Transfomers

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:

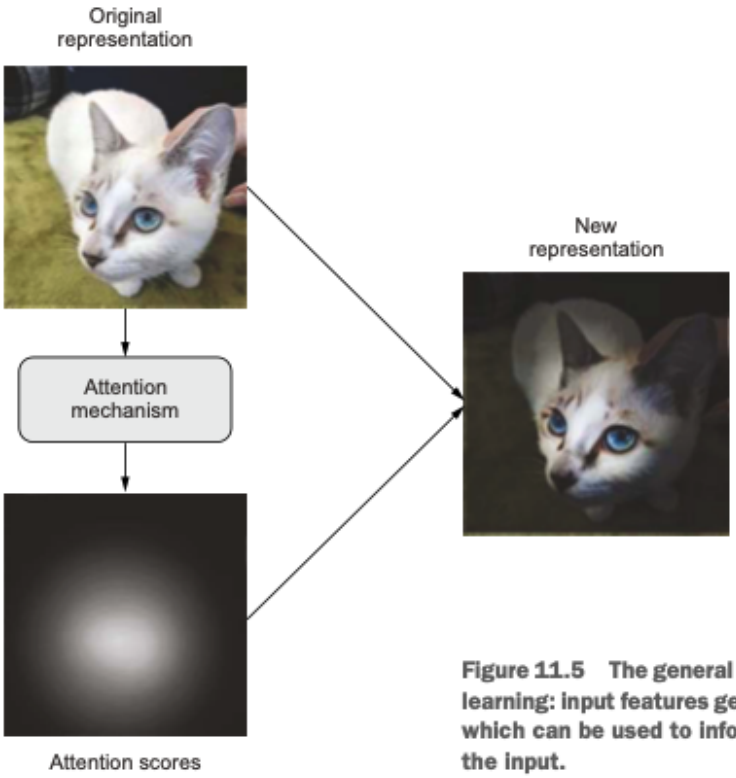


Figure 11.5 The general concept of "attention" in deep learning: input features get assigned "attention scores," which can be used to inform the next representation of the input.

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 mean- ing 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).

