



# Transformers

## Glossary:

Language Modeling:

Vectorizing your code

Positional Encoding

self-attention

Multi-Head Attention

Context aware Word Vectors

## Deep Learning with Python, 2nd edition (11.4)

### The Transformer Archi:

In simple terms transformer architecture leverages **neural attention** which is devoid of any recurrent or convolutional layers.

#### 11.4.1 Understanding self-attention

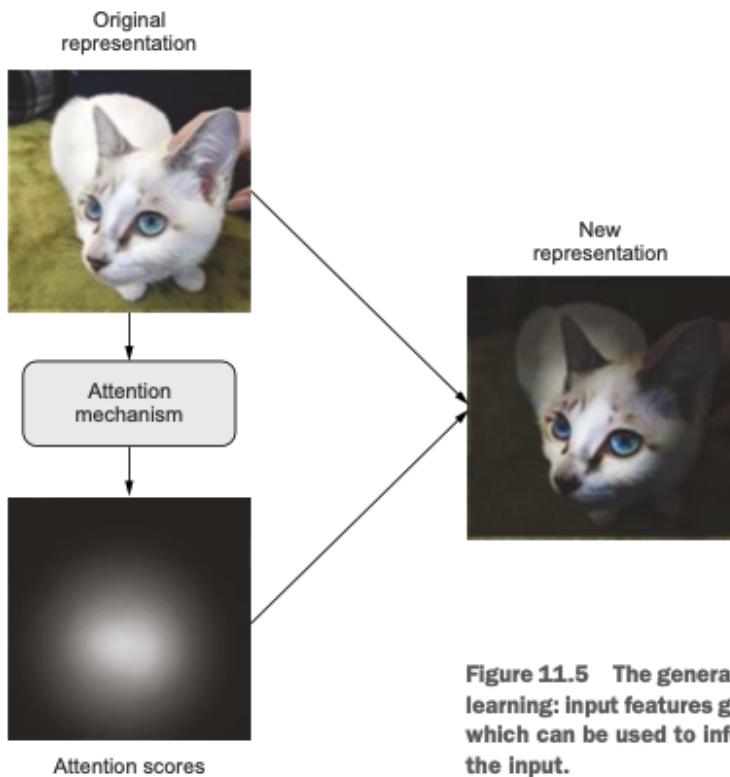
When you read something/want to learn something first thing you do is skim through the content to understand the relevant bits. You devote attention to some parts depending on your interests/goals.

- Think about the **Max Pooling layer** which pools features. That's an "**all or nothing**" form of attention keep the most important feature and discard the rest.

- **TF-IDF normalization** assigns importance scores to tokens based on how much information different tokens are likely to carry. Important tokens get boosted while irrelevant tokens get faded out. That's a **continuous form of attention**.

Humans also follow the attention mechanism in some sense, when we look at an image we don't give attention to every detail present in the image. We look for some familiar features and pay attention to those details. For example: When you look at a picture of a cat, you may tend to look at the pointy ears, whiskers, the nose and eyes which enable us to classify the picture as a cat.

**Illustration** in the image below:



**Figure 11.5** The general concept of “attention” in deep learning: input features get assigned “attention scores,” which can be used to inform the next representation of the input.

Then an interesting question is,

**How do you determine which features needs the most/ least attention?**

As a simple solution you can score your features with relevant features getting higher scores and the less relevant features being assigned lesser scores.

Now this leads to the next questions:

**How these scores should be computed? and  
What you should do with these scores ?**

Ans: It will vary from approach to approach. The crux of the idea is to make features **content-aware**.

Vector spaces that capture the “shape” of the semantic relationships between different words.

In an embedding space, a single word has a fixed position-a fixed set of relationships with every other word in the space.

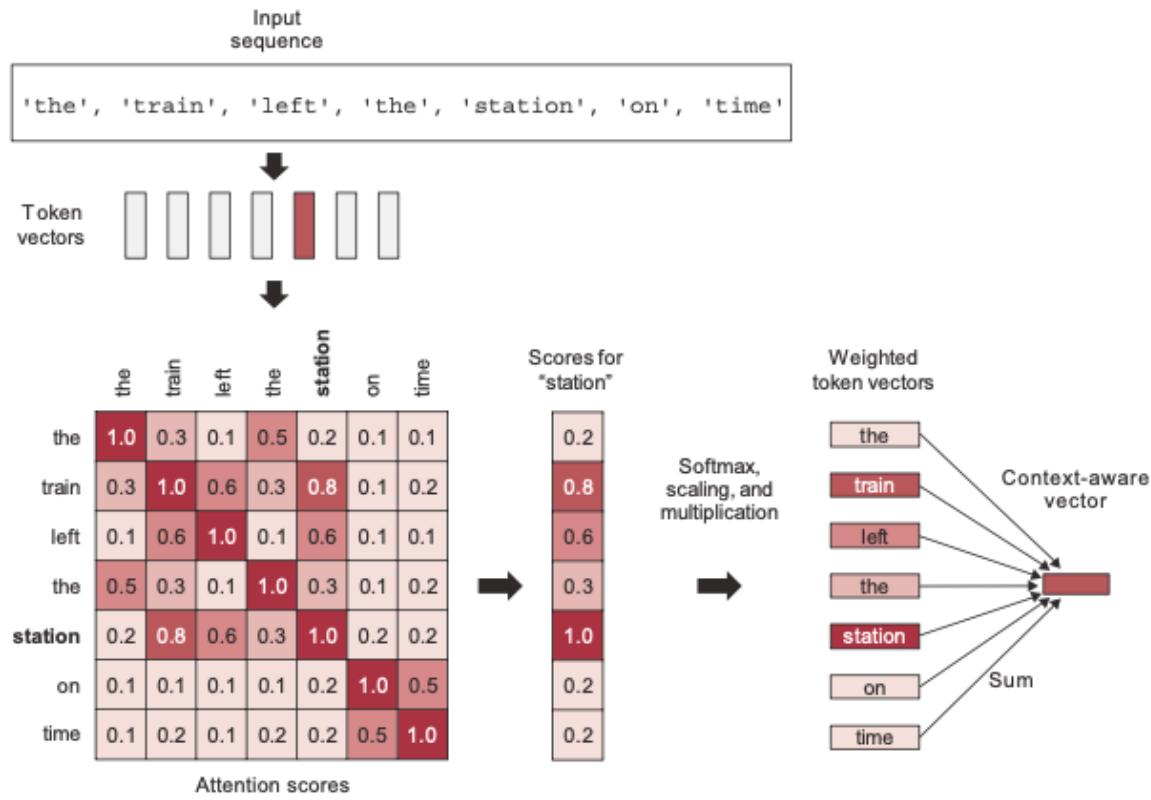
But language doesn’t work like that, the meaning of a word is usually context-specific.

Ex:- *When you mark the date, you’re not talking about the same “date” as when you go on a date, nor is it the kind of date you’d buy at the market. When you say, “I’ll see you soon,” the meaning of the word “see” is subtly different from the “see” in “I’ll see this project to its end” or “I see what you mean.” And, of course, the meaning of pronouns like “he,” “it,” “in,” etc., is entirely sentence-specific and can even change multiple times within a single sentence.*

Therefore we need vector representation of words depending on the words that surround it. That’s where self-attention comes in.

The purpose of **self-attention** is to modulate the representation of a token by using the representations of related tokens in the sequence. This produces **context aware token representations**.

