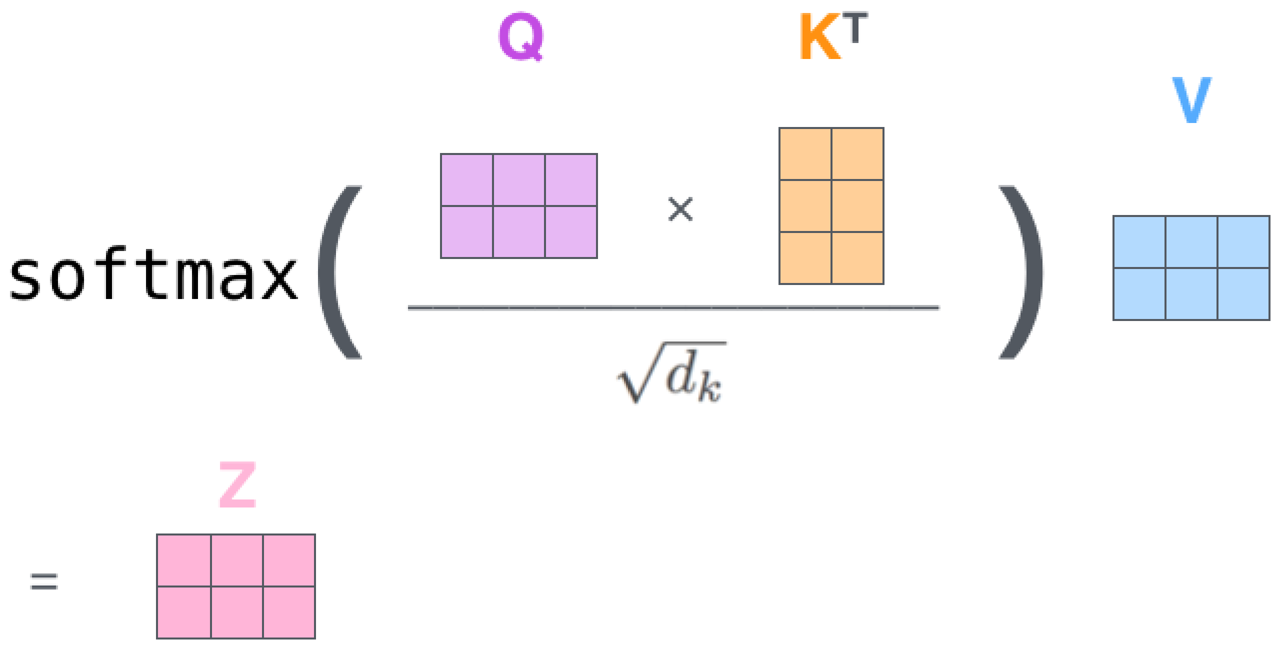


Getting Q,K,Vs

- The first step is to calculate the Query, Key, and Value matrices.
- We do that by packing our embeddings into a matrix X, and multiplying it by the weight matrices we've trained (W^Q, W^K, W^V).