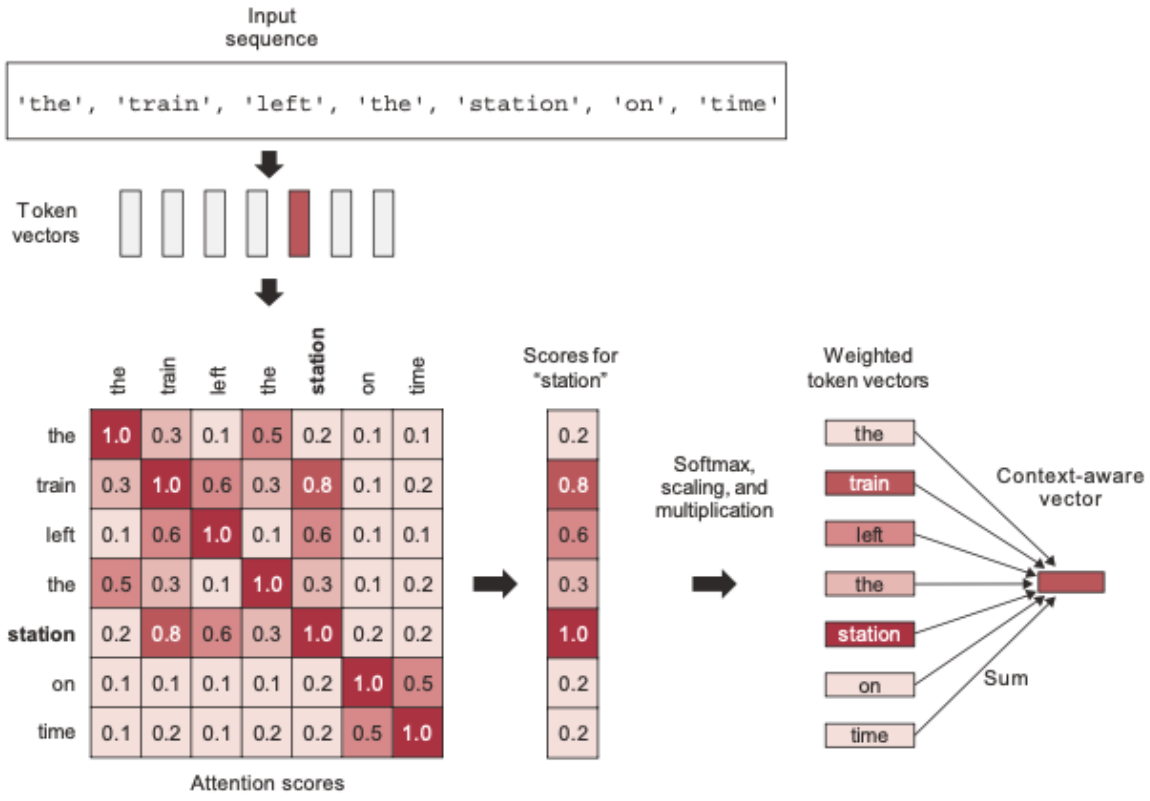


Figure 11.6 Self-attention: attention scores are computed between "station" and every other word in the sequence, and they are then used to weight a sum of word vectors that becomes the new "station" vector.

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).

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:

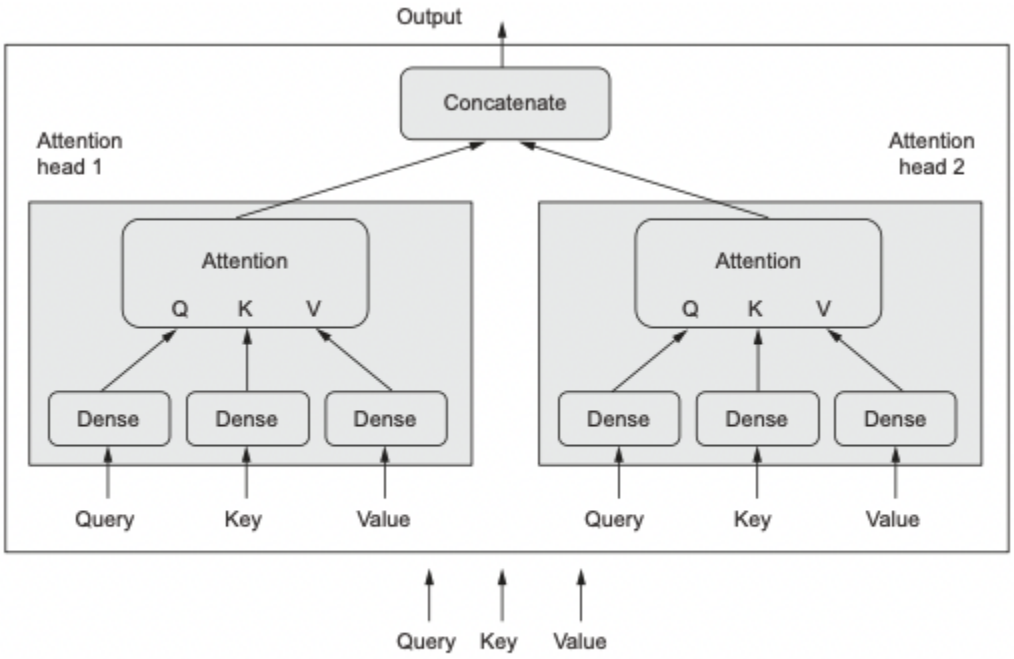


Figure 11.8 The `MultiHeadAttention` layer

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:

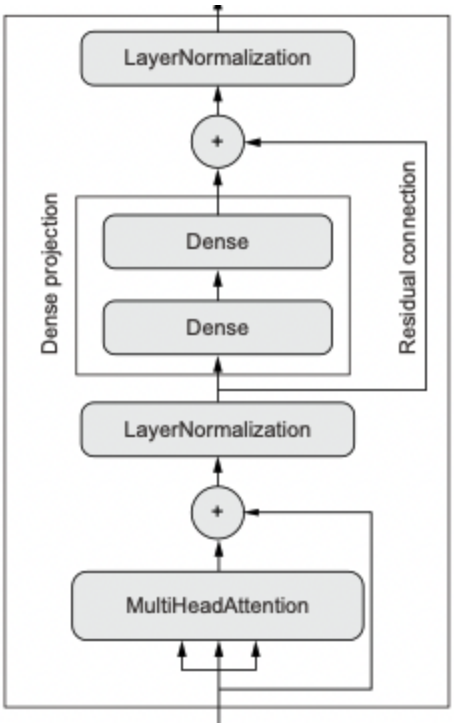


Figure 11.9 The `TransformerEncoder` chains a `MultiHeadAttention` layer with a dense projection and adds normalization as well as residual connections.

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:

```
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 Tranfomers 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}")
```

RASA WhiteBoard Videos on Attention and Transformers: Link to video (there are 4 videos in total)

Transformers

DATE: PAGE:

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

Attention - Mechanism

"VDK learns transfomers"

On language it's not a really great idea to use proximity

tokens	t_1	t_2	t_3
represent word vec	v_1	v_2	v_3
re-weigh	w_1	w_2	w_3
context vector	y_1	y_2	y_3