Consider an example sentence: “**The train left the station on time.**” Now, consider one word in the sentence: station. What kind of station are we talking about? Could it be a radio station? Maybe the International Space Station? Let’s figure it out algorithmically via self-attention (see figure 11.6).



**Figure 11.6 Self-attention:** attention scores are computed between “station” and every other word in the sequence, and they are then used to weight a sum of word vectors that becomes the new “station” vector.

### Algorithm Steps:

1. Compute relevancy score (cosine -used because computationally very efficient distance function-) between the vector for “station” and every other vector in the sentence. These scores are our **attention scores**. Later these scores will go through a scaling function and a softmax.
2. Then compute the sum of all word vectors in the sentence, weighted by the relevancy score. Words closely related to “station” will contribute more to the sum. Then the resulting vector is our new representation for the word “station”.

Then repeat this process for every word in the sentence producing a new sequence of vectors encoding the sentence.

At this point I have a few questions:

1. How are the initial vector embeddings calculated ? (Mostly reuse existing word vectors)
2. When does the vector calculation stop ? After reading each sentence or the whole corpus? (Still Unsure)

**Please read this before resuming again:** <https://towardsdatascience.com/illustrated-self-attention-2d627e33b20a>

Keras has a built-in layer to handle attention: the **MultiHeadAttention** layer. Here's how you would use it:

```
num_heads = 4
embed_dim = 256
mha_layer = MultiHeadAttention(num_heads=num_heads, key_dim=embed_dim)
outputs = mha_layer(inputs, inputs, inputs)
```

Reading this, you're probably wondering

- Why are we passing the inputs to the layer *three times*? That seems redundant.
- What are these “multiple heads” we’re referring to? That sounds intimidating do they also grow back if you cut them?

Both of these questions have simple answers. Let's take a look.

## Generalized Self-Attention.

**Note:** Original transformer was a translation machine.

A Transformer is a **sequence-to-sequence** model: it was designed to convert one sequence into another.

Schematically self-attention looks something like this:

```
outputs = sum (inputs(C) * pairwise_score(inputs(A), inputs(B)))
```

Meaning: For each token in inputs (A), compute how much the token is related to every token in inputs (B), and use these scores to weight a sum of tokens from inputs (C).

The terms Query, Keys and Values come from the Information Retrieval domain (ex: search engines)

A → Query ; B → Keys; C → Values

The operation becomes “for each element in the query, compute how much the element is related to every key, and use these scores to weight a sum of values”

### **Self-Attention Defintion:**

Self Attention, also called intra Attention, is an attention mechanism relating different positions of a single sequence in order to compute a representation of the same sequence. It has been shown to be very useful in machine reading, abstractive summarization, or image description generation.

Self-attention, also known as **intra-attention**, is an attention mechanism relating different positions of a single sequence in order to compute a representation of the same sequence. It has been shown to be very useful in machine reading, abstractive summarization, or image description generation.

### **Multi-head attention:**

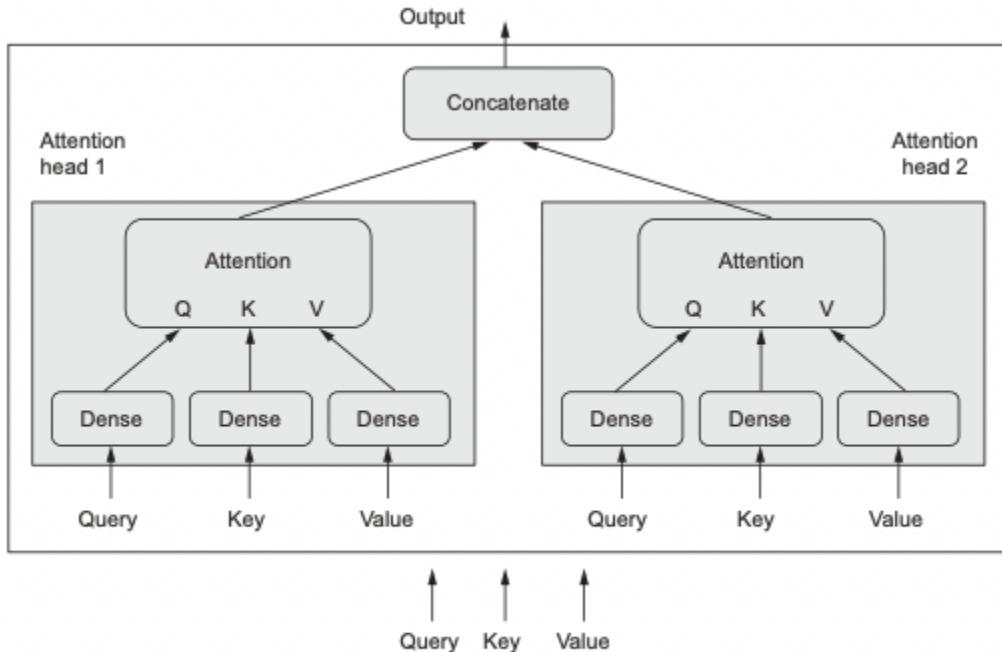


Figure 11.8 The MultiHeadAttention layer

The **multi-head** moniker refers to the fact that the output space of the self-attention layer gets factored into a set of independent subspaces, learned separately: the initial query, key, and value are sent through three independent sets of dense projections, resulting in three separate vectors. Each vector is processed via neural attention, and the three outputs are concatenated back together into a single output sequence. Each such subspace is called a **head**.

**The presence of the learnable dense projections enables the layer to actually learn something**, as opposed to being a purely stateless transformation that would require additional layers before or after it to be useful.

In addition, **having independent heads helps** the layer learn **different groups of features for each token**, where features within one group are **correlated** with each other but are mostly independent from features in a different group.

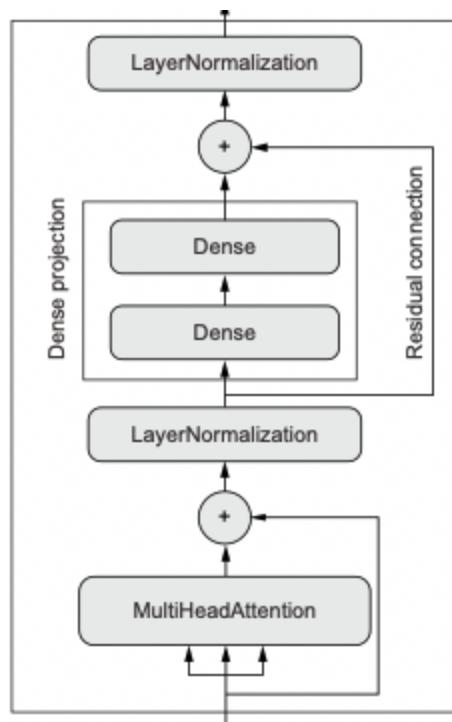
## The Transformer Encoder:

NOTE:

**Residual-Connections:** These are required so that the information is not destroyed in a deep neural network

**Normalization-Layer:** are supposed to help gradients flow better during back-propagation

So, finally with these ideas combined we have a transformer encoder head:



**Figure 11.9** The TransformerEncoder chains a MultiHeadAttention layer with a dense projection and adds normalization as well as residual connections.

**IMPORTANT:** The encoder part can be used for text classification, it's a very generic module that ingests a sequence and learns to turn it into a more useful **representation**.