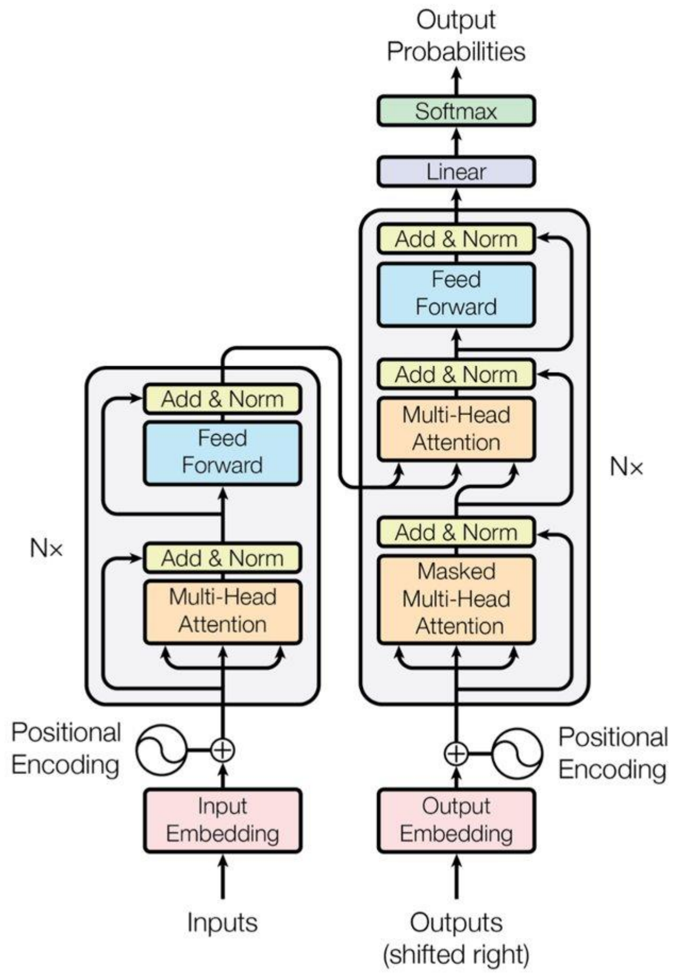


Everything as a Matrix Operation



Attention is all you need

- It is an encoder-decoder architecture
- The input goes through encoder
  - Then through decoder
- On the encoder side
  - Each block has two sublayers
    - Multihead attention then feedforward
- On the decoder side
  - Each block has three sublayers
    - Two multihead attention followed by a feedforward layer
  - However, the first MHA layer is similar to the decoder MHA but the second one is different
- The attention mechanism is a “word to word” operation
  - Token to token operation, but we can think of it at the word level to simplify the explanations
- The attention mechanism will find how each word relates to all other words in a sequence, including the word being analyzed



Self Attention

- Attention is a mechanism that enables neural networks to assign varying levels of importance—or attention—to different elements in a sequence.
    - Instead of assigning a fixed embedding to each token, the model computes a weighted average of embeddings based on context.
- Given a sequence of token embeddings $x_1, \ldots, x_n$, self-attention generates a sequence of new embeddings $x_1', \ldots, x_n'$ where each x'_i is a weighted sum of all x_j:

$$x'_i = \sum_{j=1}^{n} w_{ji} x_j$$

- The coefficients w_{ji} are called attention weights and are normalized so that $\sum_j w_{ji} = 1$.

Self–dot–product attention

- Several methods exist to implement self-attention layers, but scaled dot-product attention is the most widely used approach. It involves four key steps:
- ① Project embeddings into Query, Key, and Value vectors
    - Each token embedding is projected into three distinct representations: query (Q), key (K), and value (V).
- ② Compute Attention Scores
    - A similarity function calculates how closely related each query is to each key.
        - Scaled dot-product attention uses the dot product as the similarity measure, efficiently computed through matrix multiplication.
        - Tokens with similar queries and keys produce high attention scores, while unrelated tokens yield lower scores.
        - The result is an $n \times n$ attention-score matrix for a sequence of n tokens.
- Compute attention scores: Use a similarity function to measure how much each query relates to each key.
    - For scaled dot-product attention, the similarity is computed using the dot product, efficiently implemented via matrix multiplication.
    - Tokens with similar queries and keys will have high dot products, indicating strong relevance.
    - Tokens with unrelated queries and keys will have low or near-zero dot products.
    - The output is an n × n matrix of attention scores for an input of length n

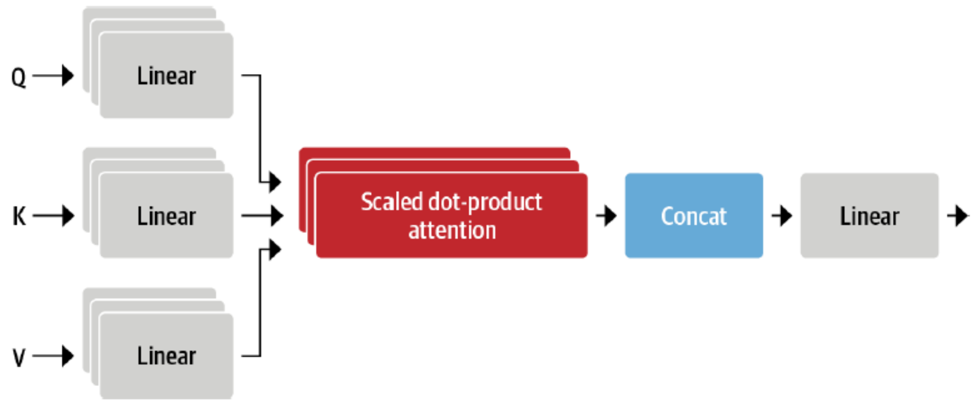
Self-dot-product attention

- ③ Compute Attention Weights
  - Dot products can produce large numbers, potentially causing unstable training.
  - To address this, attention scores are divided by a scaling factor (typically sqrt(d_k)-d_k the dimension of key vectors) to stabilize variance.
  - Scores are then passed through a softmax function, normalizing them so each token's weights sum to 1.
  - This produces an $n \times n$ attention-weights matrix w_{ji}
- ④ Update Token Embeddings
  - Attention weights are used to calculate new embeddings for each token:
    - $x_i' = \sum w_{ji} v_j$ $(j=1 \ldots n)$
  - Thus, each updated embedding captures contextual relationships based on the learned attention weights.