$\vec{V}_{king}$ $\vec{V}_{queen}$

Reweighing - Plan

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

Bank of the river

v_1 v_2 v_3 v_4

RE-WEIGHT PROCESS

y_1 y_2 y_3 y_4

consider v_1

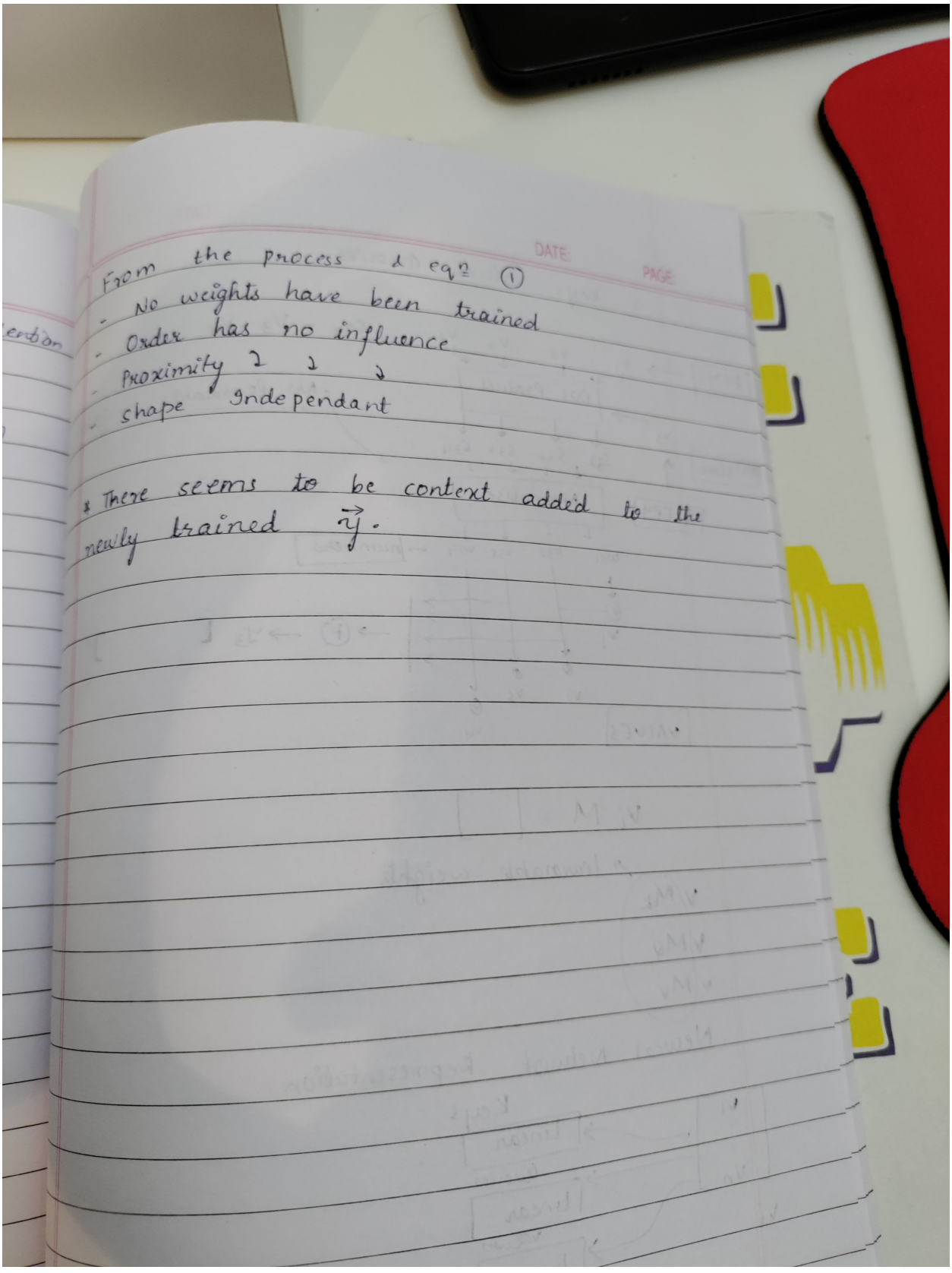
$v_1 \cdot v_1 = w_{11}$ w_{11}

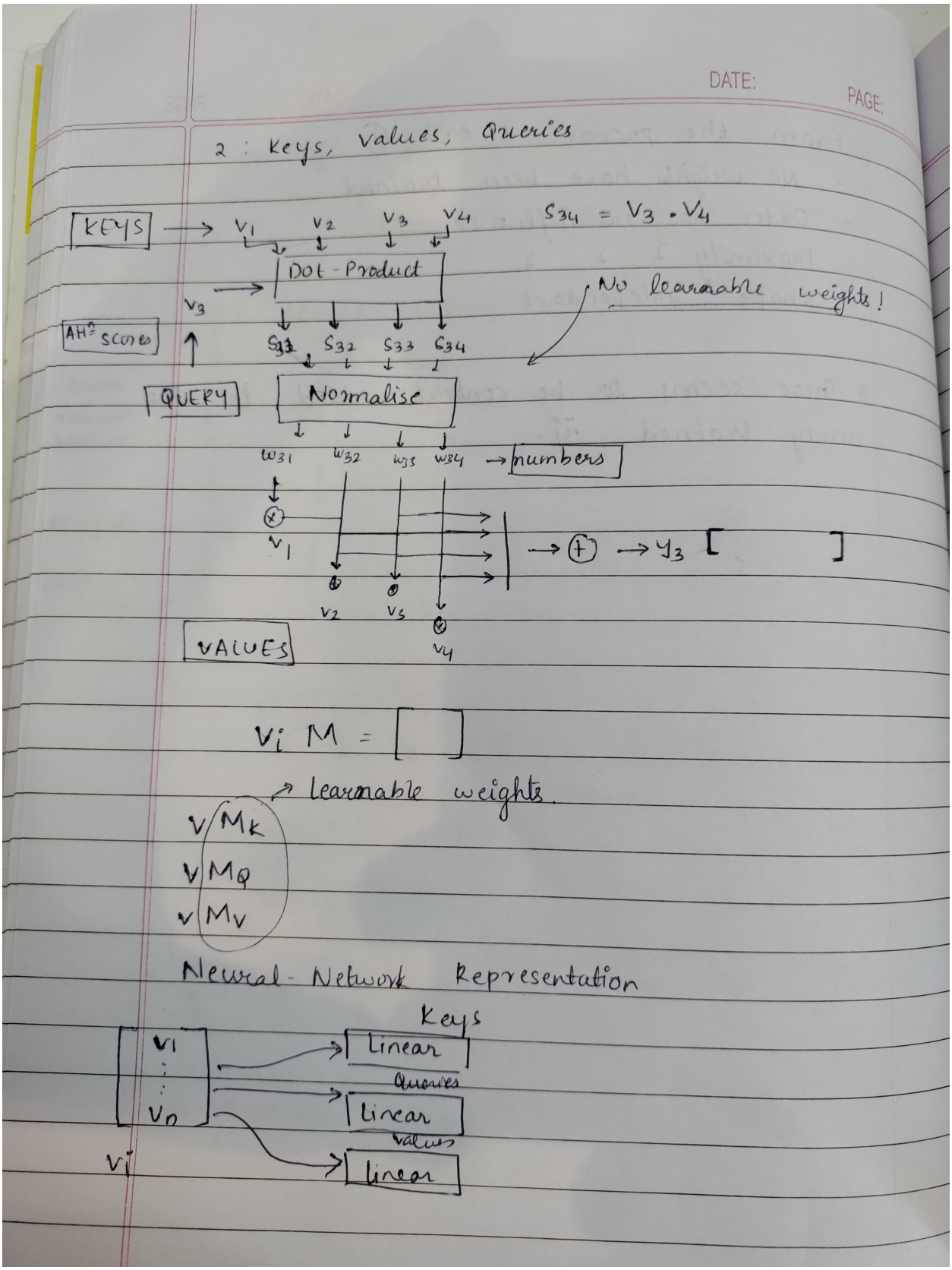
$v_1 \cdot v_2 = w_{12}$ → w_{12}

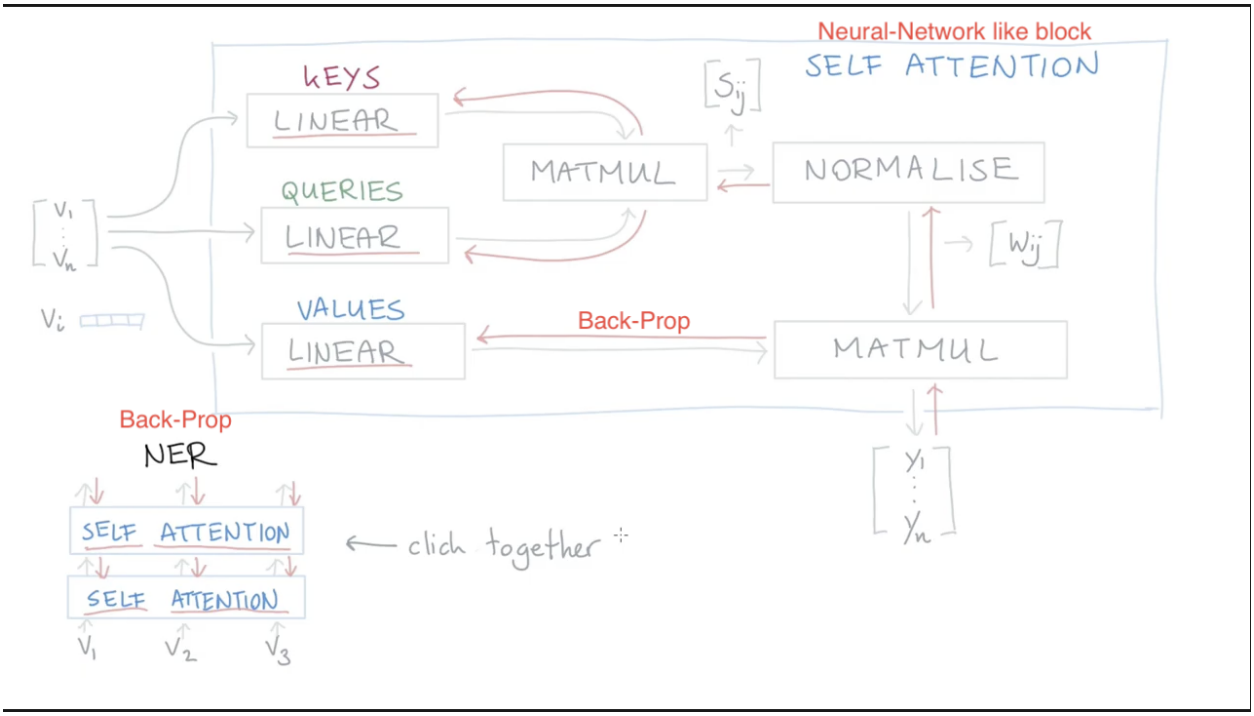