Accompanying code for the above mentioned diagram:

NOTE: the notebook [Transformer\\_Encoder\\_Keras.ipynb](#) implements and executes this code. It has the high level details of the Transformer architecture which is necessary when implementing.

```
import tensorflow as tf
from tensorflow import keras
```

```

from tensorflow.keras import layers

class TransformerEncoder(layers.Layer):
    def __init__(self, embed_dim, dense_dim, num_heads, **kwargs):
        super().__init__(**kwargs)
        self.embed_dim = embed_dim
        self.dense_dim = dense_dim
        self.num_heads = num_heads
        self.attention = layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=embed_dim)
        self.dense_proj = keras.Sequential(
            [layers.Dense(dense_dim, activation="relu"),
             layers.Dense(embed_dim),]
        )
        self.layernorm_1 = layers.LayerNormalization()
        self.layernorm_2 = layers.LayerNormalization()

    def call(self, inputs, mask=None):
        if mask is not None:
            mask = mask[:, tf.newaxis, :]
        # Multihead Attention Layer
        attention_output = self.attention(inputs, inputs, attention_mask=mask)
        # First normalization Layer
        proj_input = self.layernorm_1(inputs + attention_output)
        # Dense Layer
        proj_output = self.dense_proj(proj_input)
        # Residual and Normalization layer 2
        return self.layernorm_2(proj_input + proj_output)

    def get_config(self):
        config = super().get_config()
        config.update({
            "embed_dim": self.embed_dim,
            "num_heads": self.num_heads,
            "dense_dim": self.dense_dim,
        })
        return config

# Using Trantformers for Text Classification
vocab_size = 20000
embed_dim = 256
num_heads = 2
dense_dim = 32

inputs = keras.Input(shape=(None,), dtype="int64")
x = layers.Embedding(vocab_size, embed_dim)(inputs)
x = TransformerEncoder(embed_dim, dense_dim, num_heads)(x)
x = layers.GlobalMaxPooling1D()(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])

```

```
model.summary()

callbacks = [
    keras.callbacks.ModelCheckpoint("transformer_encoder.keras",
                                    save_best_only=True)
]

model.fit(int_train_ds, validation_data=int_val_ds, epochs=20, callbacks=callbacks)
model = keras.models.load_model(
    "transformer_encoder.keras",
    custom_objects={"TransformerEncoder": TransformerEncoder})
print(f"Test acc: {model.evaluate(int_test_ds)[1]:.3f}")
```

Next to learn visually about Transformers and Attention, I went through **RASA WhiteBoard** Videos on Attention and Transformers: Link to [video](#) (there are 4 videos in total)

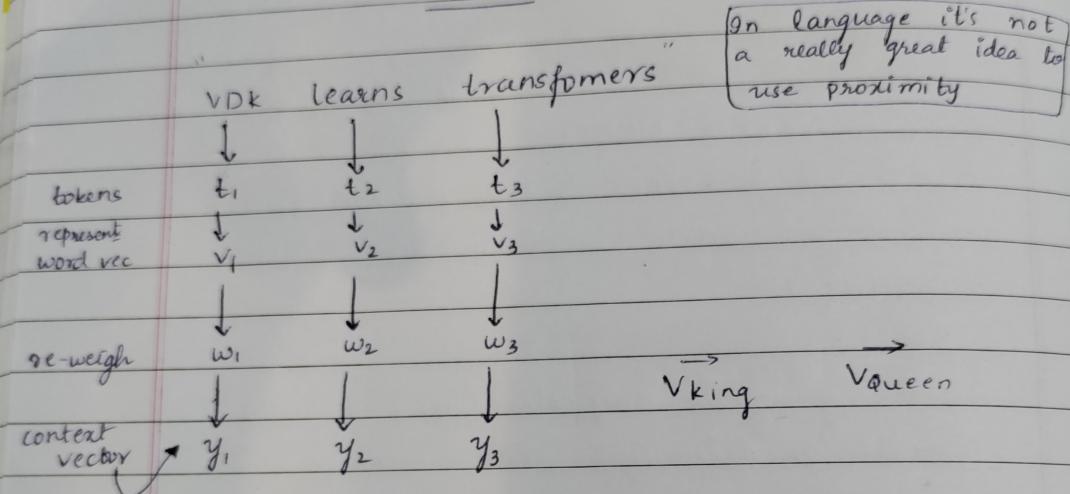
# Transformers

DATE:

PAGE:

RASA - Algo - Transformers & Attention 1: Self-Attention

## Attention - Mechanism



## Reweighting - Plan

$$w_{kq} = V_k \cdot V_q \quad (\text{cosine similarity})$$

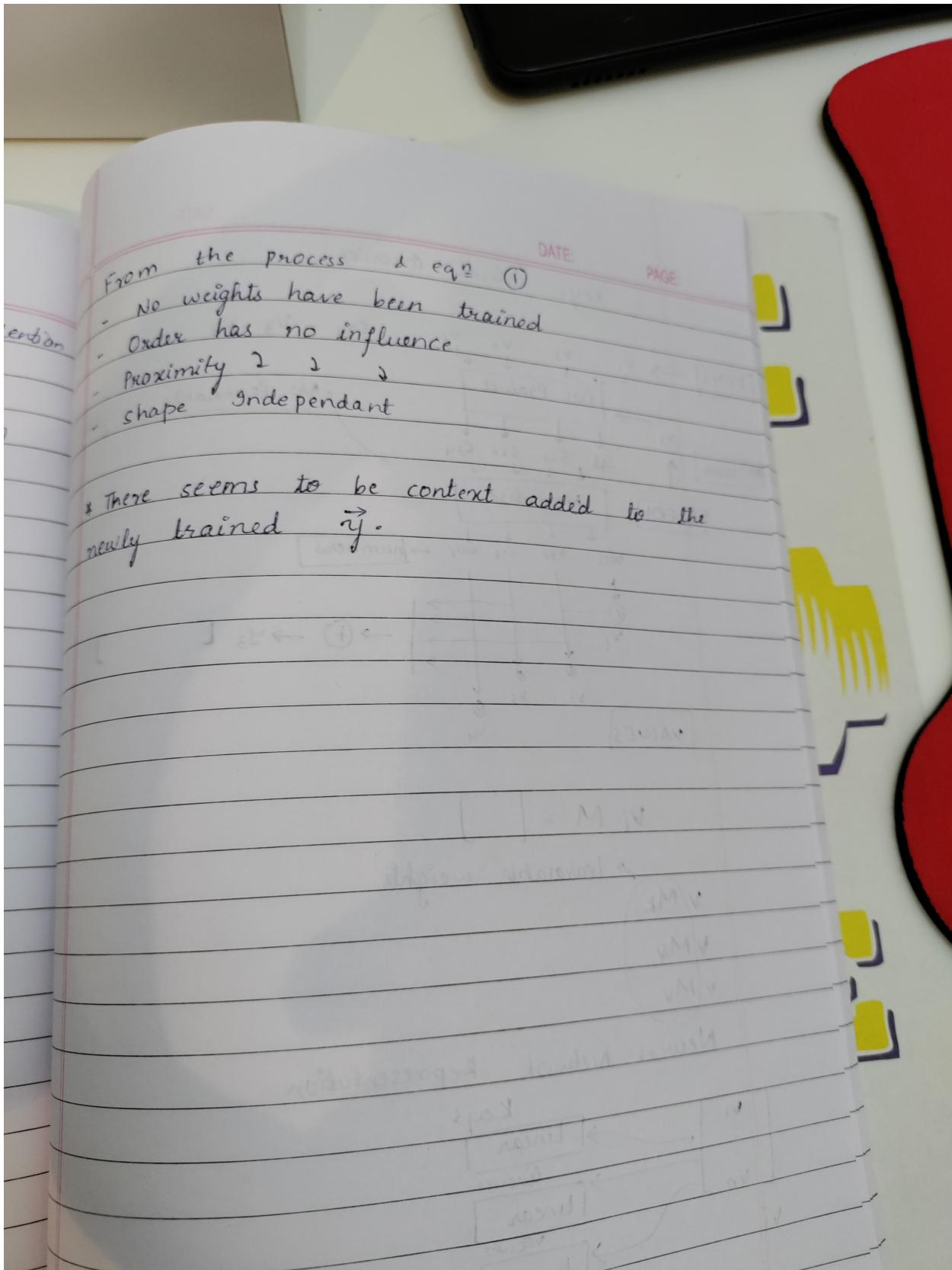
				Consider $V_1$
Bank of the given.	$v_1$	$v_2$	$v_3$	$V_1 \cdot V_1 = w_{11}$ $w_{11}$
RE-WEIGHT				$V_1 \cdot V_2 = w_{12} \rightarrow w_{12}$
PROCESS	$y_1$	$y_2$	$y_3$	$V_1 \cdot V_3 = w_{13}$ Normalize $w_{13}$ $w_{13}$
				$V_1 \cdot V_4 = w_{14}$ $w_{14}$ $w_{14}$

