Transformers

Sequence to Sequence

Used when input or/and outputs are sequence

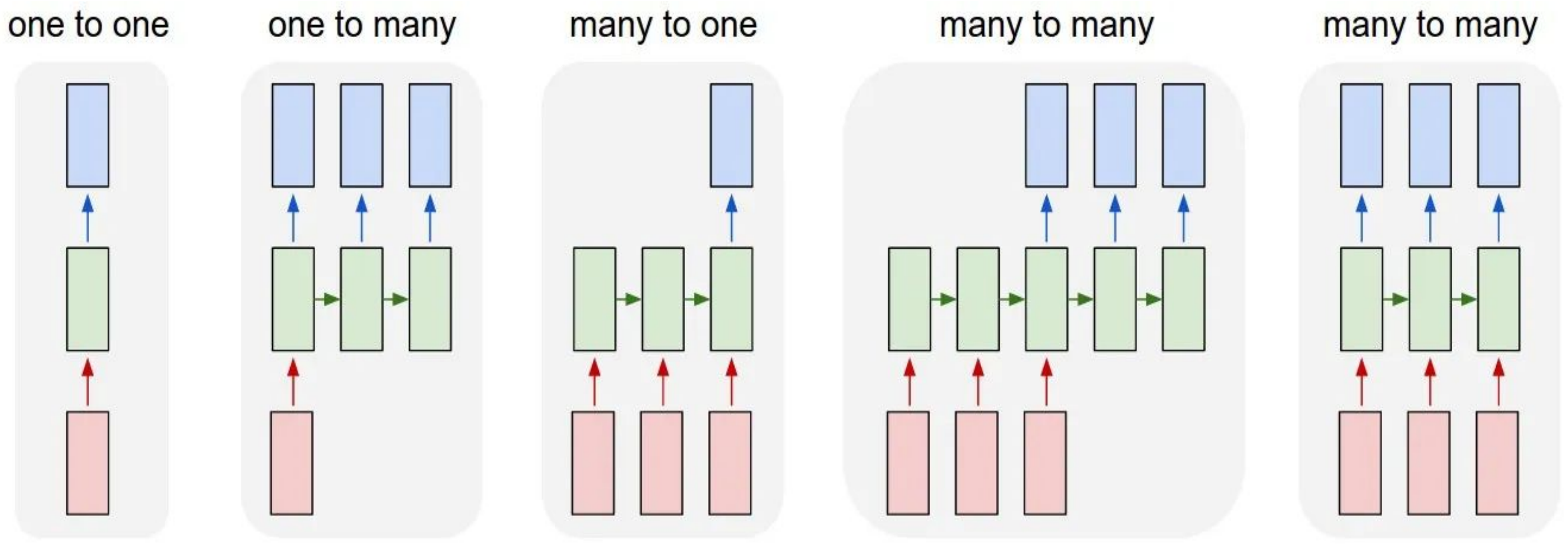
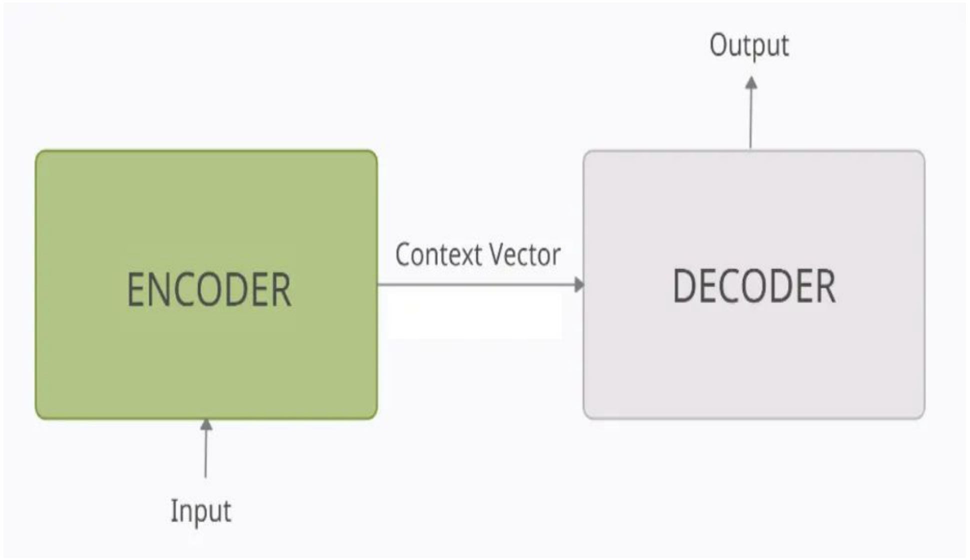


Image Classification Image captioning Sentiment Analysis Machine Translation Video Classification

Encoder - Decoder architecture

Encoder - Decoder architecture



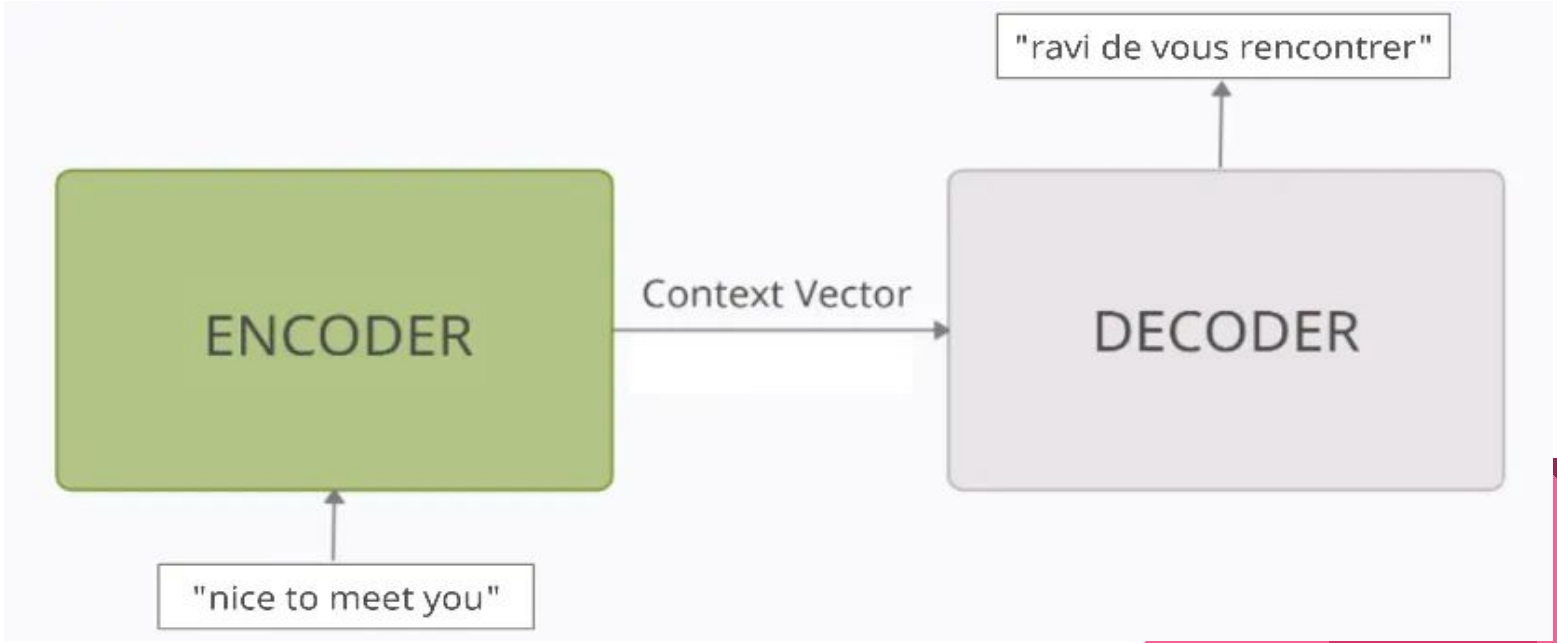
Encoder: The encoder processes each token in the input-sequence. It passes this information to decoder in form of context vector

Context vector: The vector is built in such a way that it's expected to encapsulate the whole meaning of the input-sequence and help the decoder make accurate predictions

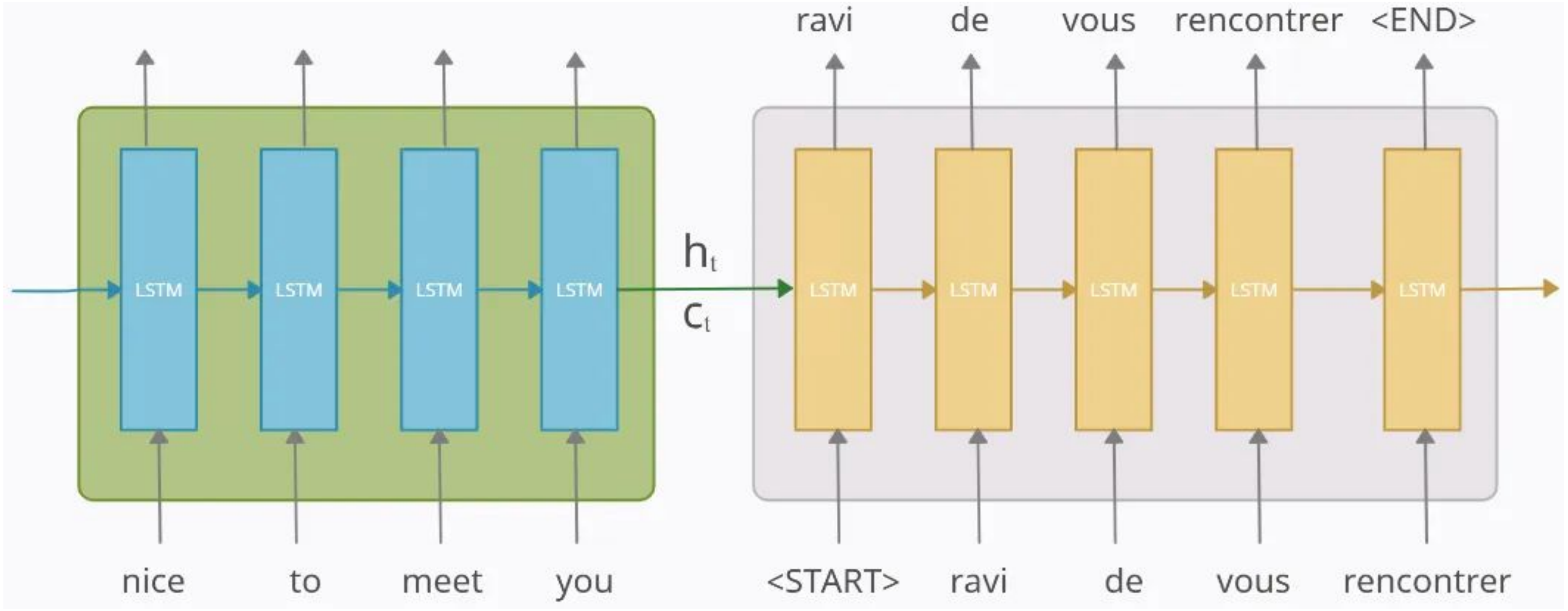
Decoder: The decoder reads the context vector and tries to predict the target-sequence token by token.

Encoder - Decoder model

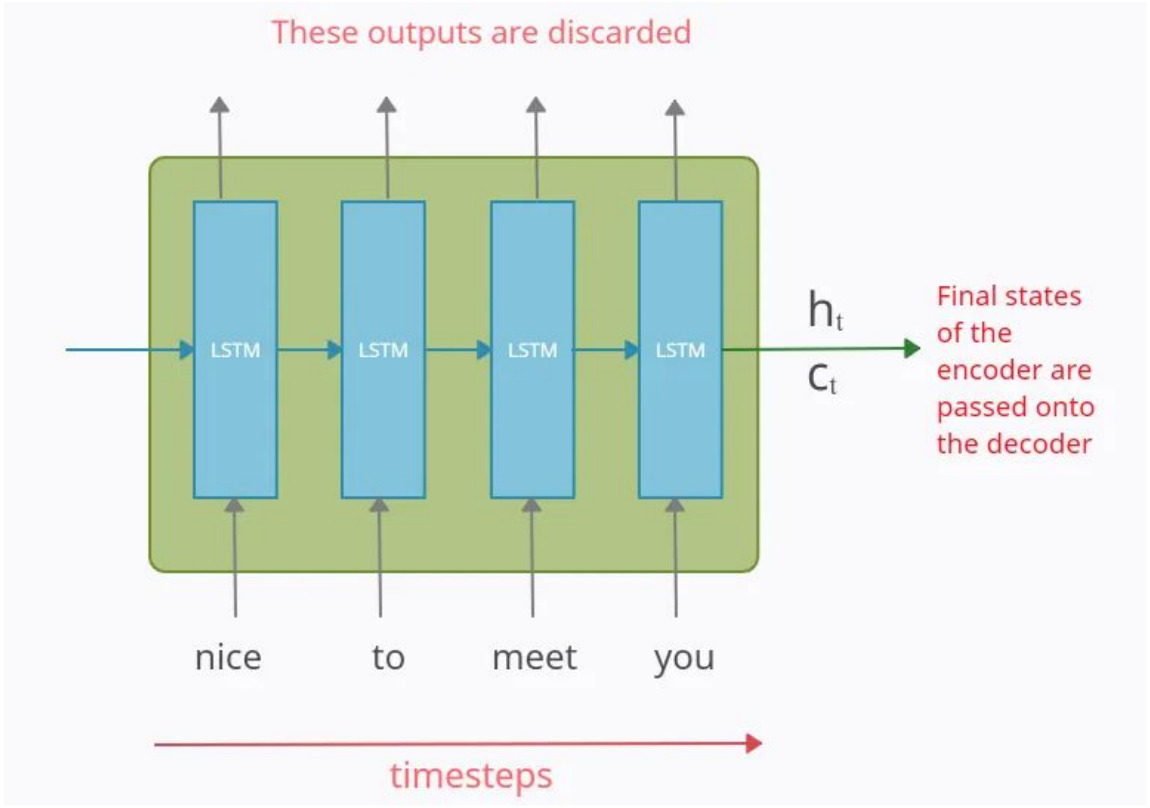
1. Encoder-Decoder models were originally built to solve such Seq2Seq problems.