$v_1 \cdot v_3$ w_{13} Normalize w_{13}

$v_1 \cdot v_4$ w_{14} w_{14}

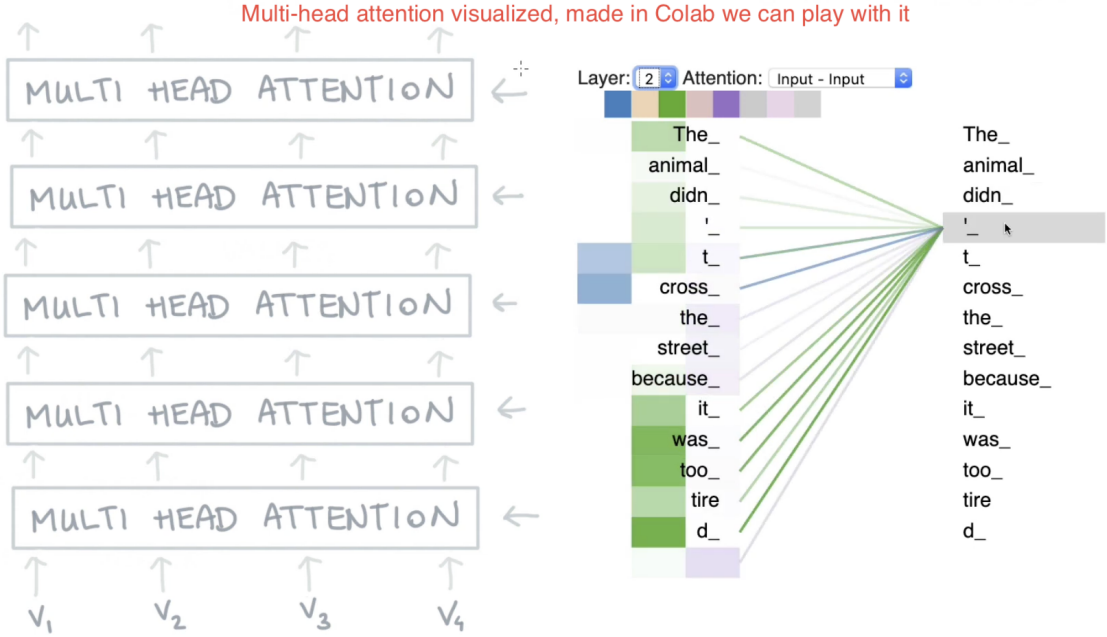
$\therefore y_1 = w_{11}v_1 + w_{12}v_2 + w_{13}v_3 + w_{14}v_4$ — (1)


From the process & eq² ①
- No weights have been trained
- Order has no influence
- Proximity ↓ ↓ ↓
- Shape Independant
* There seems to be context added to the newly trained $\vec{y}$.


DATE:
PAGE:
2 : Keys, Values, Queries
KEYS
v_1 v_2 v_3 v_4
$s_{34} = v_3 \cdot v_4$
Dot-Product
v_3
No learnable weights!
Attn scores
s_{31} s_{32} s_{33} s_{34}
QUERY
Normalise
w_{31} w_{32} w_{33} w_{34} → numbers
v_1 v_2 v_3 v_4
→ + → y_3 [ ]
VALUES
v_i M = [ ]
learnable weights.
v M_K
v M_Q
v M_V
Neural-Network Representation
Keys
v_1
v_n
Linear
Queries
Linear
values
Linear
v_i