$$\therefore y_1 = w_{11}v_1 + w_{12}v_2 + w_{13}v_3 + w_{14}v_4 \quad - (1)$$

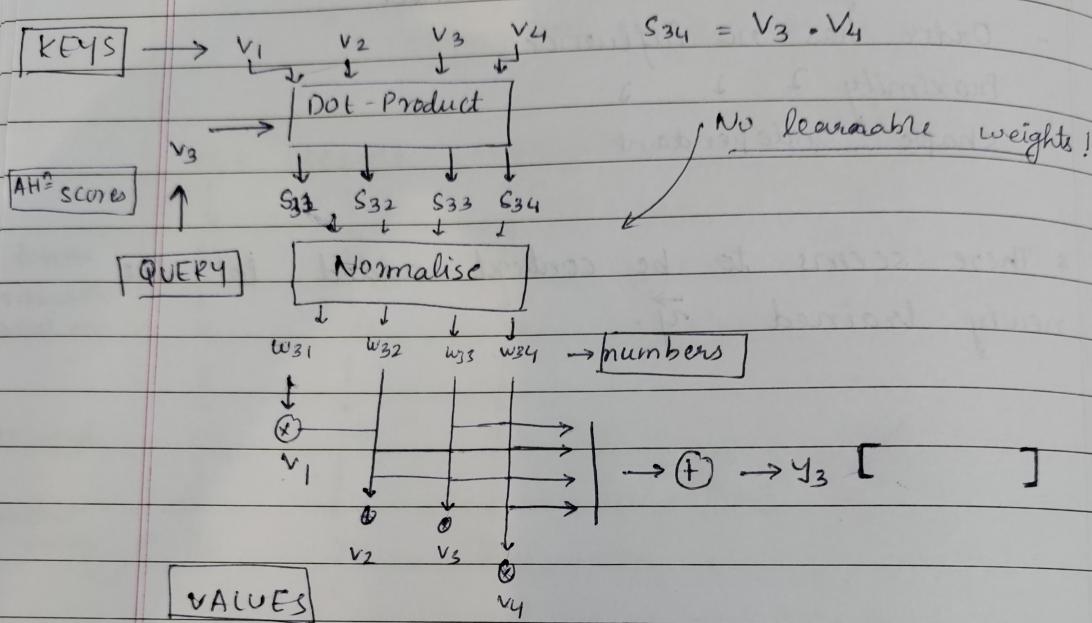
From the process & eqn ①

- No weights have been trained
- Order has no influence
- Proximity  $\rightarrow$  shape  $\rightarrow$  independent

\* There seems to be context added to the newly trained  $\vec{y}$ .



2 : Keys, values, Queries

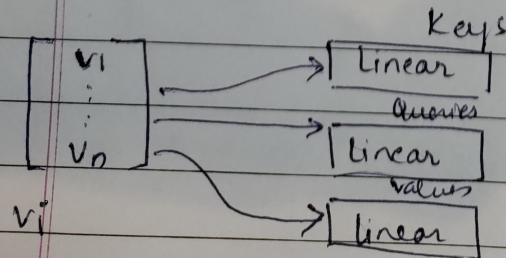


$$v_i M = [ ]$$

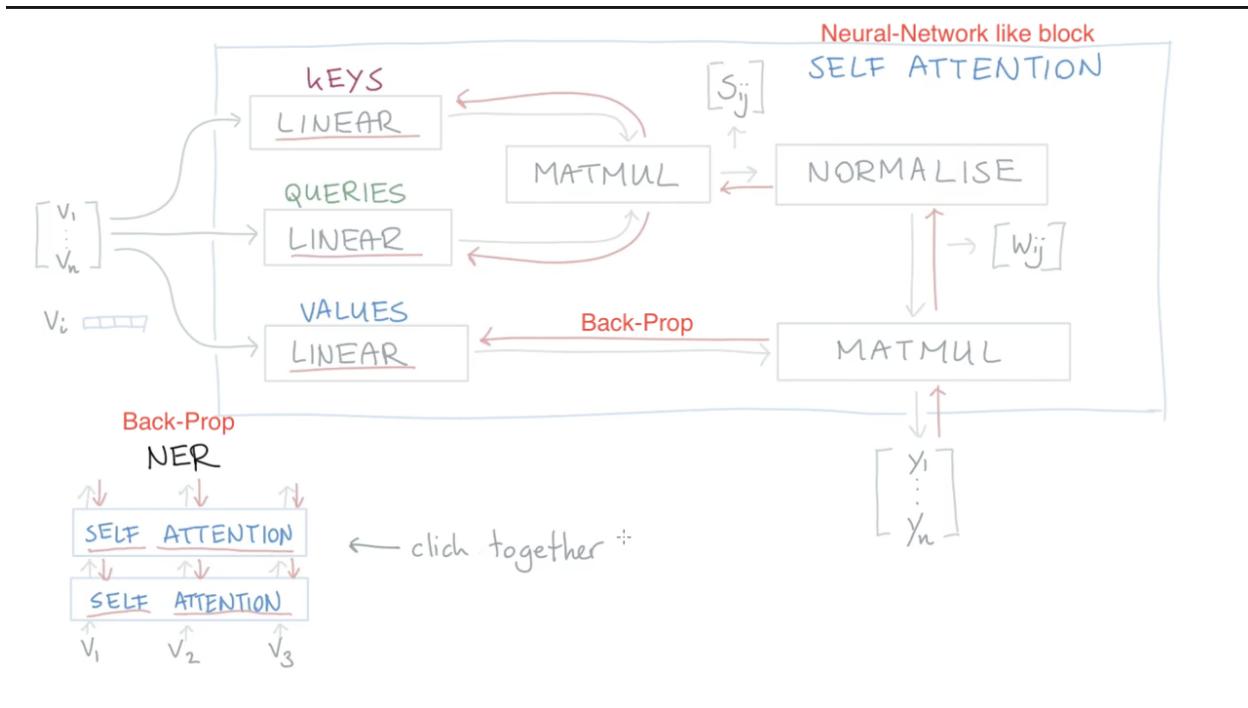
→ learnable weights.