Key Query Value

- Given an input sequence of length N, represented as a matrix X of shape (N,d), where d is the dimensionality of the input embeddings, we compute the query (Q), key (K), and value (V) vectors as follows:
- Query (Q):
  - Multiply X by a learned query weight matrix Wq of shape (d,h), where h is the output dimension (typically the dimensionality per attention head).
  - This results in a query matrix Q of shape (N,h)):
    - $Q = XWq$
- Key (K):
  - Multiply X by a learned key weight matrix Wk of shape (d,h)), resulting in a key matrix K of shape (N,h):
    - $K=XWk$
- Value (V):
  - Multiply X by a learned value weight matrix Wv Wv of shape (d,h) , resulting in a value matrix V of shape (N,h) :
    - $V = XWv$
- In each resulting matrix, every row corresponds to the respective query, key, or value vector of each token in the input sequence.

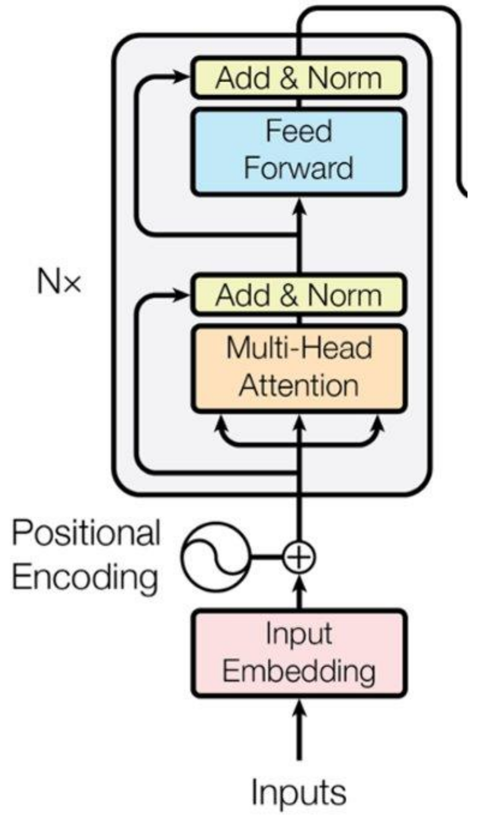
Multi-head Attention



- Transformers use multiple sets of linear projections, each forming an attention head.
- Why multiple attention heads?
  - Single-head attention limits the model to focus on just one type of relationship or similarity between tokens.
  - Multi-head attention allows the model to capture multiple relationships simultaneously, making the representation richer and more nuanced.
- For example:
  - One attention head might capture subject-verb interactions.
  - Another head could focus on finding related adjectives or descriptive words.
  - Yet another might identify long-range dependencies within sentences.
- These relationships aren't manually defined by humans—instead, the Transformer learns them entirely from data, improving its generalization and interpretative power.

The Encoder Stack

- A Transformer encoder typically consists of N=6 identical layers, each following the same structural pattern.
- Components of Each Layer:
  - Each encoder layer has two main sublayers:
    - Multi-head self-attention: allows tokens to attend to each other.
    - Position-wise feedforward neural network: applies transformations individually to each token's embedding.
- Residual Connections and Layer Normalization:
  - Residual (skip) connections surround each sublayer:
    - These connections add the original input (x) to the sublayer's output before applying normalization.
    - The output after residual connection and normalization is:
      - LayerNorm(x+Sublayer(x))
- Layer Differences Despite Identical Structure:
  - Although all layers share an identical structure, each encoder layer learns different representations:
    - Layers differ because they have separate learned parameters (weights).
    - Each subsequent layer further enriches embeddings by integrating additional context from other tokens.