Multi-head attention visualized, made in Colab we can play with it



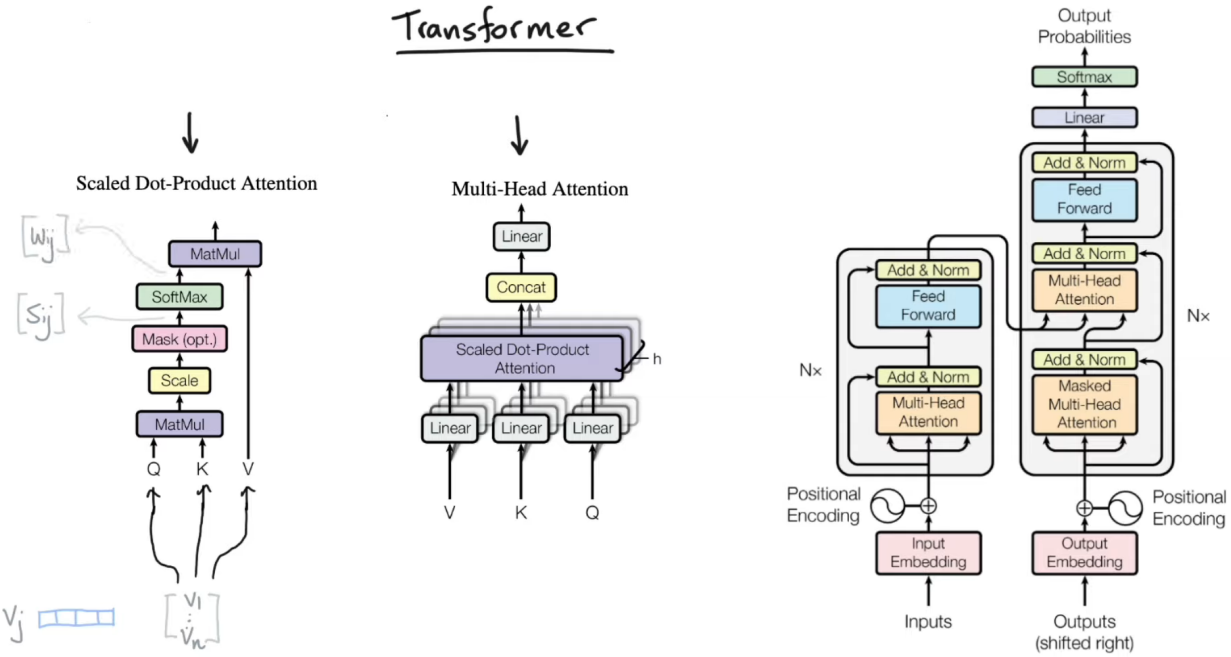
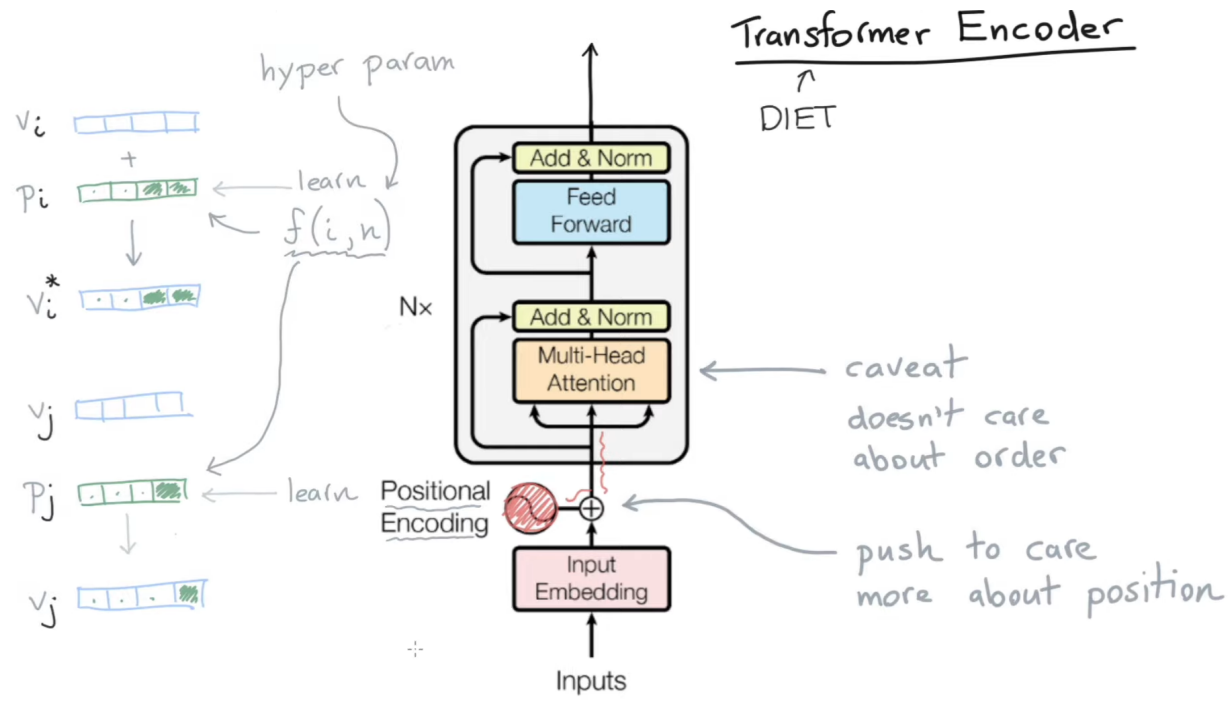