$v M_K$   
 $v M_Q$   
 $v M_V$

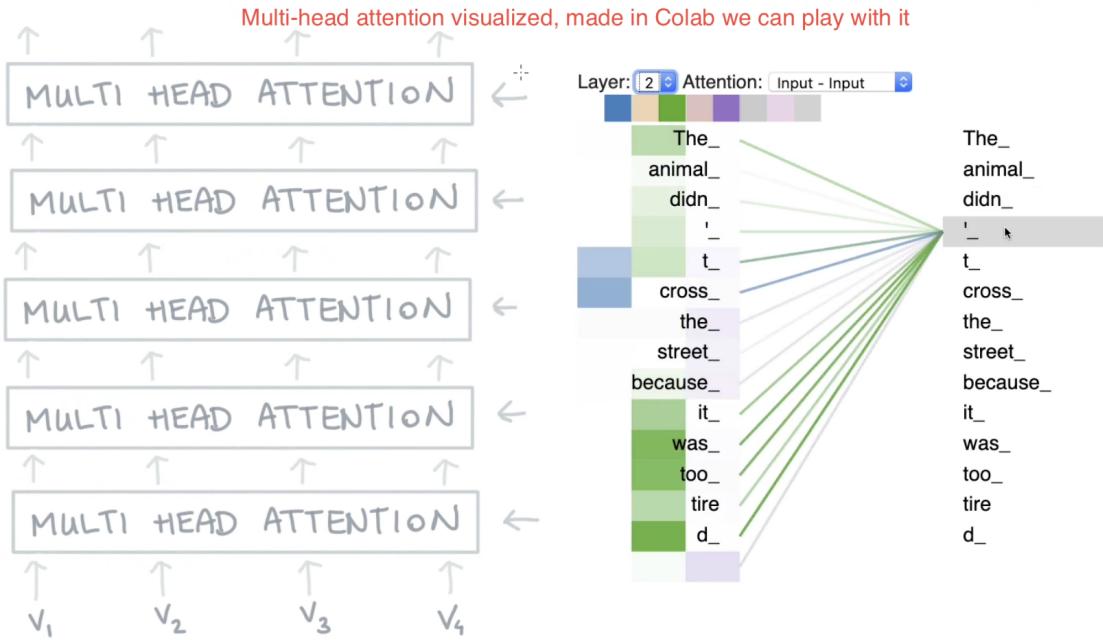
Neural-Network Representation



The following image shows how the Transformer block can fit like a neural-network block and shows how conceptually back-prop can be applied to get the context aware vectors (in image Named Entity Recognition NER is the task)



The following section is a note for self to checkout the google collab visual view of the attention.



## Self-Attention, Multi-Head Attention and the Transformer Architecture in one block

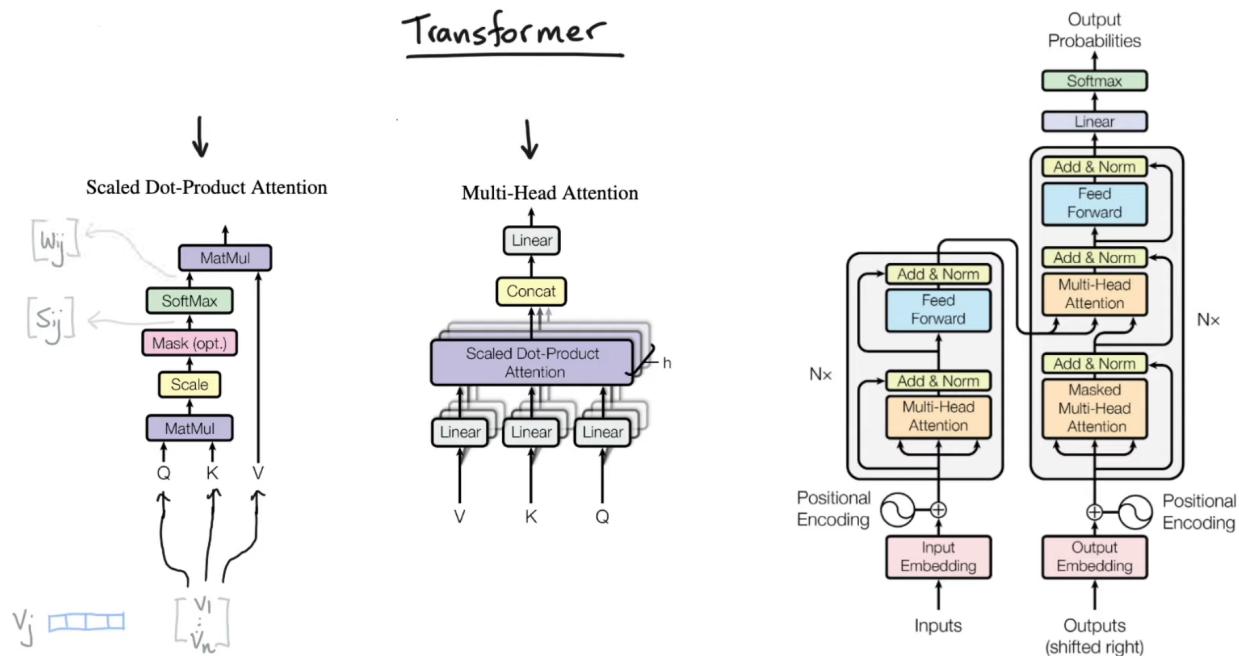
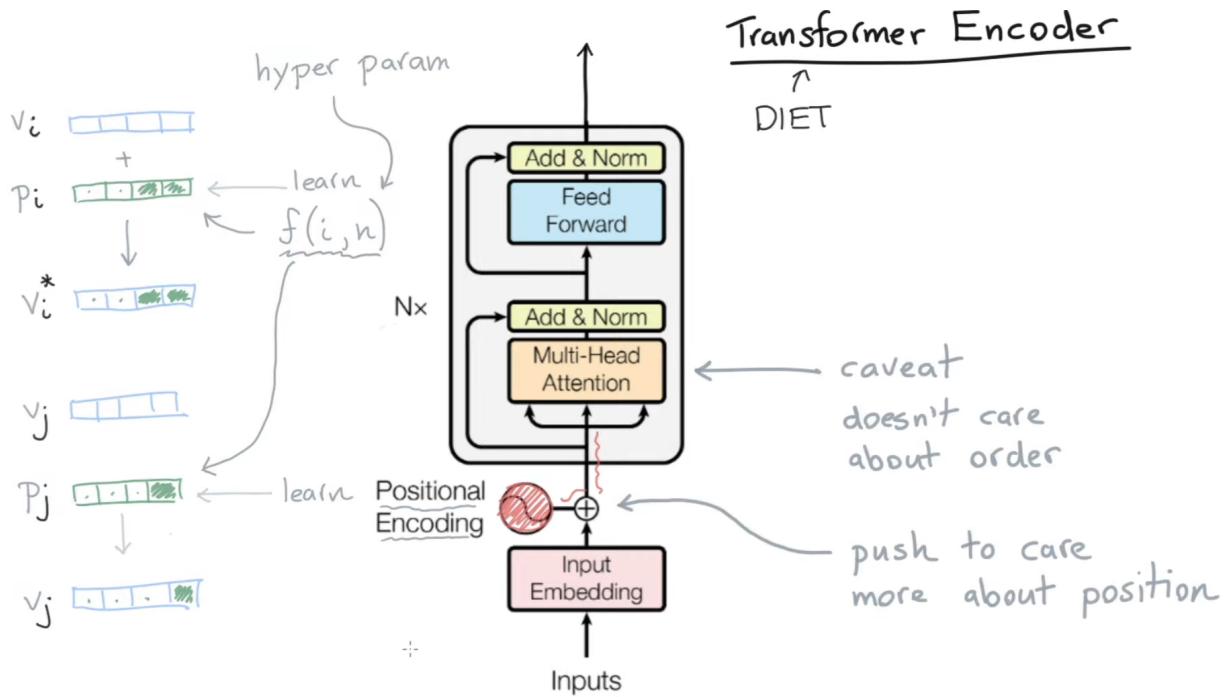


Figure 1: The Transformer - model architecture.