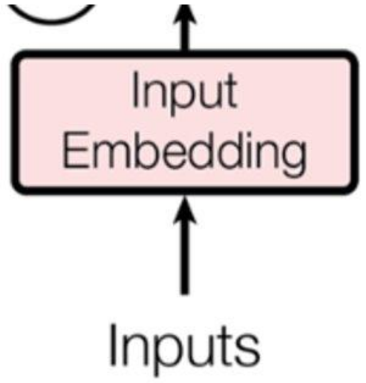


Building Contextual Representations

- Each of the multi-head attention layers (layers 1 through 6) performs the same underlying operations, but each layer learns distinct relationships.
- Why don't layers learn identical things?
  - Each layer progressively enriches token embeddings by adding increasingly nuanced contextual knowledge.
  - You can think of each layer as searching for different relationships between tokens—similar to how we discover various associations among letters and words when solving a crossword puzzle.
- Consistent dimensionality:
  - Every sublayer (attention and feedforward layers) produces outputs of constant dimension $d\ model$
  - Residual connections and embeddings also maintain this dimension, creating uniformity and stability.
- Choosing $d\ model$:
  - In the original Transformer architecture, $d\ model$=512, but it can be adjusted based on the task or resources.
  - The choice of $d\ model$is crucial because it significantly impacts model performance and capability.
- Scalability:
  - Transformers can scale by adding more layers, limited only by computational resources. More layers often lead to richer, more powerful representations.

Input Embedding

- The input embedding sublayer converts the input tokens to vectors of dimension d_{model} using learned embeddings from another model (transformer/word2vec/one-hot model)
- The embedding sublayer works on the tokens that the transformer will find embeddings
  - Tokenizer could be BPE, word piece, and sentence piece etc.
- A tokenizer provides an integer representation that will be used for the embedding process. For example:

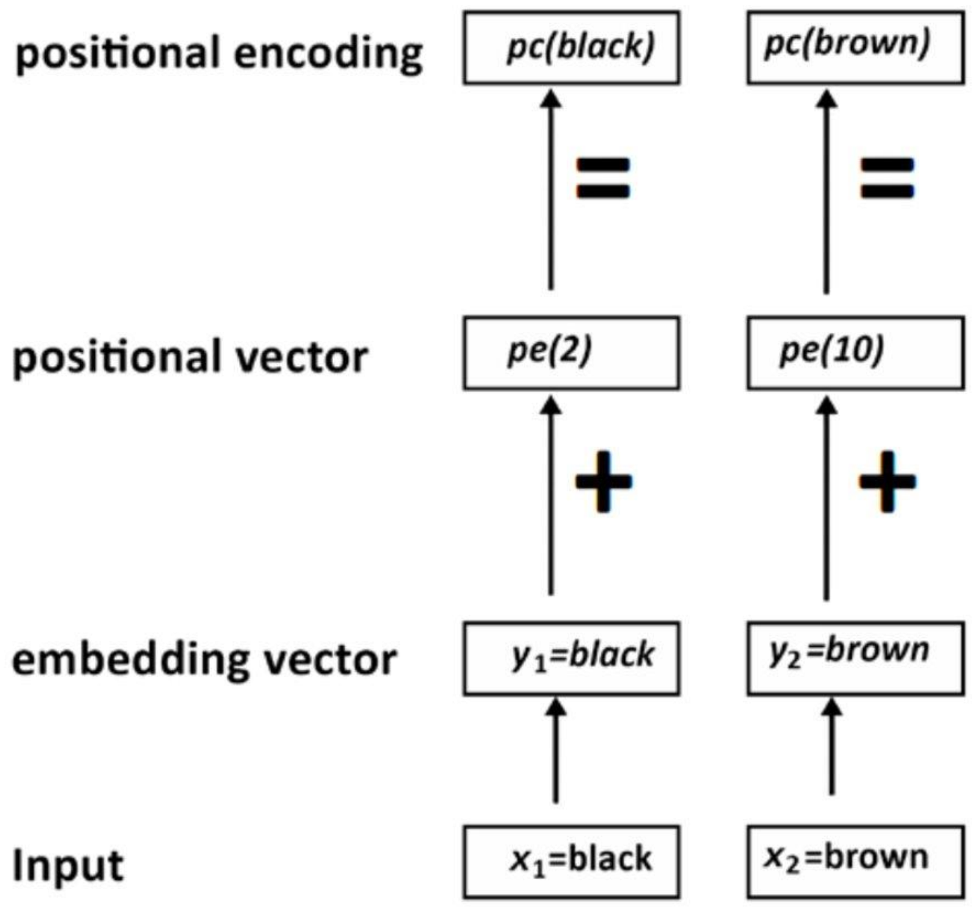


```
text = "The cat slept on the couch.It was too tired to get up."
tokenized text= [1996, 4937, 7771, 2006, 1996, 6411, 1012, 2009, 2001,
2205, 5458, 2000, 2131, 2039, 1012]
```