Internal structure



Encoder part

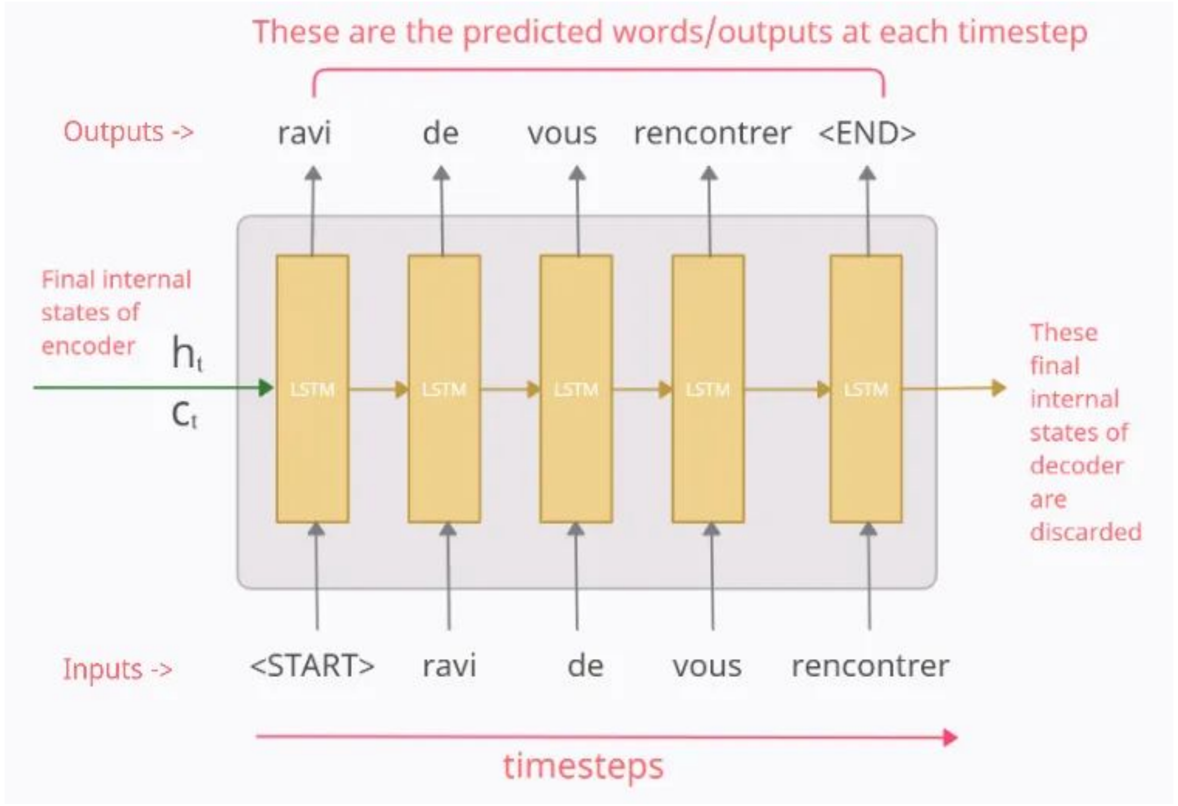

These outputs are discarded
LSTM
LSTM
LSTM
LSTM
h_t
c_t
Final states of the encoder are passed onto the decoder
nice
to
meet
you
timesteps

1. The encoder part is an LSTM cell. It is fed in the input-sequence over time and it tries to encapsulate all its information and store it in its final internal states **h**□ (hidden state) and **c**□ (cell state).

2. The internal states are then passed onto the decoder part, which it will use to try to produce the target-sequence.

3. This is the '**context vector**' which we were earlier referring to.

Decoder part


These are the predicted words/outputs at each timestep
Outputs ->
ravi
de
vous
rencontrer
<END>
Final internal states of encoder
h_t
c_t
LSTM
LSTM
LSTM
LSTM
LSTM
These final internal states of decoder are discarded
Inputs ->
<START>
ravi
de
vous
rencontrer
timesteps

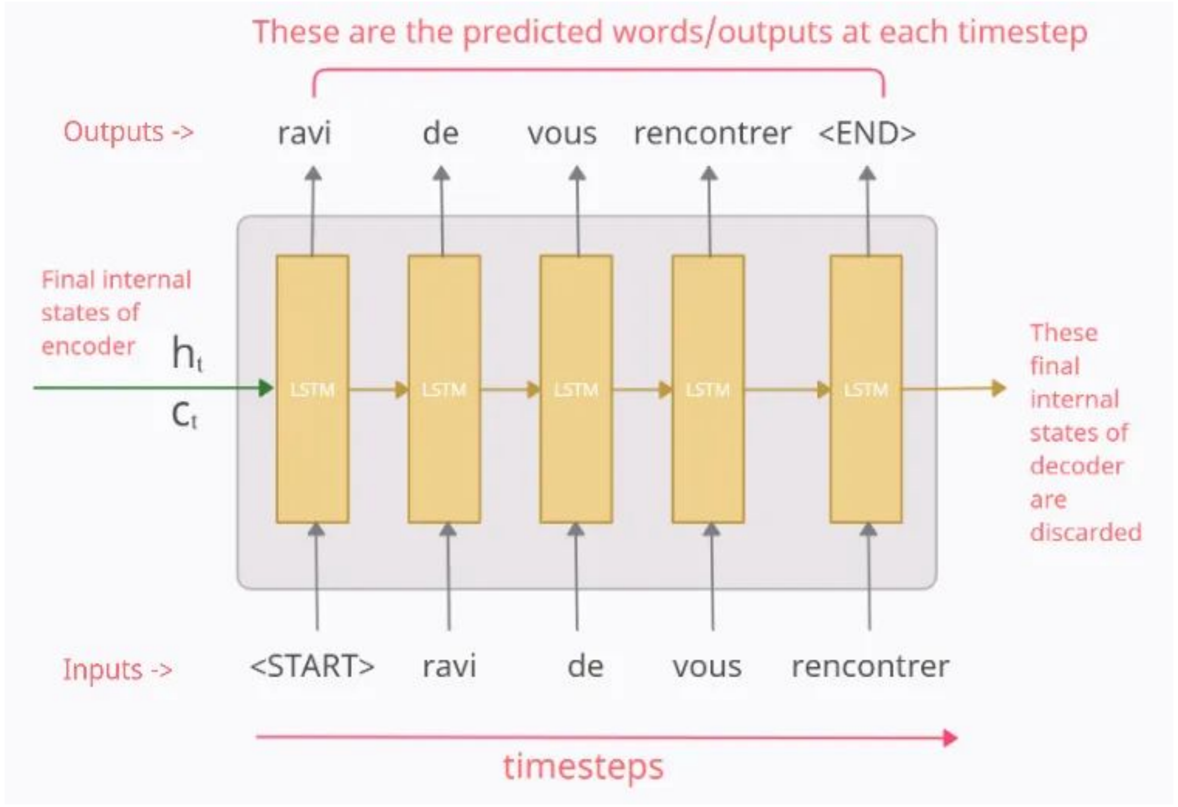
1. Decoder receives the **context vector (ht,ct)** from the encoder

2. Decoder is also a LSTM. It uses the context vector aka **ht,ct** as the initial states **(h_0, c_0)** of the decoder

3. Now the way decoder works, is, that its output at any time-step **t** is supposed to be the **tth** word in the target-sequence/Y_true

4. Now at the initial step a token of <START> is used as input to decoder LSTM and using the context vector as cell and hidden states decoder predicts first word.

Decoder part

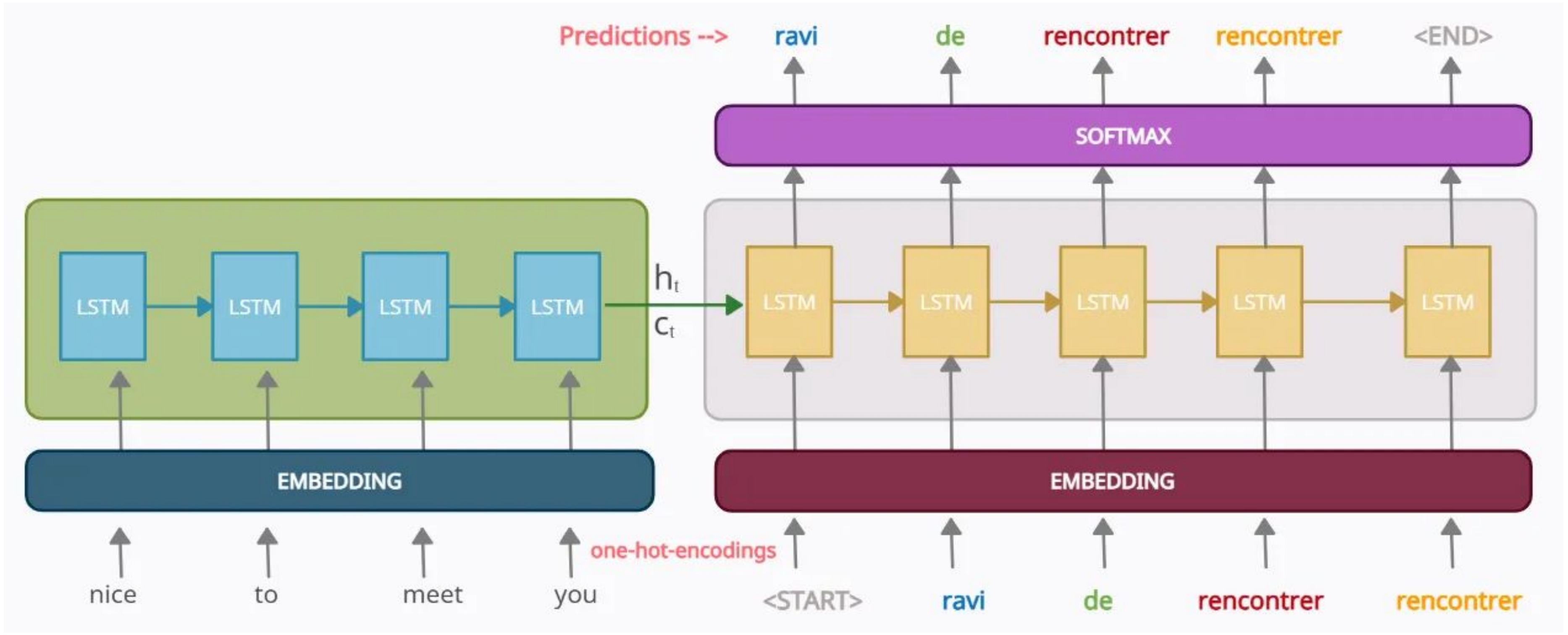

These are the predicted words/outputs at each timestep
Outputs ->
ravi
de
vous
rencontrer
<END>
Final internal states of encoder
h_t
c_t
LSTM
LSTM
LSTM
LSTM
LSTM
These final internal states of decoder are discarded
Inputs ->
<START>
ravi
de
vous
rencontrer
timesteps

1. Which is ravi in this case. Now this predicted word is fed as input in next step and updated hidden and cell states are used to calculate next word aka **de**

2. This is continued till the output of any timestep of decoder is not **<END>**

3. Once we receive end token the prediction is stopped

Overall process



Problem with the using Encoder Decoder

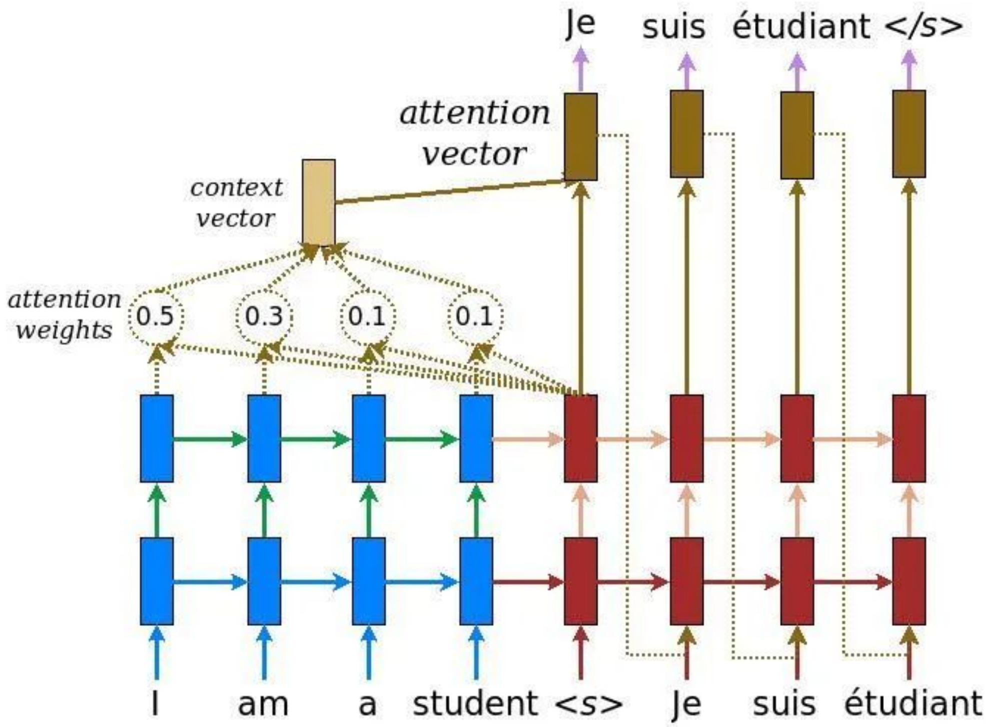
A potential issue with this Encoder-Decoder approach is that:

1. a neural network needs to be able to compress all the necessary information of a source sentence into a fixed-length vector as the encoder encodes all the words or tokens into single context vector which is sent to the decoder which should decode on the basis of it.