Currently focusing on Transformer Encoder block.



## Positional Embeddings:

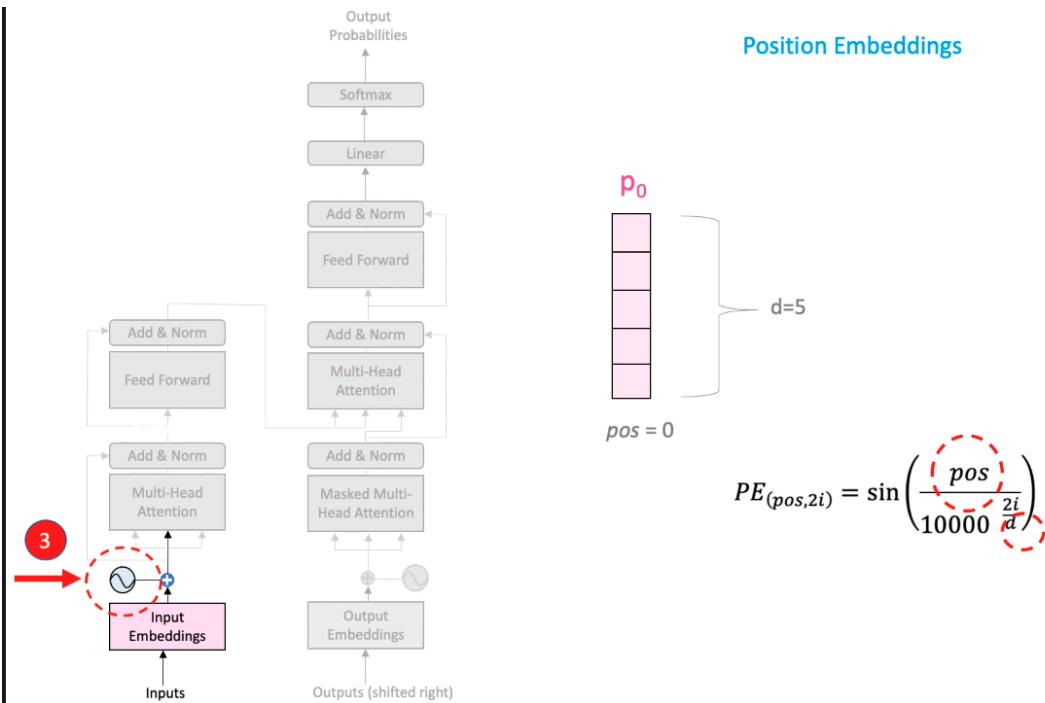
They contain information about the word order.

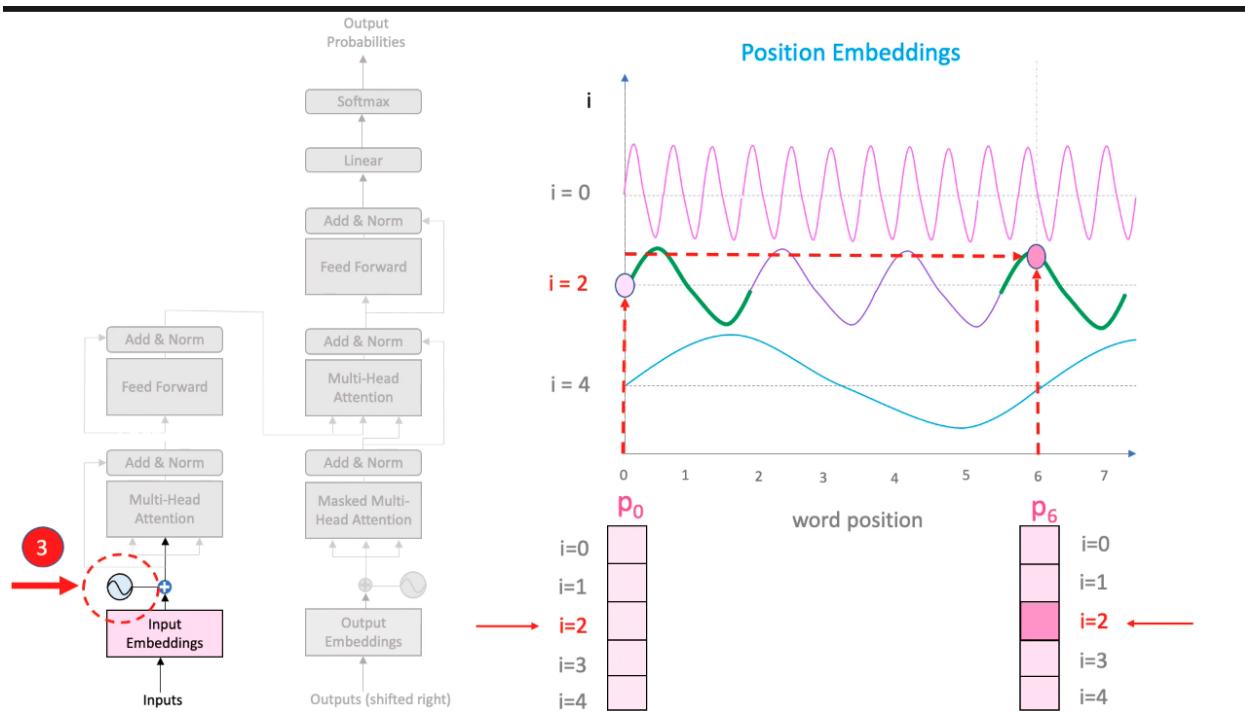
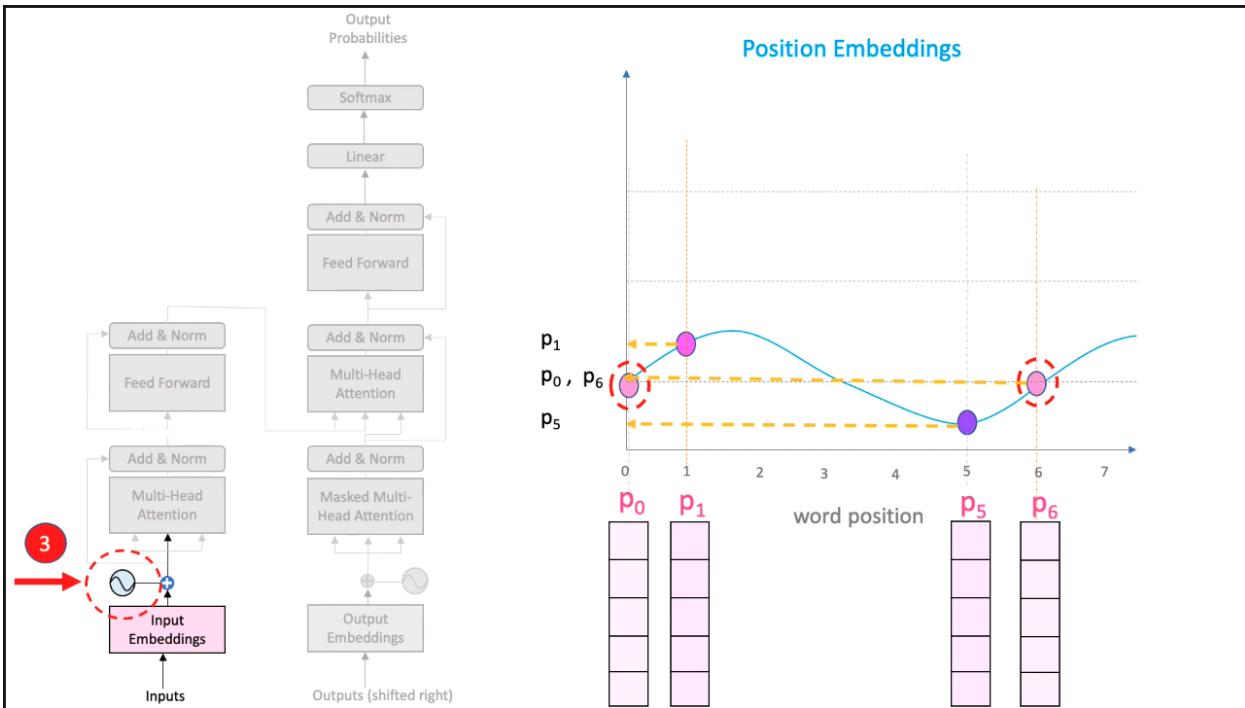
1. [Visual Guide to Transformer Neural Networks - \(Episode 1\) Position Embeddings](#)
2. [Transformer Architecture: The Positional Encoding \(Blog\)](#)
3. [The Annotated Transformer \(blog with paper and accompanying PyTorch code\)](#)