Positional Encoding

- The self-attention calculation method does not consider position of the tokens
    - Therefore it does not have any idea of the relative word/token orders
- Positional embeddings takes care of this shortcoming
    - The most basic embedding is simple: augment the token embeddings with a position-dependent pattern of values arranged in a vector
        - If the pattern is characteristic for each position, the attention heads and feed-forward layers in each stack can learn to incorporate positional information into their transformations
- It is possible to learn the pattern, especially when the pretraining dataset is sufficiently large
    - This works exactly the same way as the token embeddings, but using the position index instead of the token ID as input
- Absolute positional representations: Transformer models can use static patterns consisting of modulated sine and cosine signals to encode the positions of the tokens.
    - This works especially well if the dataset is relatively small.

Positional Encoding

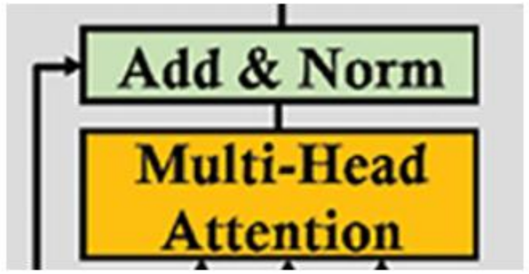


Relative Positional Encoding

- Although absolute positions are important, the neighboring tokens are most important
- Relative positional representations follow this intuition and encode the relative positions between tokens
- This cannot be done by just introducing a new relative embedding layer at the beginning since the relative embedding changes for each token depending on where from the sequence we are attending to it
- Instead, the attention mechanism itself is modified with additional terms that consider the relative position between tokens
- Models such as DeBERTa use such representations

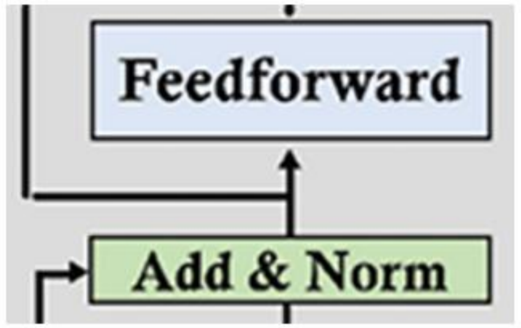
Layer Normalization

- All sublayers (attention and feedforward) of the Transformer is followed by Post-Layer Normalization
- The Post-LN contains an add function and a layer normalization process
- The add function processes the residual connections that come from the input of the sublayer and are normalized
    - The goal of the residual connections is to make sure critical information is not lost



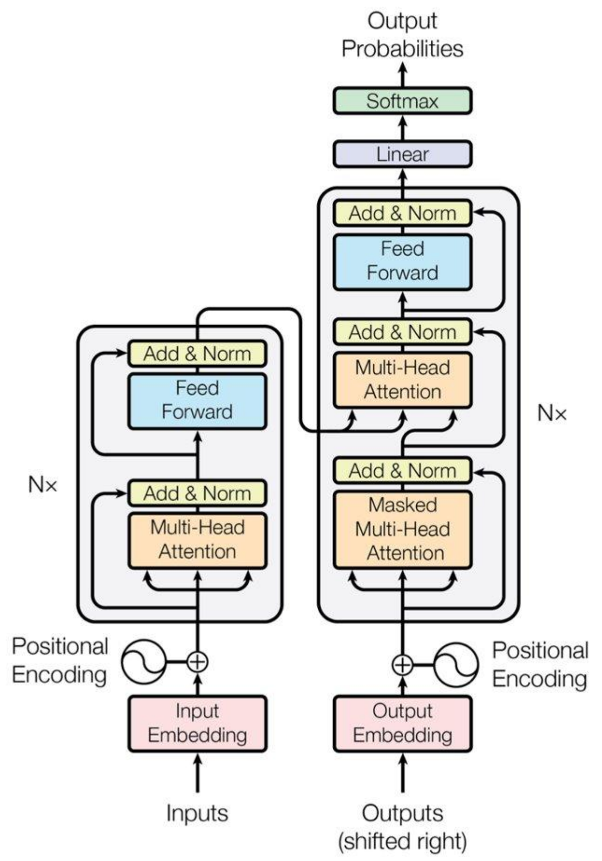
Position-wise Feedforward layer

- The FFNs in the encoder and decoder are fully connected
- The FFN is a position-wise network
  - Each position in the input sequence is processed independently but identically
  - In other words all tokens within a sublayer are transformed using the same set of weights
  - The term "position-wise" emphasizes that the linear layer operates on a 3D tensor, where the token dimension is the last one.
- The FFN is a two stage network with ReLU activation function
- The input and output of the FFN layers for original Transformer is dmodel = 512, but the inner layer is larger with dff =2048
- The output of the FFN goes to a LN module
- Then the normalized output is sent to the next layer of the encoder stack and the multi-head attention layer of the decoder stack