2. This may make it difficult for the neural network to cope with long sentences, especially those that are longer than the sentences in the training corpus as most of the information regarding the initial tokens is lost in this process.

3. To solve this issue NMT with attention was introduced in 2015 which was one of the pioneer work for the transformer model or self attention

NMT with attention

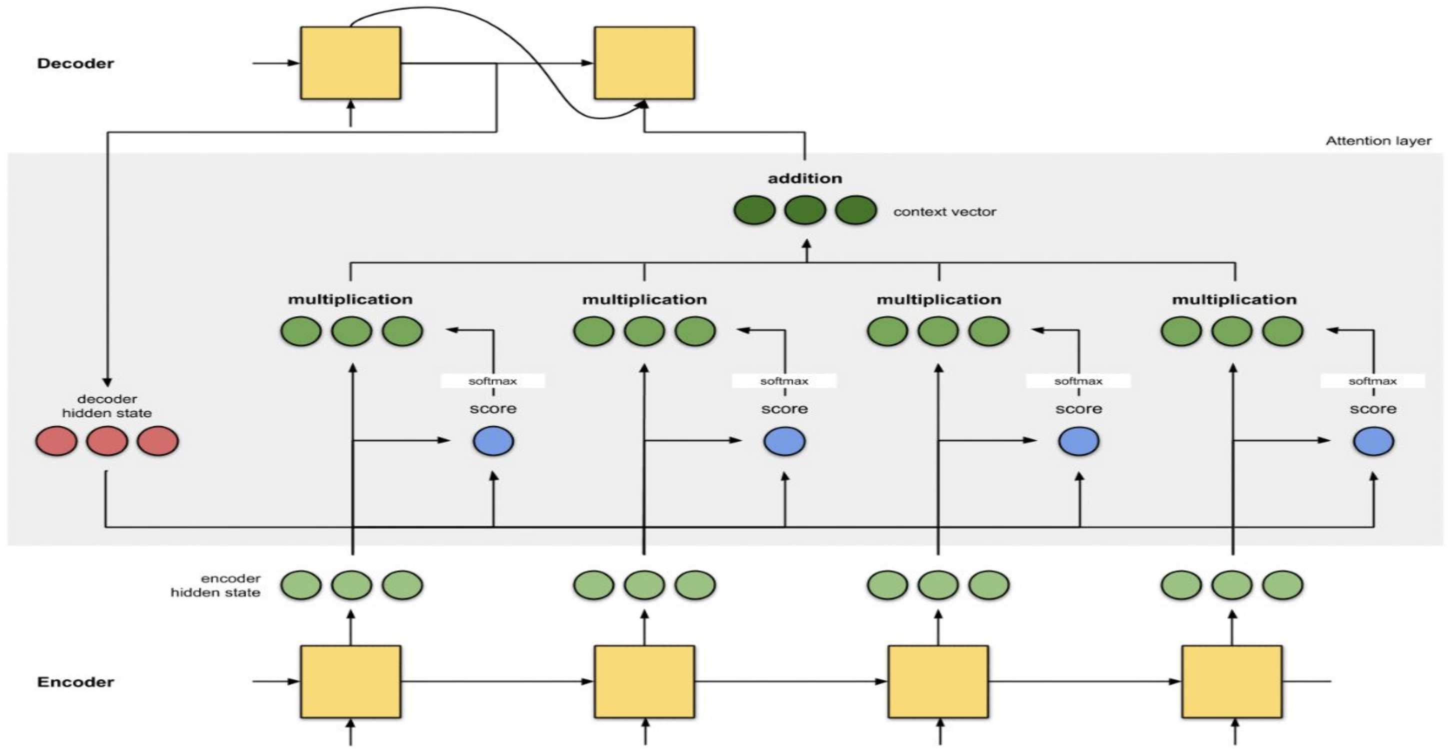


- All hidden states of the encoder and the decoder are used to generate the context vector, unlike how just the last encoder hidden state is used in seq2seq without attention.
- The attention mechanism aligns the input and output sequences, with an alignment score which can be dot product based softmax score or parameterized by a feed-forward network. It helps to pay attention to the most relevant information in the source sequence.
- The model predicts a target word based on the context vectors associated with the source position and the previously generated target words.

Attention explained

Encoder

Attention continued



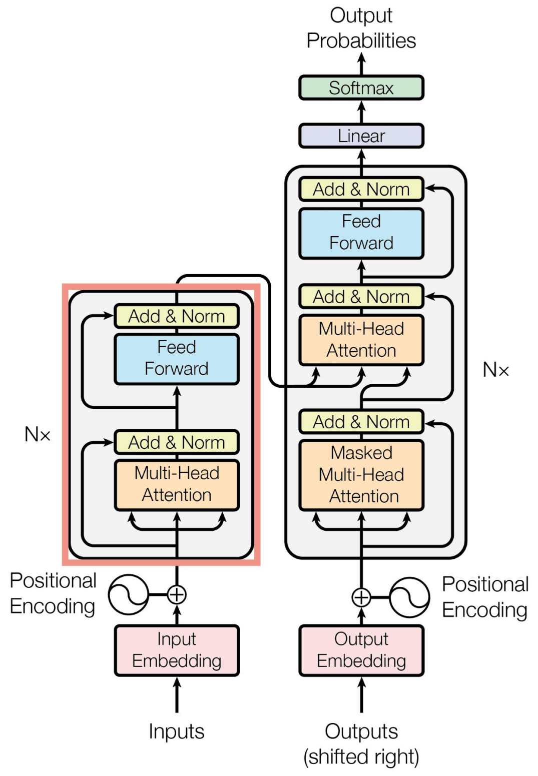
Transformers - Dawn of an era

Transformers

1. From 2017 transformers have completely replaced the rnns and lstms based architectures
2. The first transformer model was introduced for NMT
3. It is also and encoder decoder architecture
4. Recent models were developed just encoder based like bert family models or decoder architectures like GPT models
5. Today we would be looking at just encoder models
6. Like encoder decoder model, encoder in transformer also encodes the data .i.e understand the representation and converts it in vector representation which can be used for further tasks like classification etc.

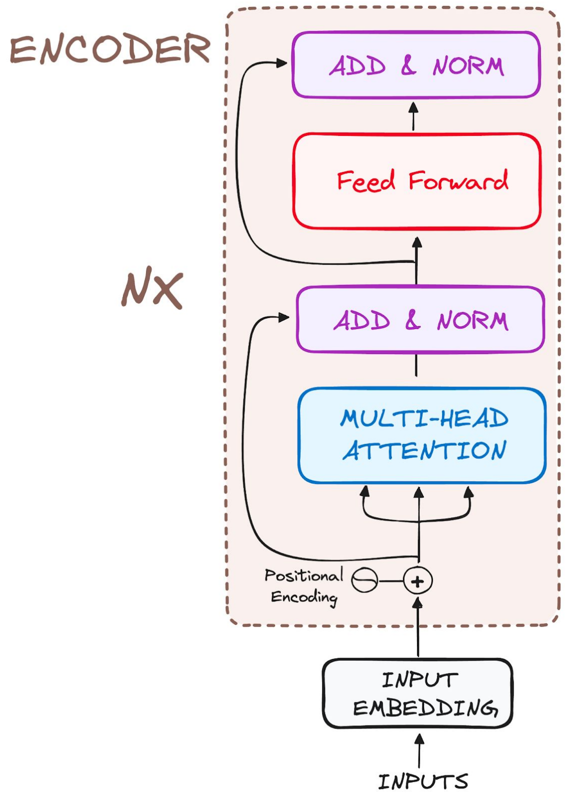
<s>

Architecture of transformers



Encoder Input: hello how are you
Decoder input: <s> namaste aap kaise hai
Target : Namaste aap kaise hai -target
<s> namaste aap kaise hai
I am trying to understand: next word prediction task

Architecture of Encoder in transformer



1. The encoder is a fundamental component of the Transformer architecture.
2. The primary function of the encoder is to transform the input tokens into contextualized representations.
3. Unlike earlier models that processed tokens independently, the Transformer encoder captures the context of each token with respect to the entire sequence by means of attention
4. We can stack multiple transformer blocks on top of each other similar to the original paper

Components of Transformer Encoder

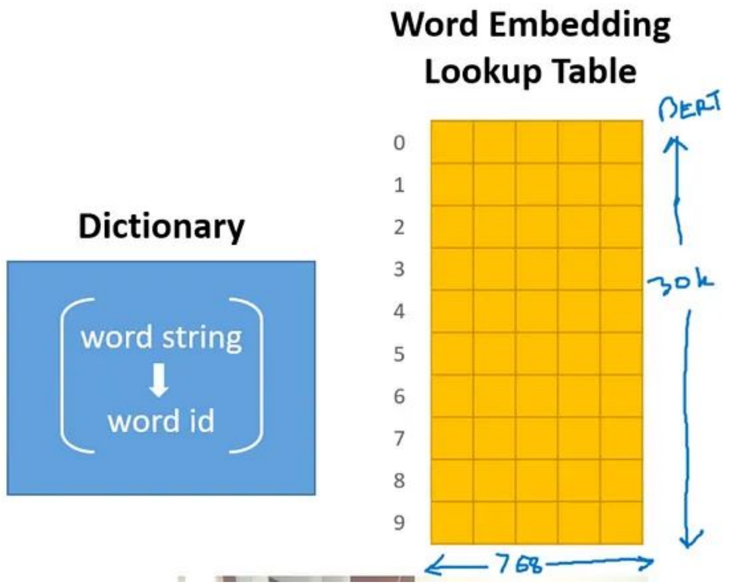
Input processing

Word Embeddings

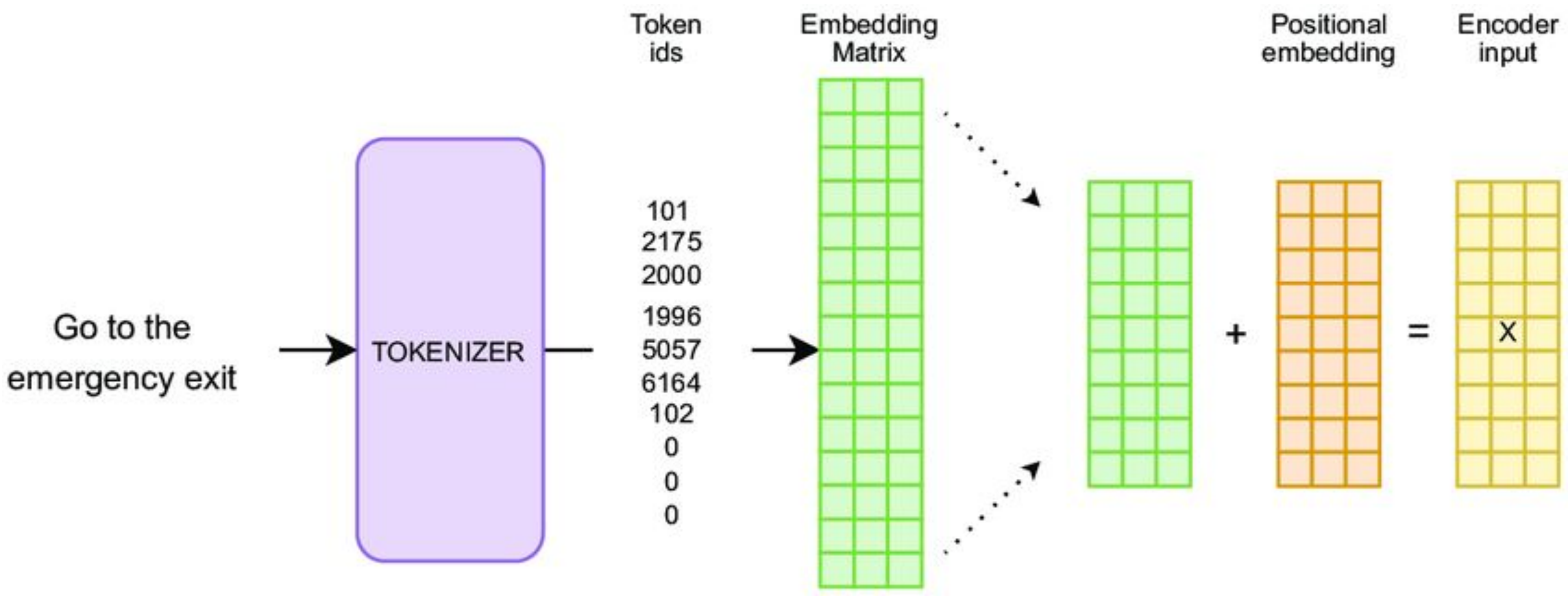
"Word Embedding" = Feature Vector representation of a word

e.g. 300

<0.4125, -1.6098, 0.6047, ..., -1.4257, -1.2321>



Input Embeddings

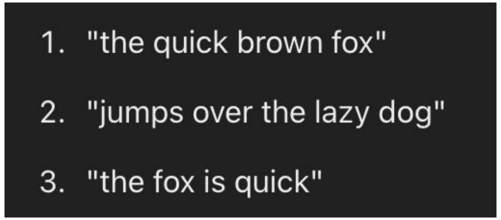


Generation of token ids: text vectorisation

1. Input to the model will be a sequence for e.g consider a movie review for sentiment analysis : The movie was terrible and badly made
2. Now we would first tokenize the input on word, char or subword level. We will be using word level.
3. Before tokenisation we can apply standardisation to reduce the unique tokens by converting the text to lowercase and removing punctuations.
4. Using the whole vocabulary we can learn the mapping between tokens to the integer indexes
5. We can use uncapped vocabulary mapping all the unique tokens or we can cap them and use the token OOV to handle the remaining tokens
6. We can also specify max length of the input sequence such that the sequences above the max length would be truncated and for lesser a padding of 0 would be added. So 0 can be considered as padding token.