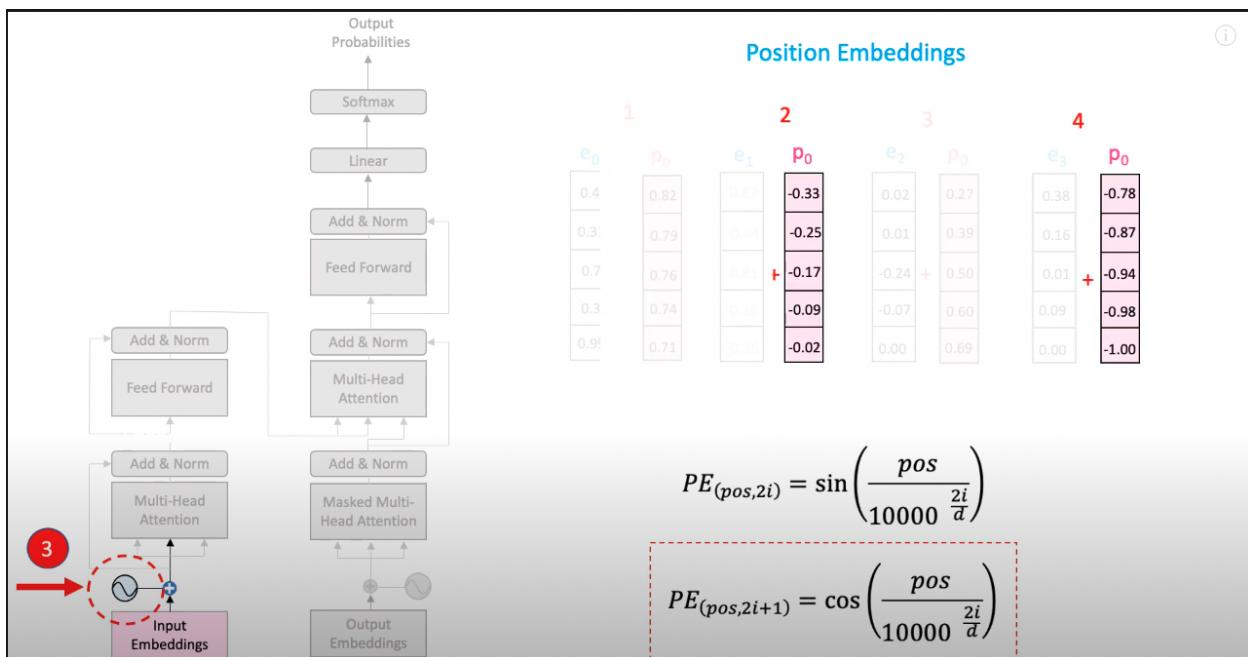
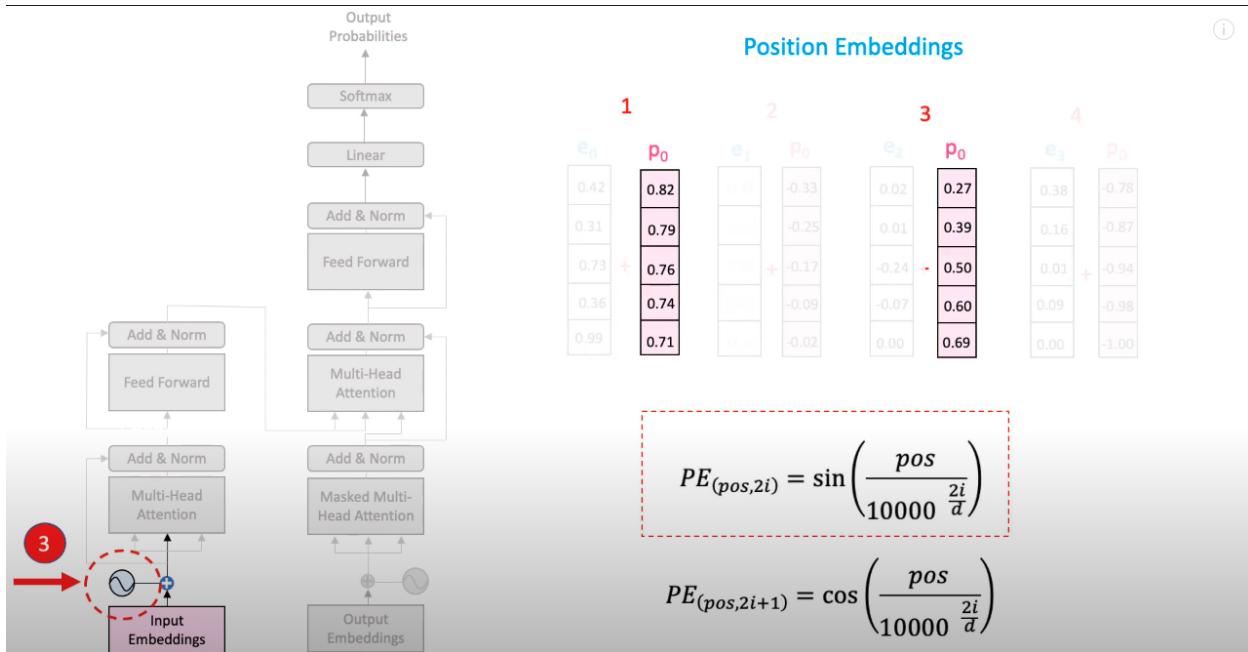
[Visual Guide to Transformer Neural Networks - \(Episode 1\) Position Embeddings](#) Notes

The next series of images show how positional embeddings were achieved, the images build on simple concepts and illustrate how sin and cosine functions help in creating positional embedding (word vectors which carry the position information)

[The Transformer\\_Encoder\\_Keras.ipynb](#) implements a similar form of positional embeddings.