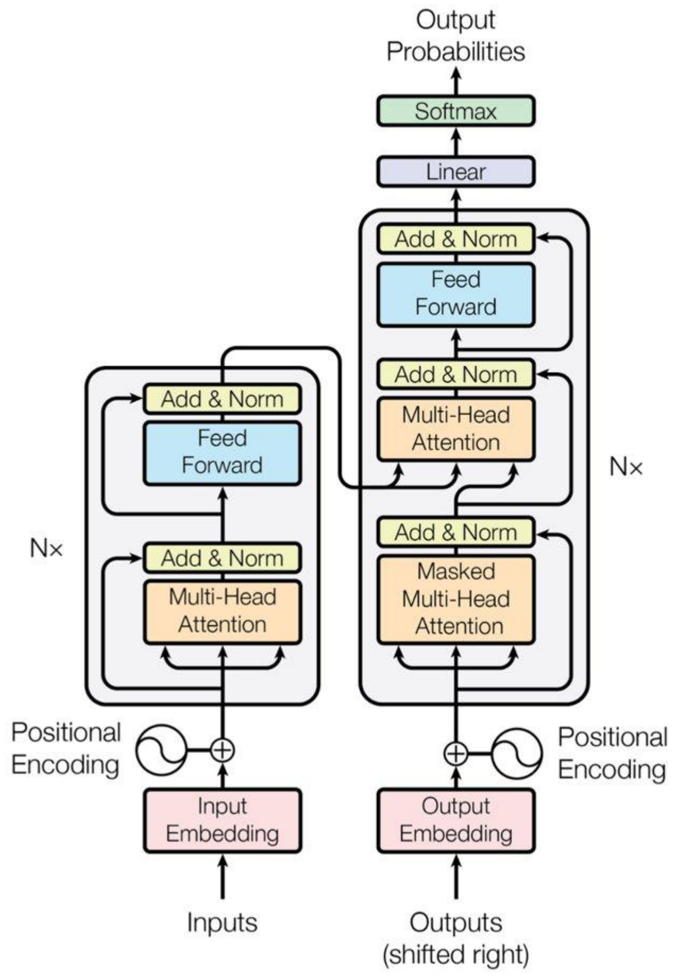
Decoder Layer

- The structure of the decoder layer remains the same similar to the encoder for all the N = 6 layers of the original Transformer model
- Each layer contains three sublayers:
  - A multi-headed masked attention mechanism
  - A multi-headed attention mechanism
  - A fully connected position-wise feedforward network.
- The decoder has a different sublayer (than encoder), which is the masked multi-head attention mechanism
- In this sublayer; inputs are partially masked so that the Transformer can predict the next token on its own without looking into the rest of the sequence
- Similar to the encoder, residual connection, Sublayer(x), surrounds each of the three main sublayers to ensure proper flow of gradients