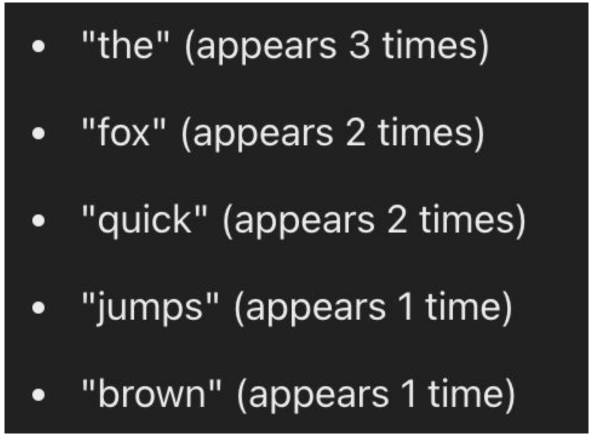
Process of text vectorisation

1. Dataset

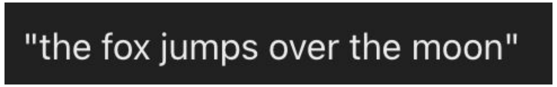


1. "the quick brown fox"
2. "jumps over the lazy dog"
3. "the fox is quick"

2.Using 6 as max tokens we would use 5 most frequent tokens and rest as OOV 0 is pad

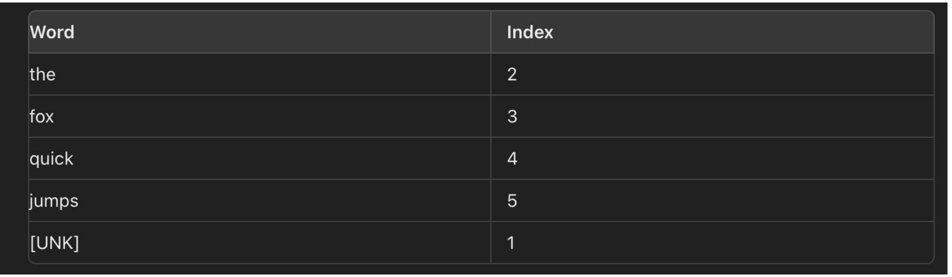


- "the" (appears 3 times)
- "fox" (appears 2 times)
- "quick" (appears 2 times)
- "jumps" (appears 1 time)
- "brown" (appears 1 time)



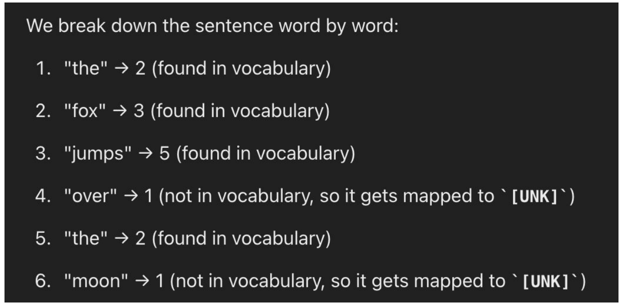
"the fox jumps over the moon"

3.Vocabulary



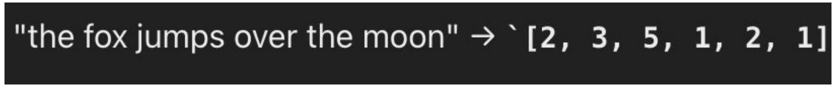
Word	Index
the	2
fox	3
quick	4
jumps	5
[UNK]	1

4.Vectorising input



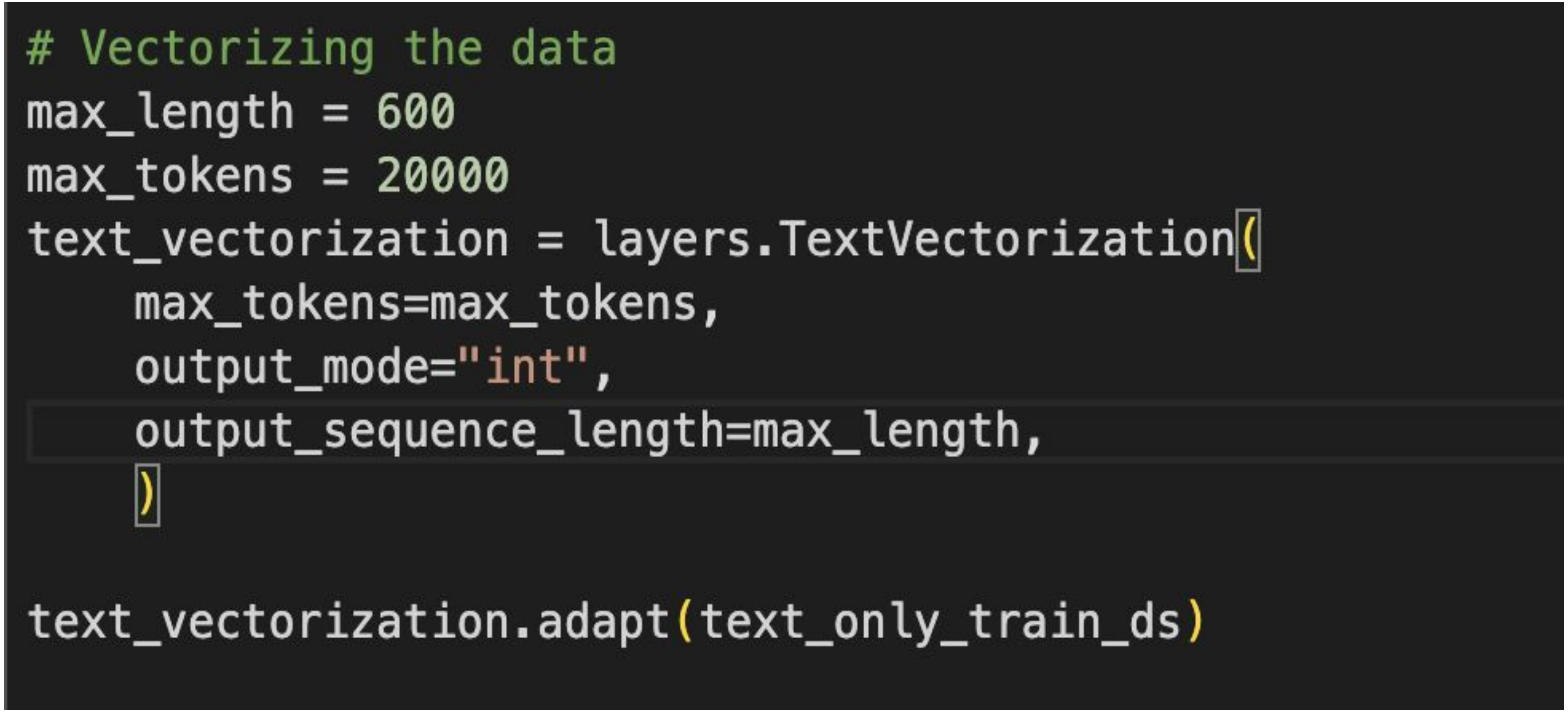
We break down the sentence word by word:

1. "the" → 2 (found in vocabulary)
2. "fox" → 3 (found in vocabulary)
3. "jumps" → 5 (found in vocabulary)
4. "over" → 1 (not in vocabulary, so it gets mapped to `[UNK]`)
5. "the" → 2 (found in vocabulary)
6. "moon" → 1 (not in vocabulary, so it gets mapped to `[UNK]`)



"the fox jumps over the moon" → `[2, 3, 5, 1, 2, 1]

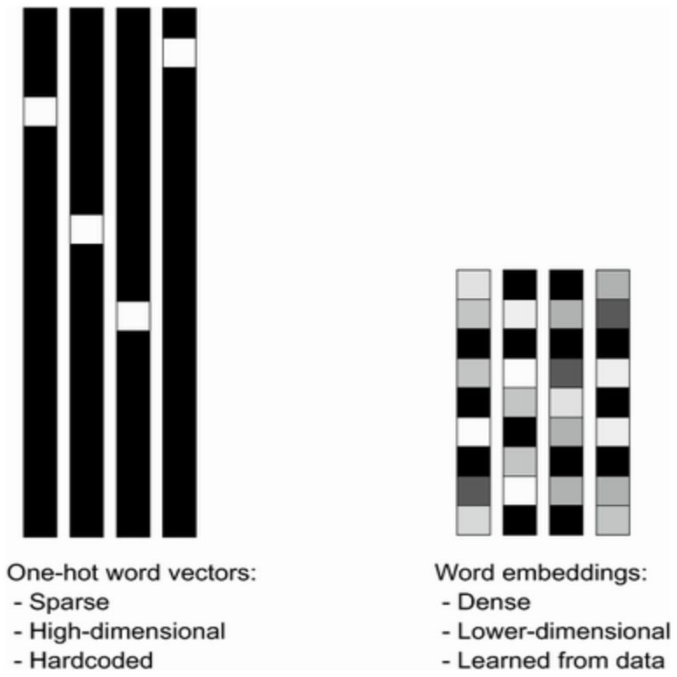
Using tensorflow



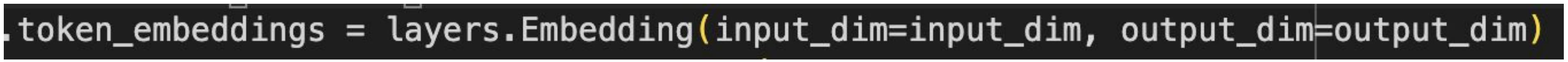
```
# Vectorizing the data
max_length = 600
max_tokens = 20000
text_vectorization = layers.TextVectorization(
    max_tokens=max_tokens,
    output_mode="int",
    output_sequence_length=max_length,
    )

text_vectorization.adapt(text_only_train_ds)
```

Learned Embedding vs one hot embeddings

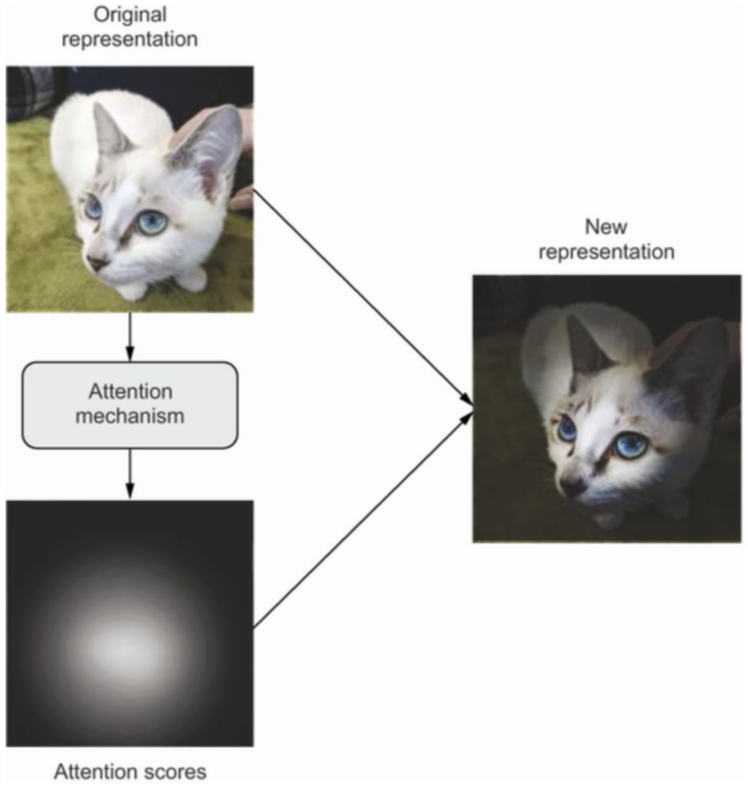


1. We could directly using text vectorisation to output multi-hot or tf-idf vectors which we can use as embeddings
2. But most common trend is to use the embedding layer which instead of hardcoding the values learns the embedding as weight matrix along with the model
3. We can do that by using keras embedding layer. We can provide parameters like embedding size and it would give us for each id of text vectorisation embedding of size embedding size.
4. These embeddings have semantic meaning and words which are synonymous would be placed closer on the space than all the words being orthogonal to each other in case of one hot vectors



```
token_embeddings = layers.Embedding(input_dim=input_dim, output_dim=output_dim)
```