More notes on better understanding positional encoding and getting better with its theory.

## Transformer Architecture: The Positional Encoding (Blog)

Ideal criteria for positional encodings are as follows:

- It should output a unique encoding for each time-step (word's position in a sentence)
- Distance between any two time-steps should be consistent across sentences with different lengths.
- Our model should generalize to longer sentences without any efforts. Its values should be bounded.
- It must be deterministic.

Let  $t$  be the desired position in an input sentence,  $\vec{p}_t \in \mathbb{R}^d$  be its corresponding encoding, and  $d$  be the encoding dimension (where  $d \equiv_2 0$ ) Then  $f : \mathbb{N} \rightarrow \mathbb{R}^d$  will be the function that produces the output vector  $\vec{p}_t$  and it is defined as follows:

$$\vec{p}_t^{(i)} = f(t)^{(i)} := \begin{cases} \sin(\omega_k \cdot t), & \text{if } i = 2k \\ \cos(\omega_k \cdot t), & \text{if } i = 2k + 1 \end{cases}$$

where

$$\omega_k = \frac{1}{10000^{2k/d}}$$

## Other details

Earlier in this post, I mentioned that positional embeddings are used to equip the input words with their positional information. But how is it done? In fact, the original paper added the positional encoding on top of the actual embeddings. That is for every word  $w_t$  in a sentence  $[w_1, \dots, w_n]$ , Calculating the correspondent embedding which is fed to the model is as follows:

$$\psi'(w_t) = \psi(w_t) + \vec{p}_t$$

To make this summation possible, we keep the positional embedding's dimension equal to the word embeddings' dimension i.e.

$$d_{\text{word embedding}} = d_{\text{positional embedding}}$$

## Relative Positioning

Another characteristic of sinusoidal positional encoding is that it allows the model to attend relative positions effortlessly. Here is a quote from the original paper:

We chose this function because we hypothesized it would allow the model to easily learn to attend by relative positions, since for any fixed offset  $k$ ,  $\text{PE}_{\text{pos}+k}$  can be represented as a linear function of  $\text{PE}_{\text{pos}}$ .

### FAQ:

1. **Why positional embeddings are summed with word embeddings instead of concatenation?**
2. **Doesn't the position information get vanished once it reaches the upper layers?**
3. **Why are both sine and cosine used?**

## **USING POSITIONAL ENCODING TO RE-INJECT ORDER INFORMATION (DL Book pg 346)**

Here in the text-book positional embeddings are learnt in same way as the embedding vectors are learnt to embed word indices.

Ref: [Transformer\\_Encoder\\_Keras.ipynb](#)

### **Sequence-to-Sequence Learning:**

Seq-to-Sq Learning involves taking in a input sequence and producing out an output sequence. Some NLP seq-seq tasks are:

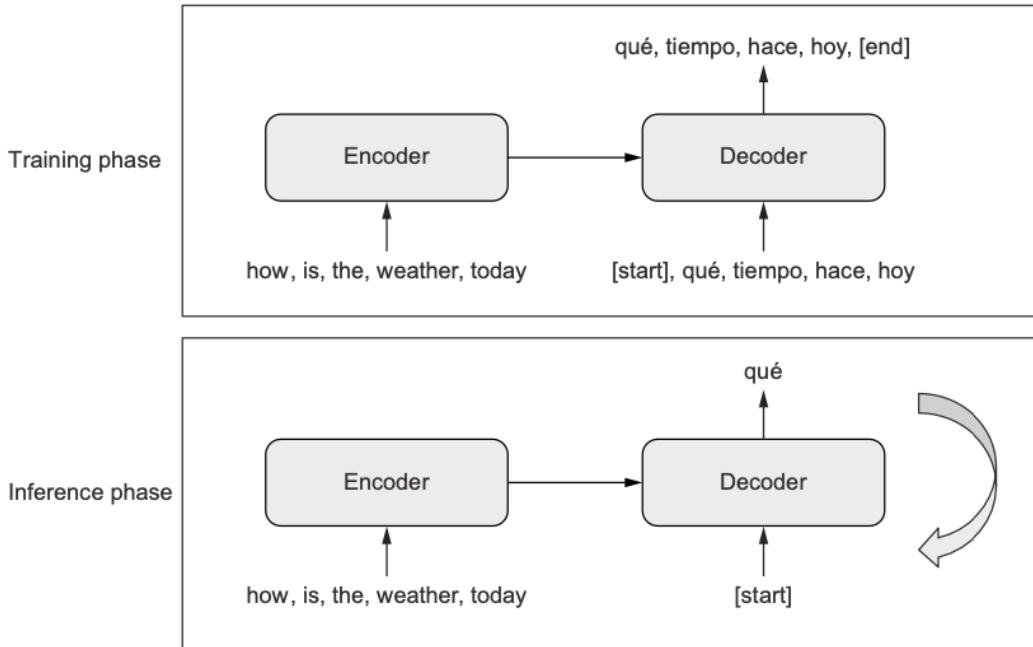
1. Machine Translation
2. Text Summarization
3. Question Answering
4. Chat-Bots
5. Text-Generation

### **General Template behind seq-seq models are:**

**Encoder** turns the input into an intermediate representation

**Decoder** is trained to predict the next token i by

1. looking at tokens (0 - i-1)
2. also looking at the encoded sequence



**Figure 11.12 Sequence-to-sequence learning:** the source sequence is processed by the encoder and is then sent to the decoder. The decoder looks at the target sequence so far and predicts the target sequence offset by one step in the future. During inference, we generate one target token at a time and feed it back into the decoder.

During inference, we don't have access to the target sequence we're trying to predict it from scratch. We'll have to generate it one token at a time:

1. We obtain the encoded source sequence from the encoder.
2. The decoder starts by looking at the encoded source sequence as well as an initial "seed" token (such as the string "[start]"), and uses them to predict the first real token in the sequence.
3. The predicted sequence so far is fed back into the decoder, which generates the next token, and so on, until it generates a stop token (such as the string "[end]").

## Pending Tasks before learning Transformers Decoder

1. Understand Machine Translation task and look at the dataset Eng - Span
2. Understand and practice Tensorflow TextVectorizer to convert to tf.dataset
3. Understand and implement the Decoder of the Transformer

## **Decoder-Block of Transformer:**

### **Next-Steps:**

To better the understanding on self-attention itself lets go through these videos: [Jan 5]

1. a. <https://www.youtube.com/watch?v=yGTUuEx3GkA>  
b. <https://www.youtube.com/watch?v=tIVKXrEDMhk>  
c. <https://www.youtube.com/watch?v=KmAISyVvE1Y>  
d. <https://www.youtube.com/watch?v=0PjHri8tc1e>
2. Next better your understanding on Multi Head attention  
a. <https://www.youtube.com/watch?v=A1eUVxscNq8>
3. Some other Transformer Video : [Jan 6]  
a. <https://www.youtube.com/watch?v=dichIcUZfOw>  
b. <https://www.youtube.com/watch?v=mMa2PmYJICe>
4. Stanford Lecture 9 - Self- Attention and Transformers **[Later, during revision]** [Jan 6]
5. Full Stack DL—> Transformers **[Later, during revision]** [Jan 6]
6. Go through DL FC text-book and implement Transformer Encoder [Jan 7]
7. Positional embedding in Keras.
8. Understand the sequence to sequence models.
9. Revise on weekend in detail for 1 hour
10. Understand the decoder part of the transformers.
11. Build better intuition for why self-attention, positional encoding works ?