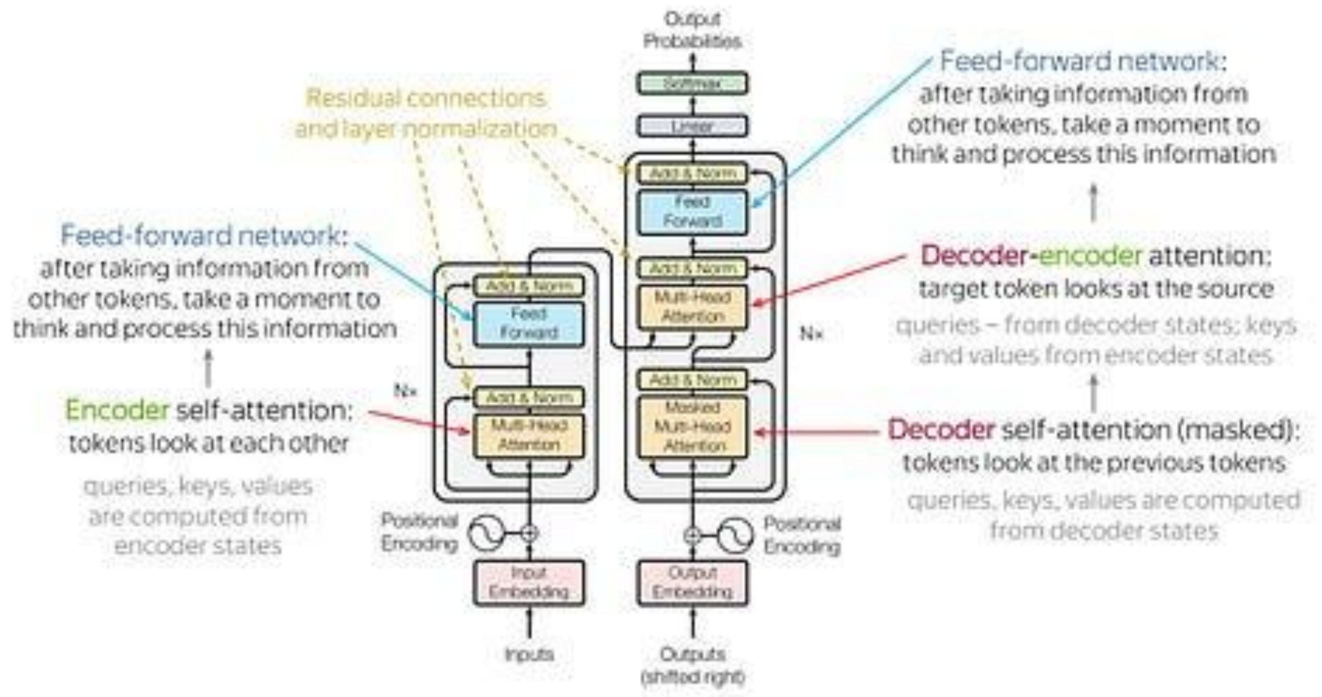
Decoder Layer

- The output embedding sublayer is only present at the bottom level of the stack, like for the encoder stack
- The output of every sublayer of the decoder stack has a constant dimension, d_{model}, like in the encoder stack, including the embedding layer and the output of the residual connections.
- We can see that the design is symmetrical: decoder decodes a sequence from the encoded information
- The structure of each sublayer and function of the decoder is similar to the encoder



Cross Attention (Co-attention)

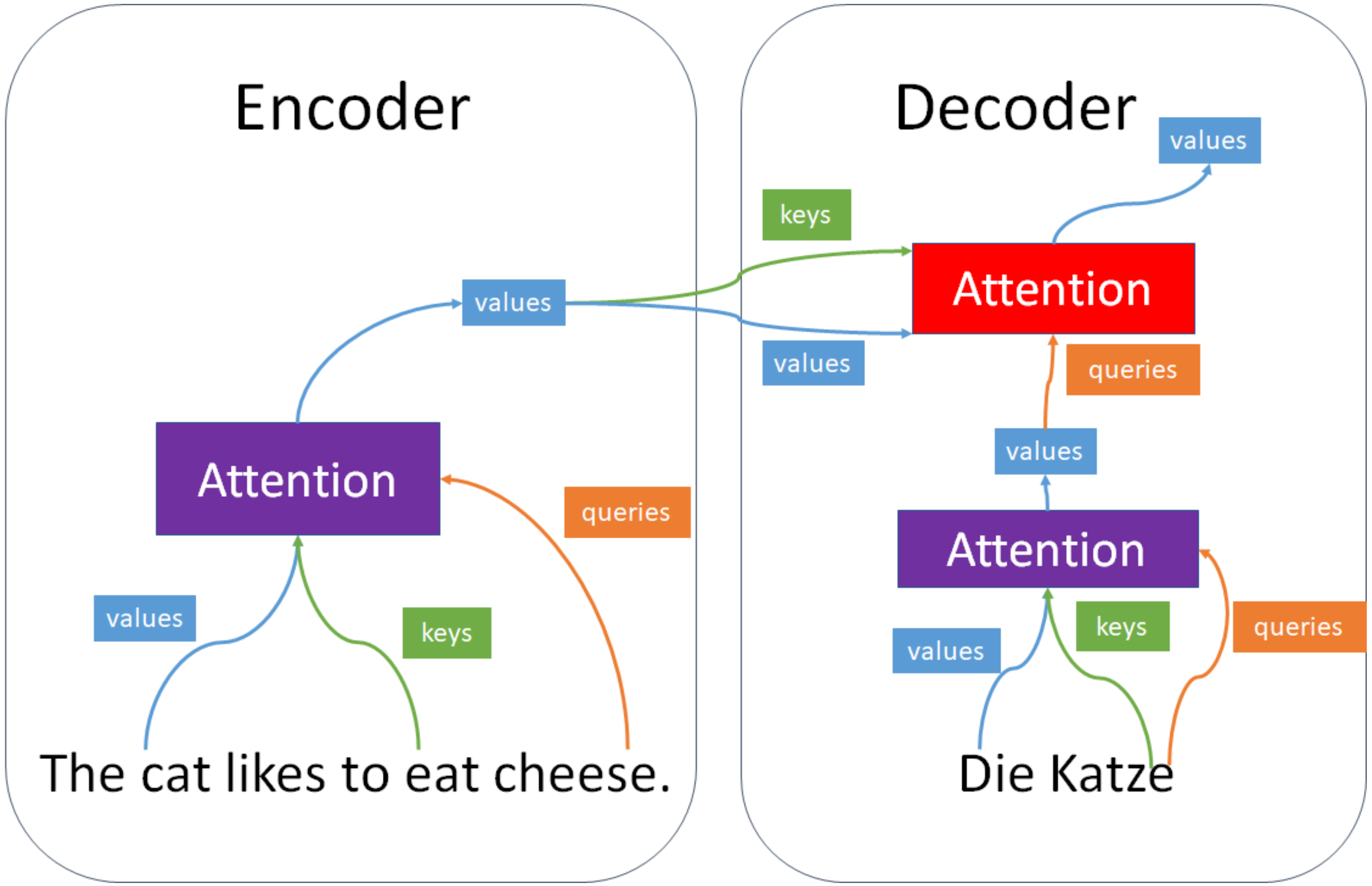
How Transformers Work: A Step-by-Step Breakdown