Concept of Attention

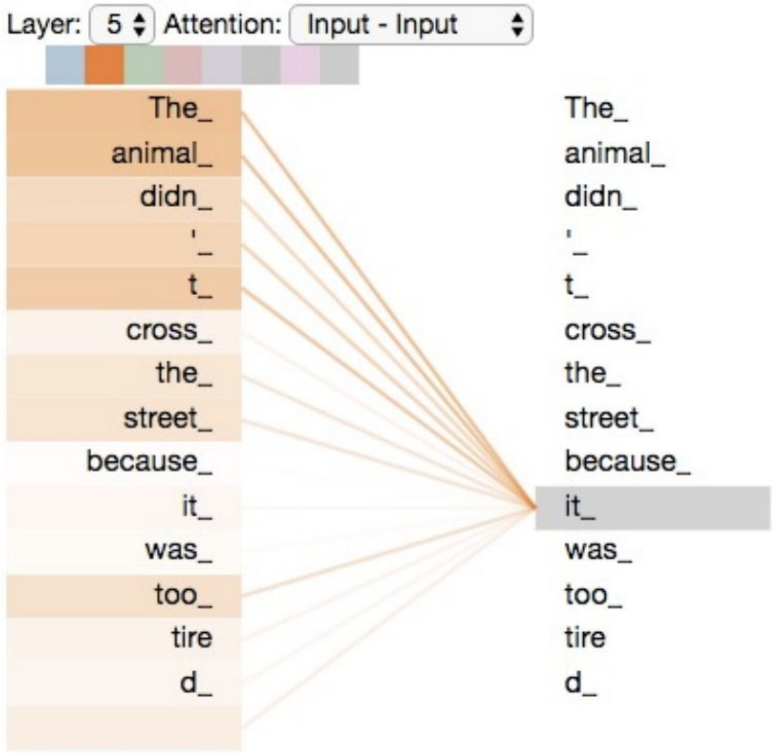


1.This **attention** helps us to focus on important part of input.
2. In case of our **sequence** we generated **embedding** for token which preserves **semantic relationships** between tokens
3. These words can have **fixed** set of relationships with other words and are **not context driven** which is not case in real language we speak
4. For e.g the word "**date**" can mean please mark the date : **(calendar date)**. I went to buy **dates : (fruit)** and They went on date to park: (**outing**).
5. Thus each word can mean different based on its context. **Pronouns** are also **sentence-specific** and can even change multiple times within a single sentence. For e.g Tony has a cat. He named it loki. Loki is orange cat and he loves playing in the morning.
6. We need this **smart embedding** which would also take consideration context: **Self attention** comes in play

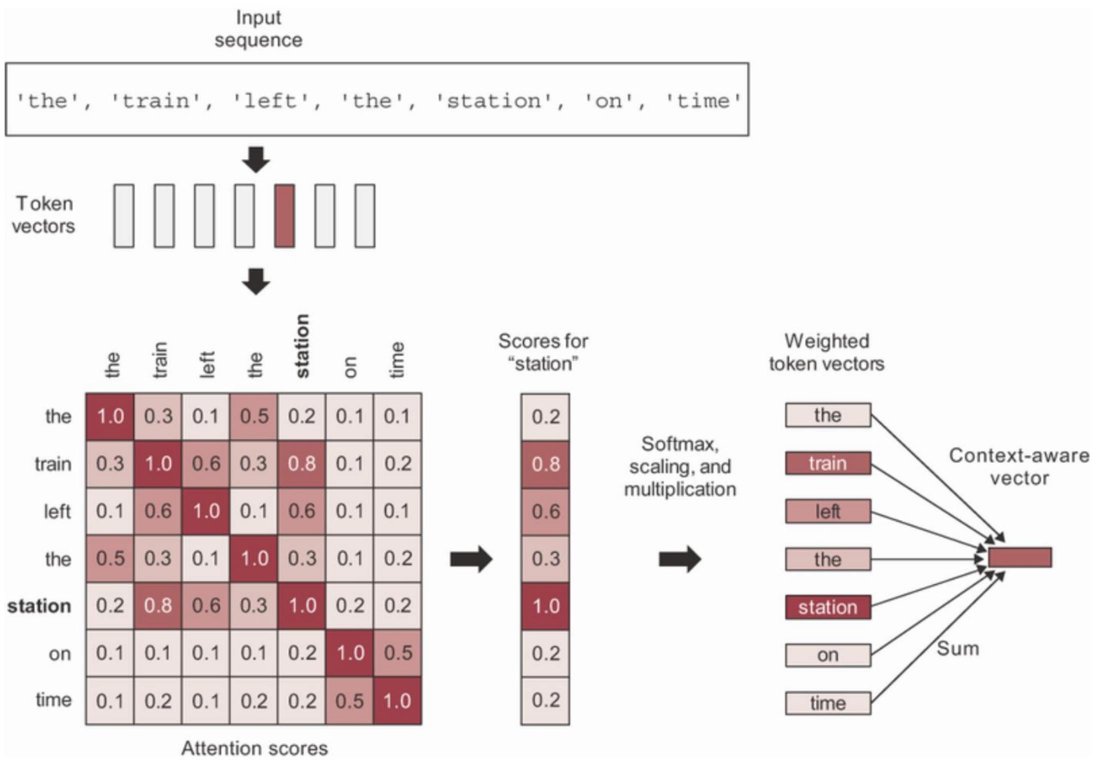
Attention helps us answers such questions

1. **The animal didn't cross the street because it was too tired'.**
2. What does it refer to in above sentence
3. Also it helps us understand in this sentence

   The train left the station on time
4. What does station refer to here, does it mean train station, radio station etc
5. Self attention helps us understanding this



Self attention



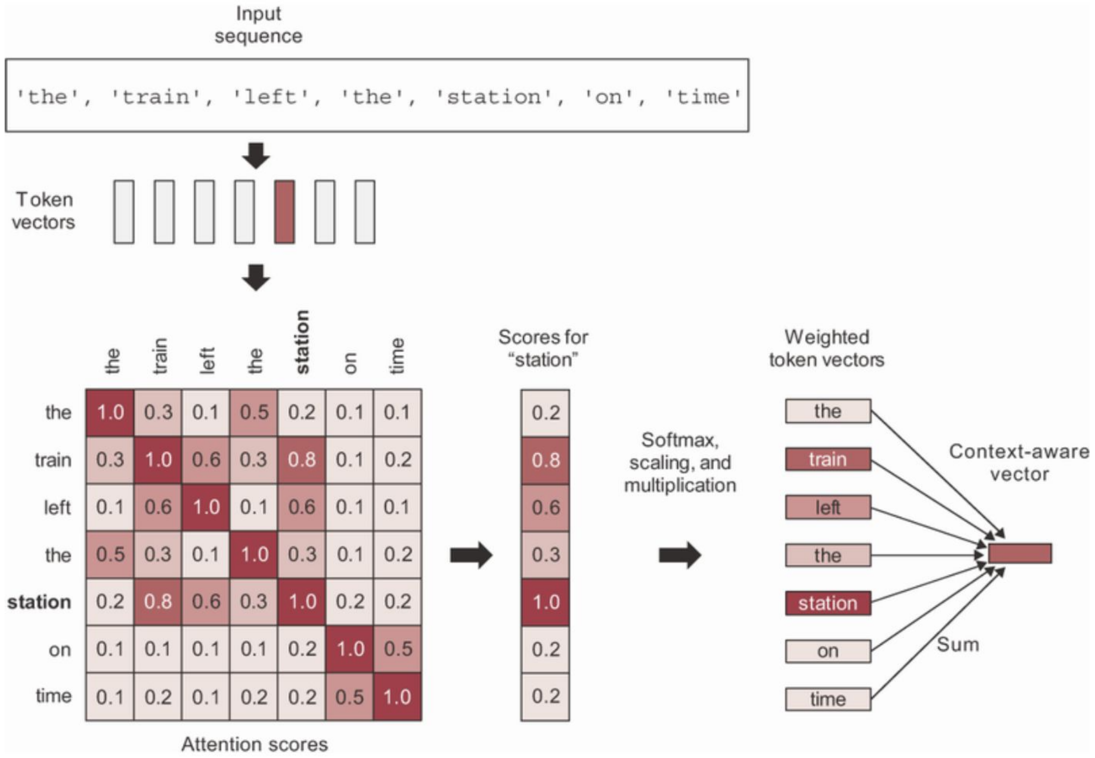
	the	train	left	the	station	on	time
the	1.0	0.3	0.1	0.5	0.2	0.1	0.1
train	0.3	1.0	0.6	0.3	0.8	0.1	0.2
left	0.1	0.6	1.0	0.1	0.6	0.1	0.1
the	0.5	0.3	0.1	1.0	0.3	0.1	0.2
station	0.2	0.8	0.6	0.3	1.0	0.2	0.2
on	0.1	0.1	0.1	0.1	0.2	1.0	0.5
time	0.1	0.2	0.1	0.2	0.2	0.5	1.0

Scores for "station"
0.2
0.8
0.6
0.3
1.0
0.2
0.2

To calculate context rich vector for word station we follow following:

1. We use **embedding vectors** for each word and calculate **similarity** between each word in the sequence to the current word aka station
2. For this we can use cosine similarity or dot product or l2 distance
3. Dot product is used as its intuitive and cost of computation is also less and would give us words scores based on how similar they are.
4. Once we get scores, we apply **softmax** to them to get **normalised scores**. These scores are our **attention weights** which can be use to weight the embeddings of each word.

Self attention



	the	train	left	the	station	on	time
the	1.0	0.3	0.1	0.5	0.2	0.1	0.1
train	0.3	1.0	0.6	0.3	0.8	0.1	0.2
left	0.1	0.6	1.0	0.1	0.6	0.1	0.1
the	0.5	0.3	0.1	1.0	0.3	0.1	0.2
station	0.2	0.8	0.6	0.3	1.0	0.2	0.2
on	0.1	0.1	0.1	0.1	0.2	1.0	0.5
time	0.1	0.2	0.1	0.2	0.2	0.5	1.0

Scores for "station"
0.2
0.8
0.6
0.3
1.0
0.2
0.2

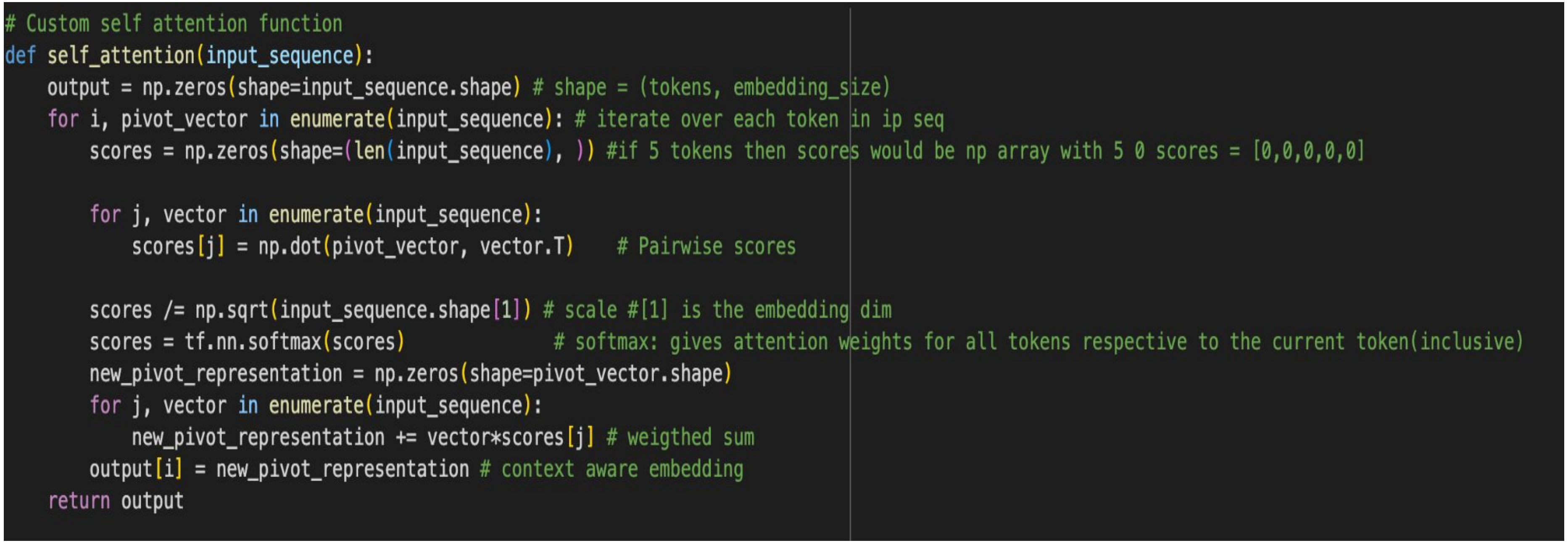
5. Multiplication of attention weight with embedding and adding them should give us our final context rich embedding
6. Words closely related to "station" will contribute more to the sum (including the word "station" itself), while irrelevant words will contribute almost nothing.
7. The resulting vector is our new representation for "station": a representation that incorporates the surrounding context. In particular, it includes part of the "train" vector, clarifying that it is, in fact, a "train station."

currenttoken= the = [1,2,34]
The = [1,2,3,4]
a[0][0] = b
Train = [2,3,4,5]
A[1][0] = c

Pseudocode of self attention

1. We want to iterate over each token and calculate score vector for all of them
2. For current token to calculate the score vector we have to iterate over all other words for e.g the train left station on time .. to calculate the score vector for station we iterate over all the 6 words
3. Then we calculate dot product of each word with the current token aka station
4. Then we apply softmax and multiply embedding of the word with their corresponding attention scores and all of weighted embeddings together
5. THis gives us context based embedding for one word.
6. We continue the same for rest .
7. The final context based embedding would be a matrix of (no of words, embedding size)