Output Probabilities
Softmax
Linear
Residual connections and layer normalization
Feed-forward network: after taking information from other tokens, take a moment to think and process this information
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Decoder-encoder attention: target token looks at the source
queries – from decoder states; keys and values from encoder states
N×
Add & Norm
Masked Multi-Head Attention
Decoder self-attention (masked): tokens look at the previous tokens
queries, keys, values are computed from decoder states
Feed-forward network: after taking information from other tokens, take a moment to think and process this information
Add & Norm
Feed Forward
N×
Add & Norm
Multi-Head Attention
Encoder self-attention: tokens look at each other
queries, keys, values are computed from encoder states
Positional Encoding
Input Embedding
Inputs
Positional Encoding
Output Embedding
Outputs (shifted right)

Input Sequence

A sequence of words or tokens that the model will process.

Cross Attention (Co-attention)

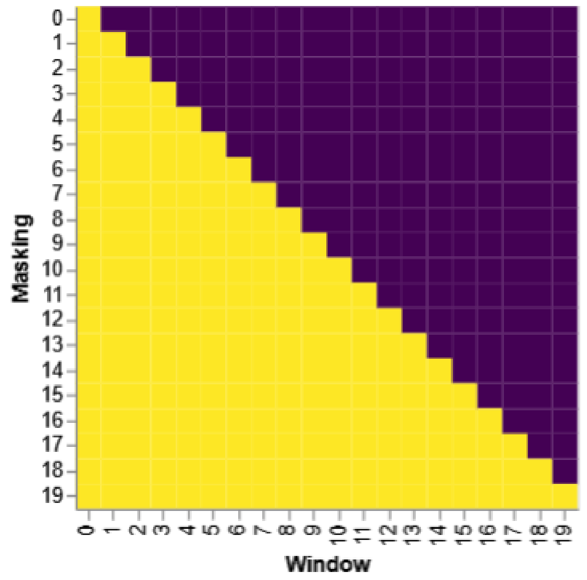


Output Embedding

- The output embedding layer and position encoding function are the same as in the encoder stack
- The output is a translation (or seq-seq transformation) we need to learn
- For example for an English to French translation
    - Input = The black cat sat on the couch and the brown dog slept on the rug
    - Output = Le chat noir était assis sur le canapé et le chien marron dormait sur le tapis
- The output words/tokens go through the word embedding layer and then the positional encoding function, like in the first layer of the encoder stack

Masked multi-head attention

- One of the self-attention sub-layer in the decoder stack is modified to prevent positions from attending to subsequent positions.
- This masking, combined with fact that the output embeddings are offset by one position, ensures that the predictions for position *i* can depend only on the known outputs at positions less than *i*.


Masking
Window