Pseudocode of self attention



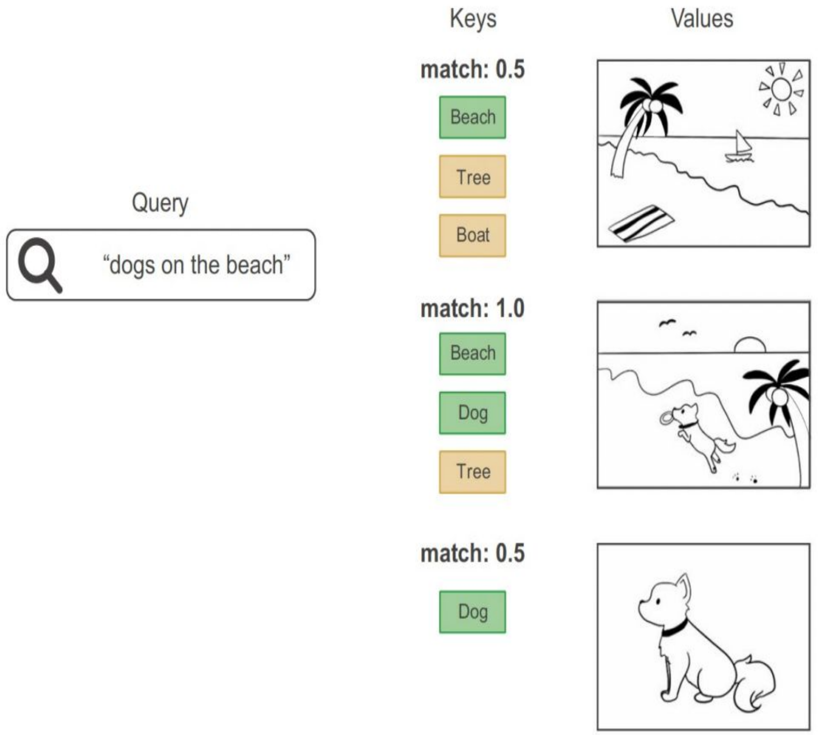
```
# Custom self attention function
def self_attention(input_sequence):
    output = np.zeros(shape=input_sequence.shape) # shape = (tokens, embedding_size)
    for i, pivot_vector in enumerate(input_sequence): # iterate over each token in ip seq
        scores = np.zeros(shape=(len(input_sequence), )) #if 5 tokens then scores would be np array with 5 0 scores = [0,0,0,0,0]

        for j, vector in enumerate(input_sequence):
            scores[j] = np.dot(pivot_vector, vector.T)    # Pairwise scores

        scores /= np.sqrt(input_sequence.shape[1]) # scale #[1] is the embedding dim
        scores = tf.nn.softmax(scores)              # softmax: gives attention weights for all tokens respective to the current token(inclusive)
        new_pivot_representation = np.zeros(shape=pivot_vector.shape)
        for j, vector in enumerate(input_sequence):
            new_pivot_representation += vector*scores[j] # weigthed sum
        output[i] = new_pivot_representation # context aware embedding
    return output
```

Generalized self-attention: The query-key-value model

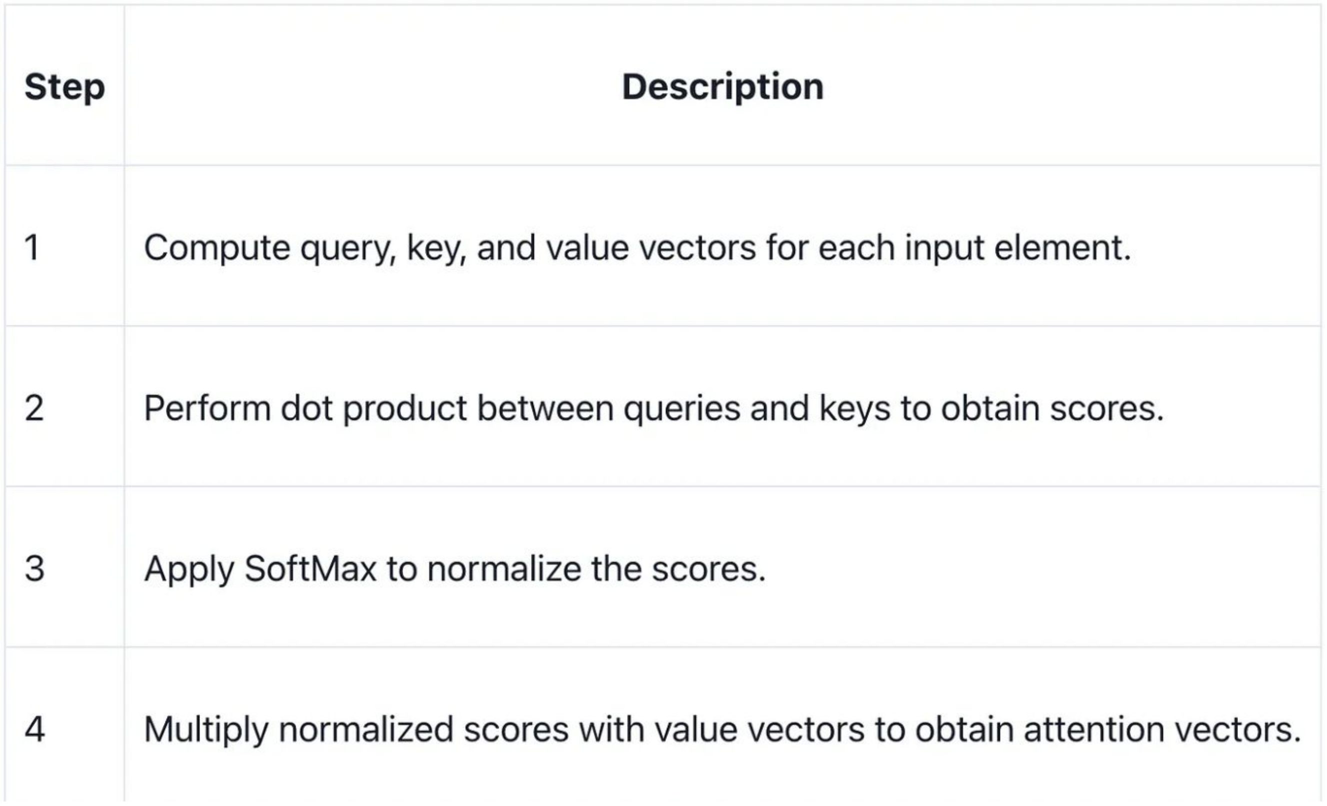
1.We computed the Self Attention based on the inputs of vectors themselves.
2. This means that for fixed inputs, these attention weights would always be fixed.
3.In other words, there are no learnable parameters. Need to introduce some learnable parameters which will make the self attention mechanism more flexible and tunable for various tasks.
4.To fulfill this purpose, three weight matrices are introduced and multiplied with input
xi separately and three new terms **Queries(Q), Keys(K) and Values(V)** comes into picture as given by equations below.
5. These queries ,key and value vectors can be considered analogous to the database or retrieval systems.



Importance of QKV

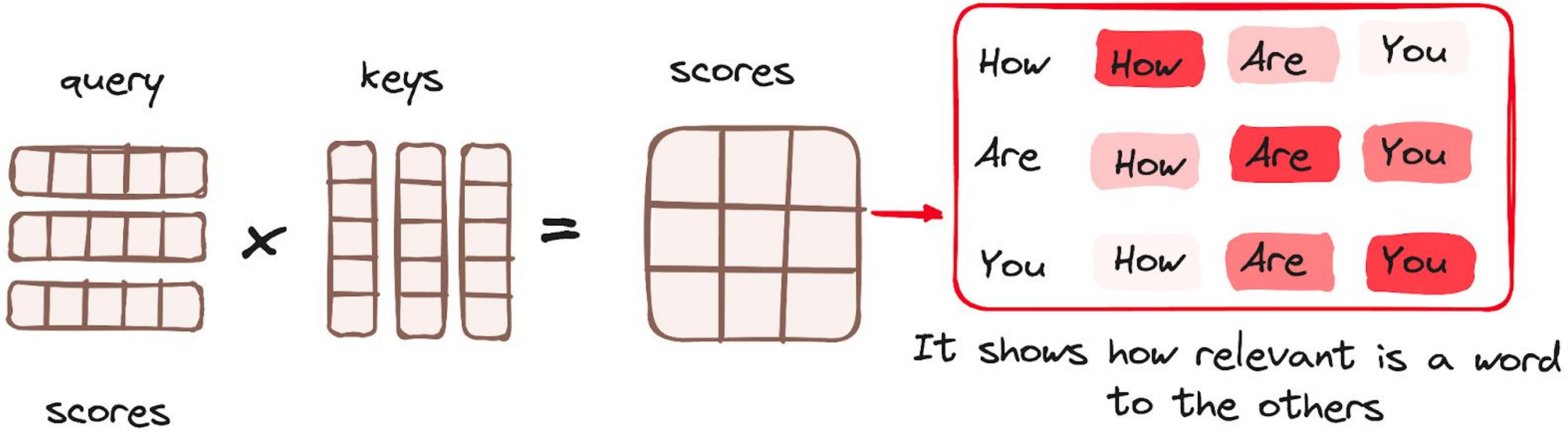
1. For e.g for station when we multiply the learned Q matrix with embedding the query vector generated can intuitively as queries like what what action is associated with station, what entity is associated with etc
2. For such queries now from the remaining words key vectors where key of train for e.g can be entity , key vector for left can be action. When we multiply this with such query vectors we will get higher similarity scores which means that these words will have higher representations in the final context aware vectors
3. Thus we can learn various queries using these learned Q matrices which was lacking when we used static embeddings. Similarly for key and value matrices

Process of self attention with QKV



Step	Description
1	Compute query, key, and value vectors for each input element.
2	Perform dot product between queries and keys to obtain scores.
3	Apply SoftMax to normalize the scores.
4	Multiply normalized scores with value vectors to obtain attention vectors.

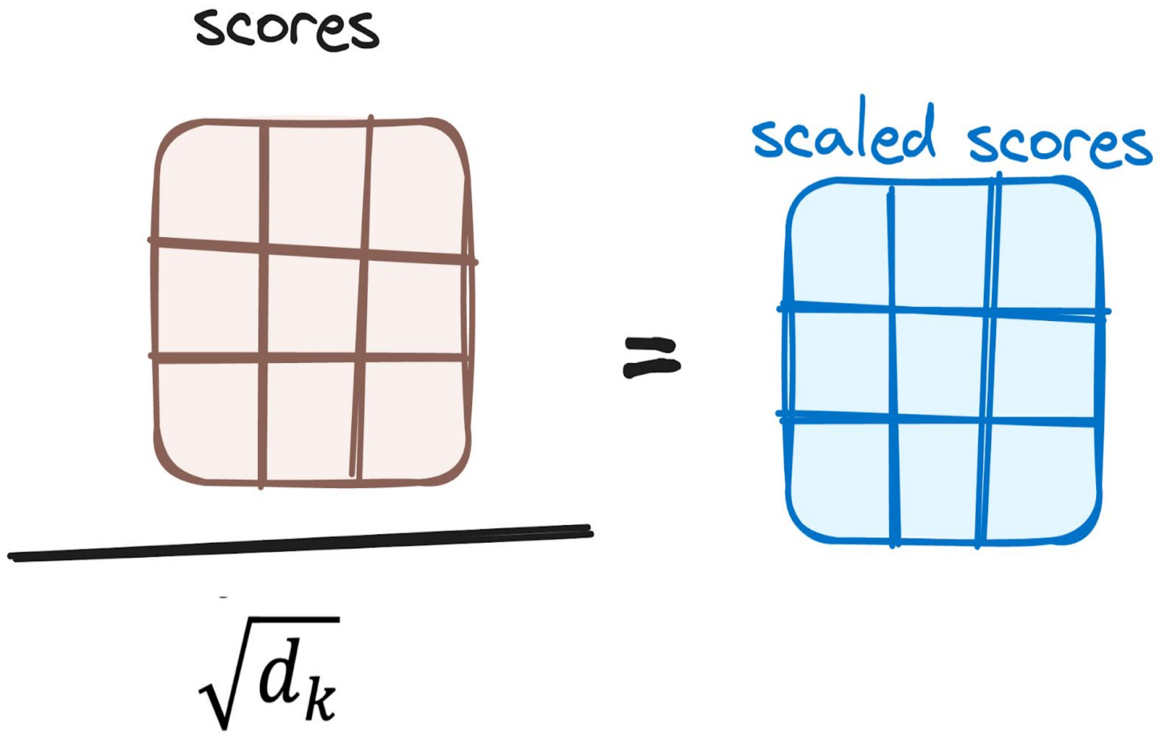
Visualisation of process



Example

```
How = [1,2,3,4]
Are = [2,4,8,7]
You = [1,2,1,2]
Wq matrix =[[1,1,1]
            [0,0,0]
            [2,1,1]
            [3,2,4]]
Query matrix = [[1,2,3,4],  * Wq matrix
                [2,4,8,7],
                [1,2,1,2]]
               3x4 (t x dmodel)  *  4x3 (dmodel x dk)
               3*3
```

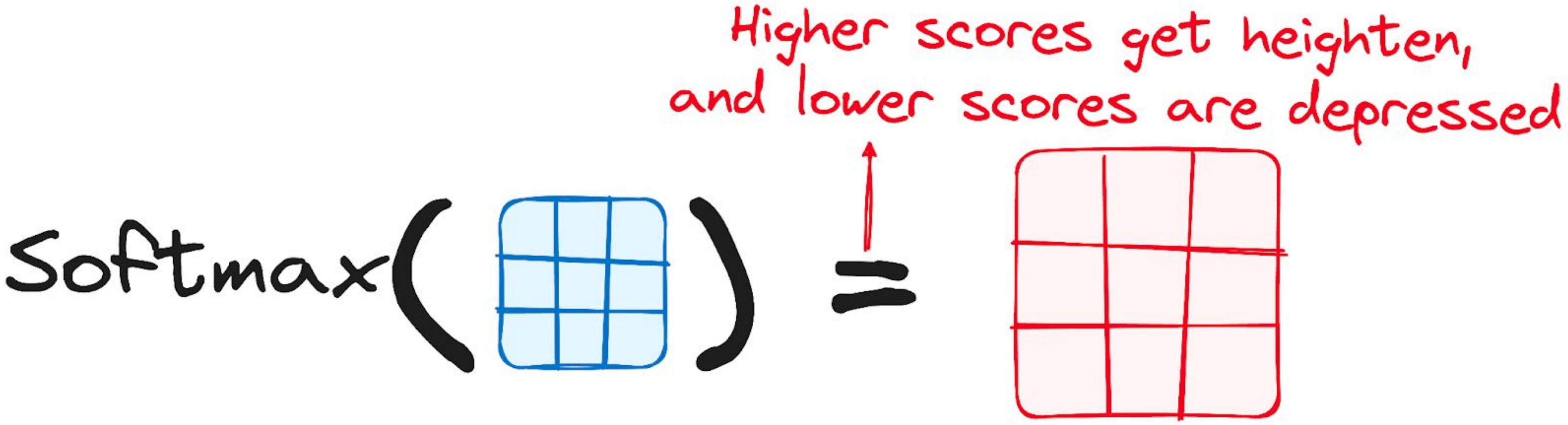
Scaling of scores



$$\frac{\text{scores}}{\sqrt{d_k}}$$

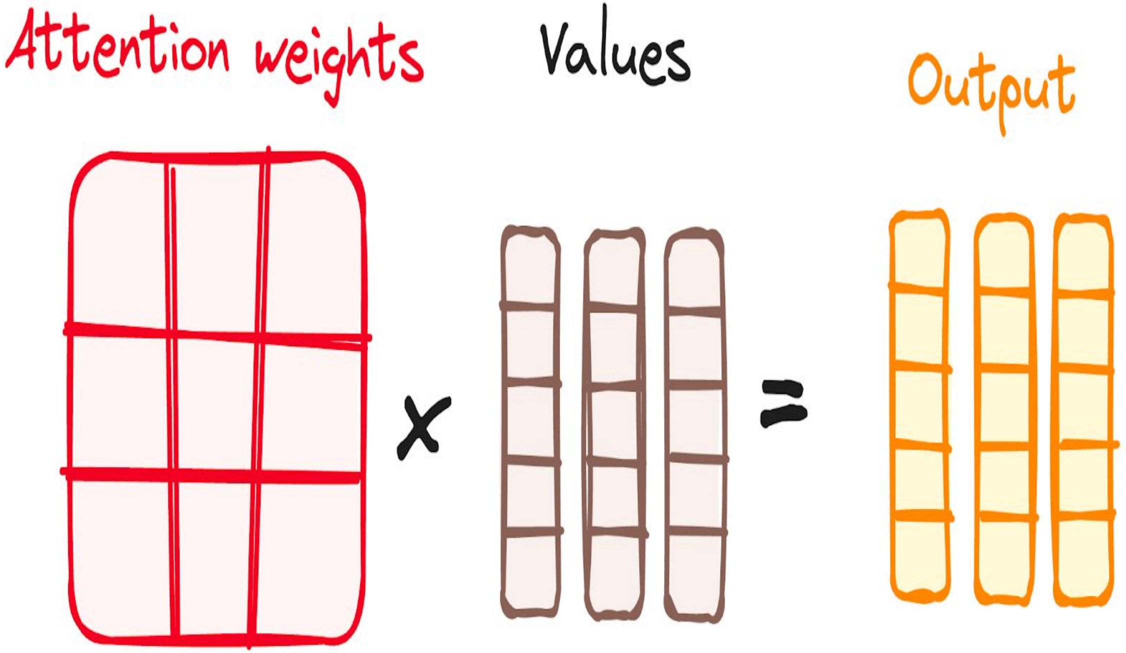
The scores are then scaled down by dividing them by the square root of the dimension of the query and key vectors. This step is implemented to ensure more stable gradients, as the multiplication of values can lead to excessively large effects.

Applying softmax



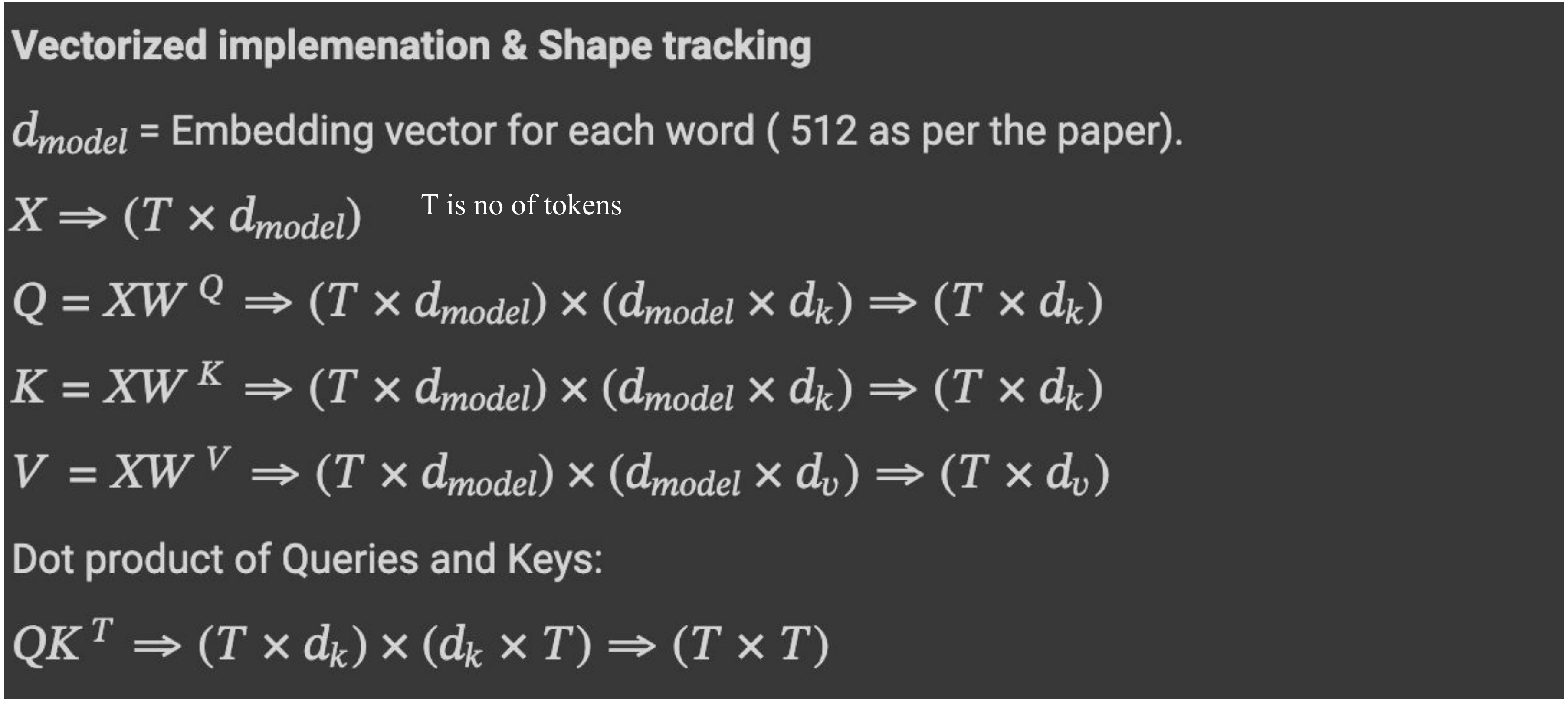
Subsequently, a softmax function is applied to the adjusted scores to obtain the attention weights. This results in probability values ranging from 0 to 1. The softmax function emphasizes higher scores while diminishing lower scores, thereby enhancing the model's ability to effectively determine which words should receive more attention.

Combining Softmax Results with the Value Vector



1. The following step of the attention mechanism is that weights derived from the softmax function are multiplied by the value vector, resulting in an output vector.
2. In this process, only the words that present high softmax scores are preserved. Finally, this output vector is fed into a linear layer for further processing.

Implementation of self attention with QKV



Vectorized implemenation & Shape tracking

d_{model} = Embedding vector for each word (512 as per the paper).

$X \Rightarrow (T \times d_{model})$ T is no of tokens

$$Q = XW^{Q} \Rightarrow (T \times d_{model}) \times (d_{model} \times d_k) \Rightarrow (T \times d_k)$$

$$K = XW^{K} \Rightarrow (T \times d_{model}) \times (d_{model} \times d_k) \Rightarrow (T \times d_k)$$

$$V = XW^{V} \Rightarrow (T \times d_{model}) \times (d_{model} \times d_v) \Rightarrow (T \times d_v)$$

Dot product of Queries and Keys:

$$QK^{T} \Rightarrow (T \times d_k) \times (d_k \times T) \Rightarrow (T \times T)$$

Multihead Attention Overview

Core of the Transformer is Self Attention as in the original paper: "**Attention** is all you need".

arxiv.org/pdf/1706.03762.pdf

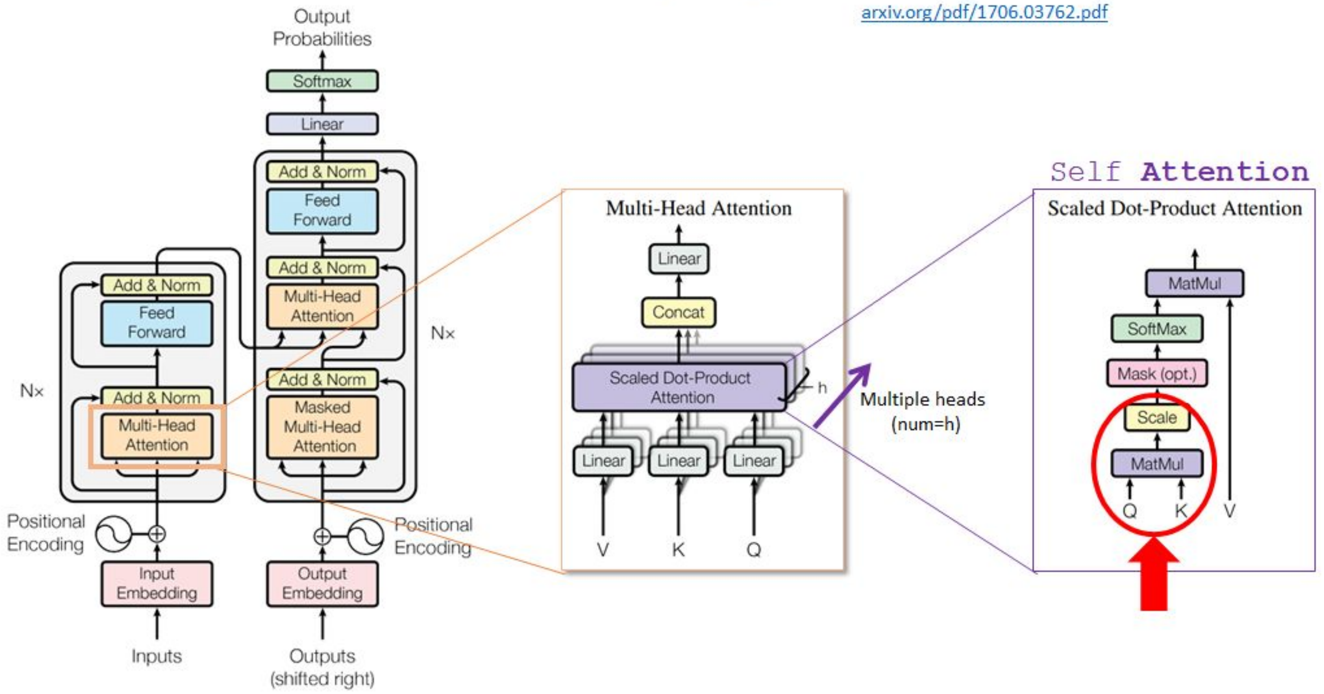


Figure 1: The Transformer - model architecture.

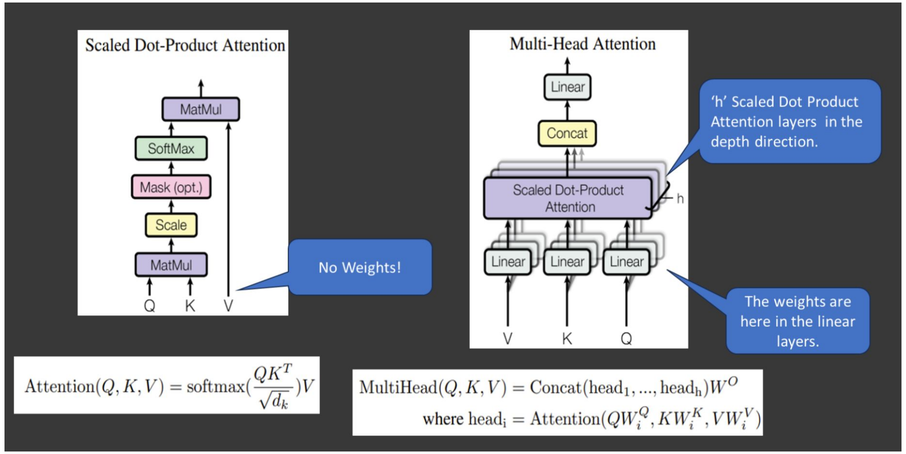
Multi-head attention

Several such attention heads are applied in parallel to an input sequence, making it a multi-head attention layer

This process, known as self-attention, happens separately in each of these smaller stages or 'heads'. Each head works its magic independently, conjuring up an output vector.

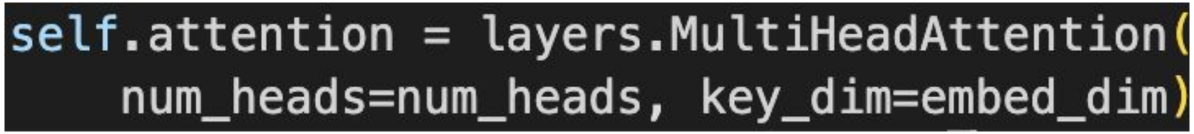
This ensemble passes through a final linear layer, much like a filter that fine-tunes their collective performance. The beauty here lies in the diversity of learning across each head, enriching the encoder model with a robust and multifaceted understanding.

As we can learn many such weight for query, key and value vectors allowing us to learn different facets and generating various context aware vectors which are concatenated at end


Scaled Dot-Product Attention
MatMul
SoftMax
Mask (opt.)
Scale
MatMul
Q
K
V
No Weights!
Multi-Head Attention
Linear
Concat
Scaled Dot-Product Attention
h
Linear
Linear
Linear
V
K
Q
'h' Scaled Dot Product Attention layers in the depth direction.
The weights are here in the linear layers.

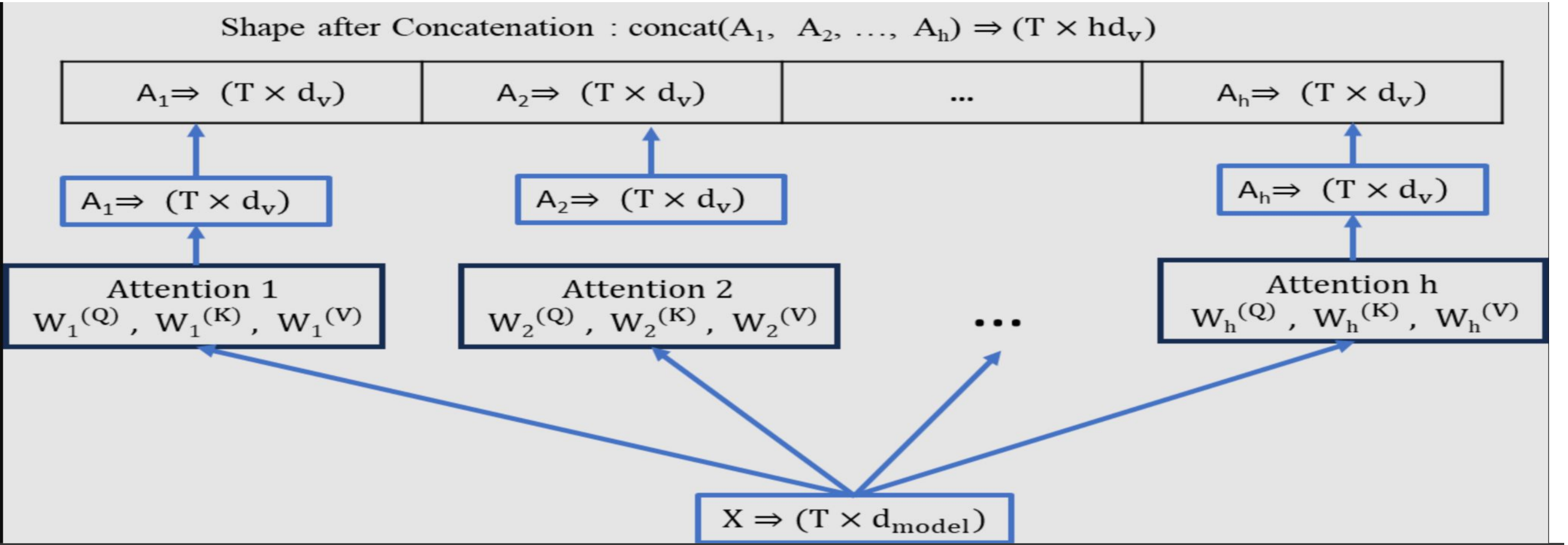
$$\mathrm{Attention}(Q,K,V) = \mathrm{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

$$\mathrm{MultiHead}(Q,K,V) = \mathrm{Concat}(\mathrm{head_1},...,\mathrm{head_h})W^O$$
$$\text{where } \mathrm{head_i} = \mathrm{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$



```
self.attention = layers.MultiHeadAttention(
    num_heads=num_heads, key_dim=embed_dim)
```

Final computation



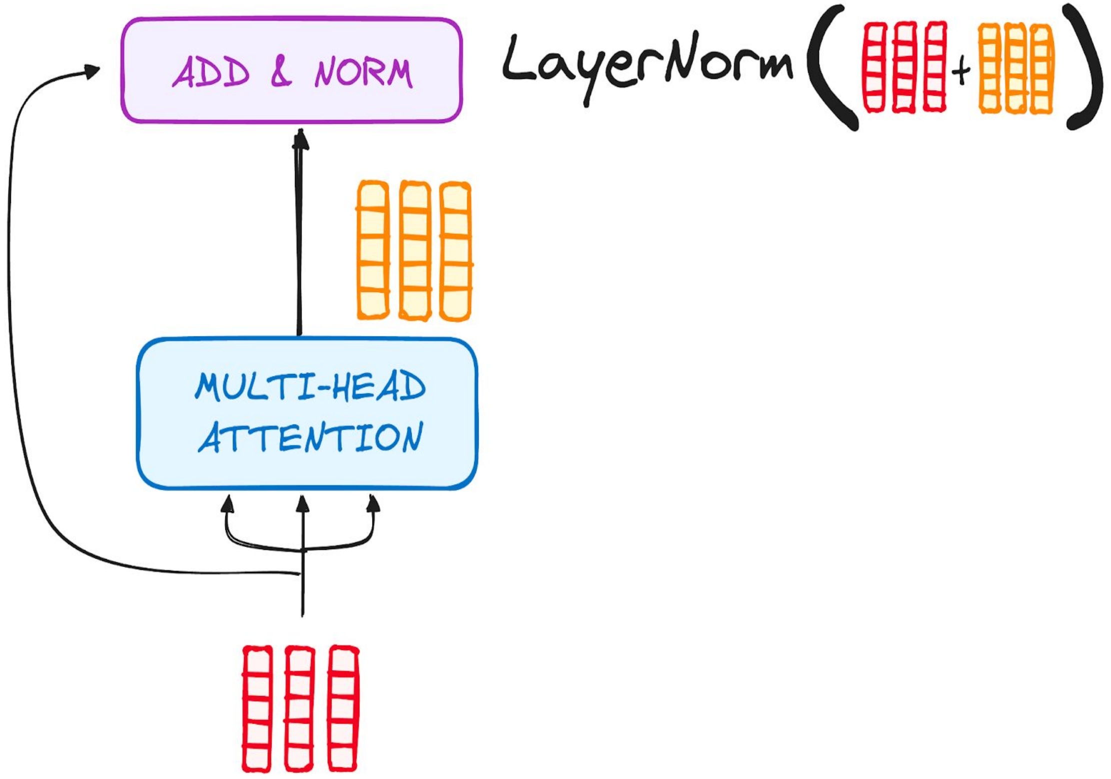
Final Projection : $Output = concat(A_1, A_2, ..., A_h)W^o$

Shape of : $concat(A_1, A_2, ..., A_h) \Rightarrow (T \times hd_v)$

Shape of: $W^o \Rightarrow (hd_v \times d_{model})$

Shape of final: $Ouput = concat(A_1, A_2, ..., A_h)W^o \Rightarrow (T \times hd_v) \times (hd_v \times d_{model}) \Rightarrow (T \times d_{model}) \Leftarrow$ **Back to the initial input shape.**

Layer norm and residual connections



Each sub-layer in an encoder layer is followed by a normalization step. Also, each sub-layer output is added to its input (residual connection) to help mitigate the vanishing gradient problem, allowing deeper models.

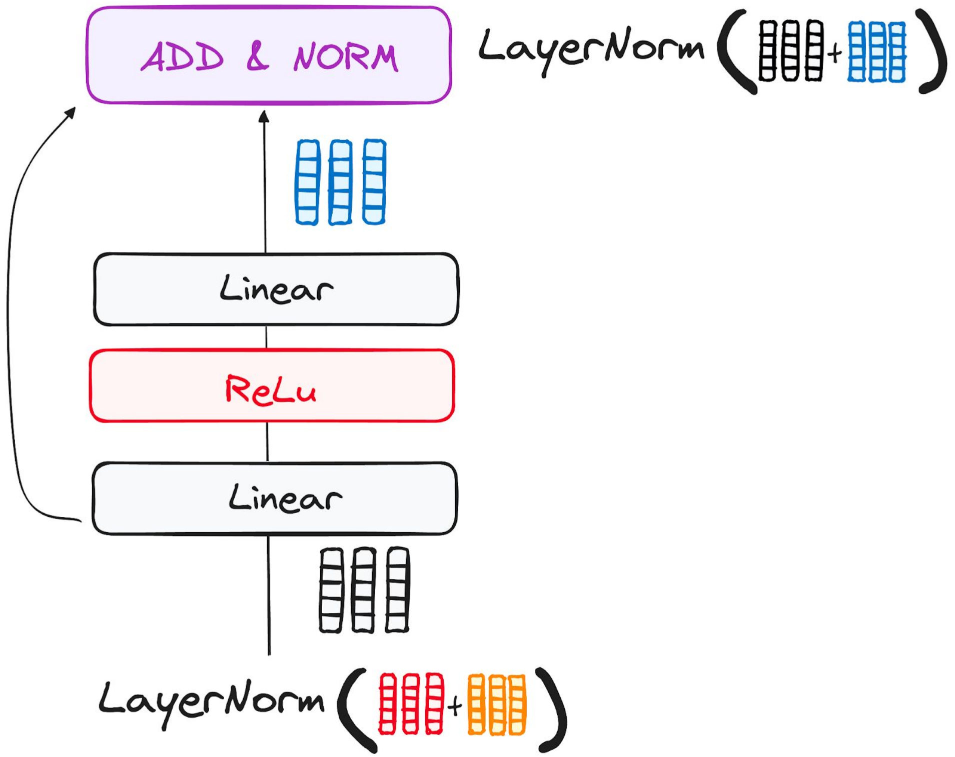
Feedforward layers and further

The journey of the normalized residual output continues as it navigates through a pointwise feed-forward network, a crucial phase for additional refinement.

Picture this network as a duo of linear layers, with a ReLU activation nestled in between them, acting as a bridge. Once processed, the output embarks on a familiar path: it loops back and merges with the input of the pointwise feed-forward network.

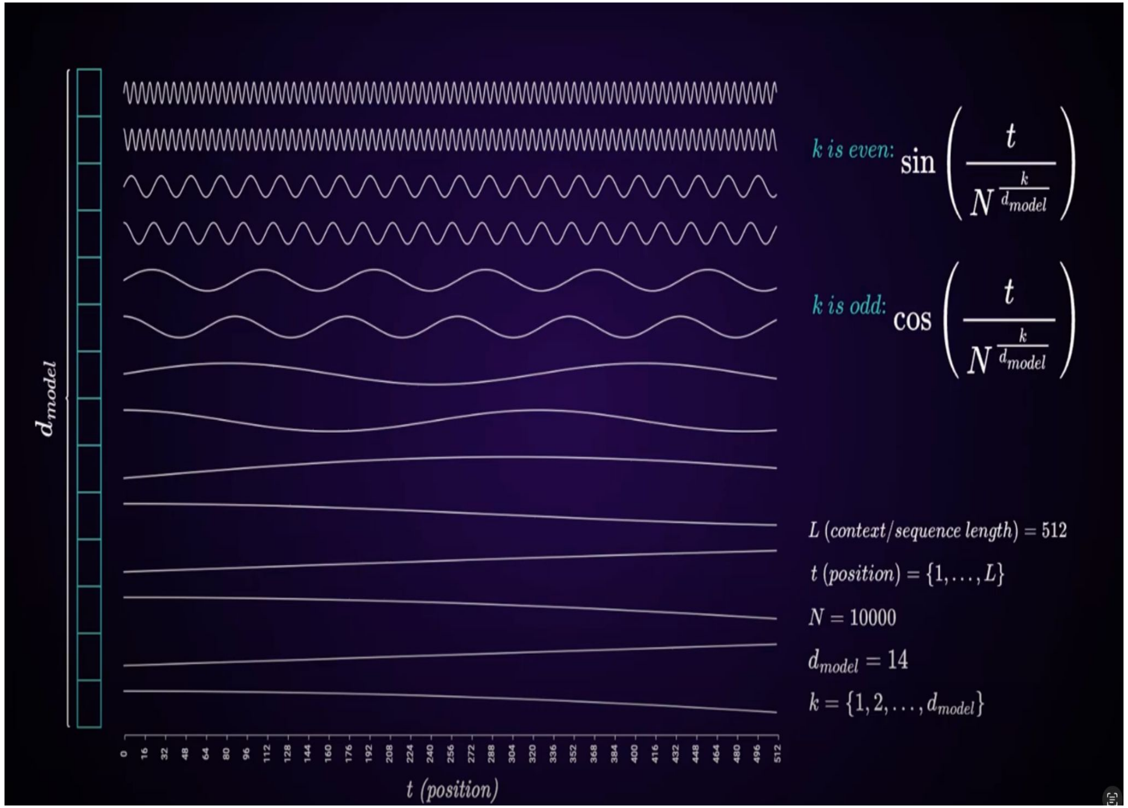
This reunion is followed by another round of normalization, ensuring everything is well-adjusted and in sync for the next steps.

This completes encoder architecture. We can have multiple such encoder layers stacked on top of one other and finally a dense layer as classification head



What is missing ????

Positional encoding


d_{model}
k is even: $\sin\left(\frac{t}{N^{\frac{k}{d_{model}}}}\right)$
k is odd: $\cos\left(\frac{t}{N^{\frac{k}{d_{model}}}}\right)$
L (context/sequence length) $= 512$
t (position) $= \{1, \dots, L\}$
$N = 10000$
$d_{model} = 14$
$k = \{1, 2, \dots, d_{model}\}$
0 16 32 48 64 80 96 112 128 144 160 176 192 208 224 240 256 272 288 304 320 336 352 368 384 400 416 432 448 464 480 496 512
t (position)

1. Transformers self attention doesn't have any notion of positions. Even if u scramble the whole sequence the output of attention would stay the same
2. So to introduce the positional configuration the original transformers paper used positional embedding along with token embeddings.
3. The size of positional embeddings is same to token embeddings and they are added together as final embedding before feeding to self attention
4. These are non-trainable ones but in our demo we would use the embedding similar to word embedding for position by passing where instead of passing vectorised tokens for word we pass a range of (0,no_of_tokens) in input signal to get the embeddings