Figure 1: The Transformer - model architecture.

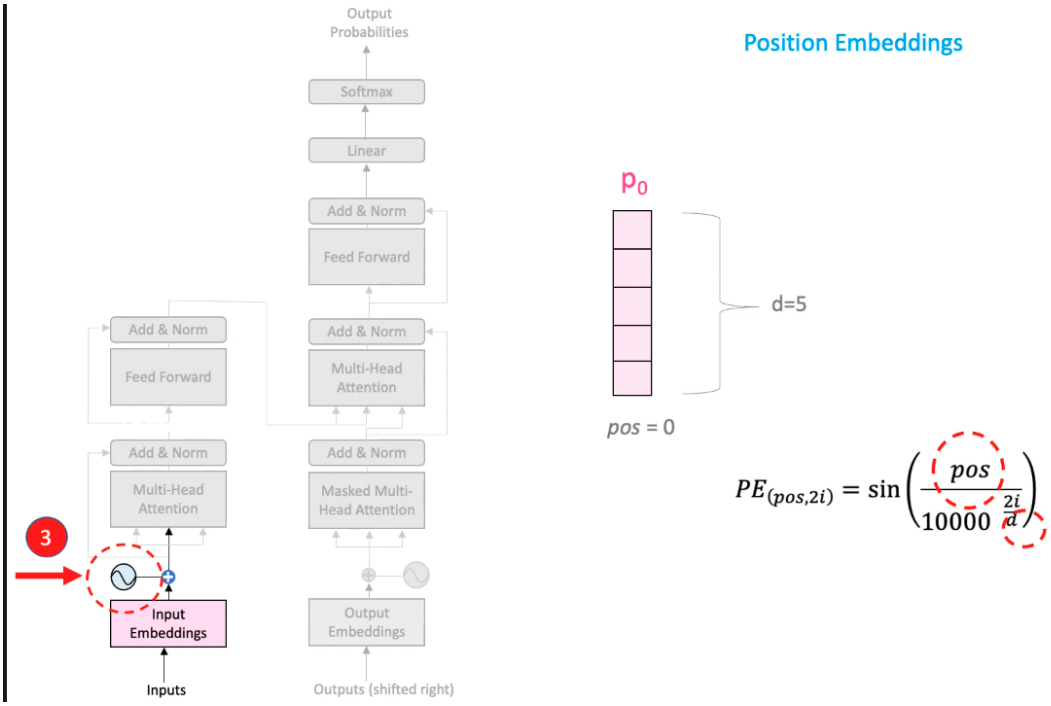


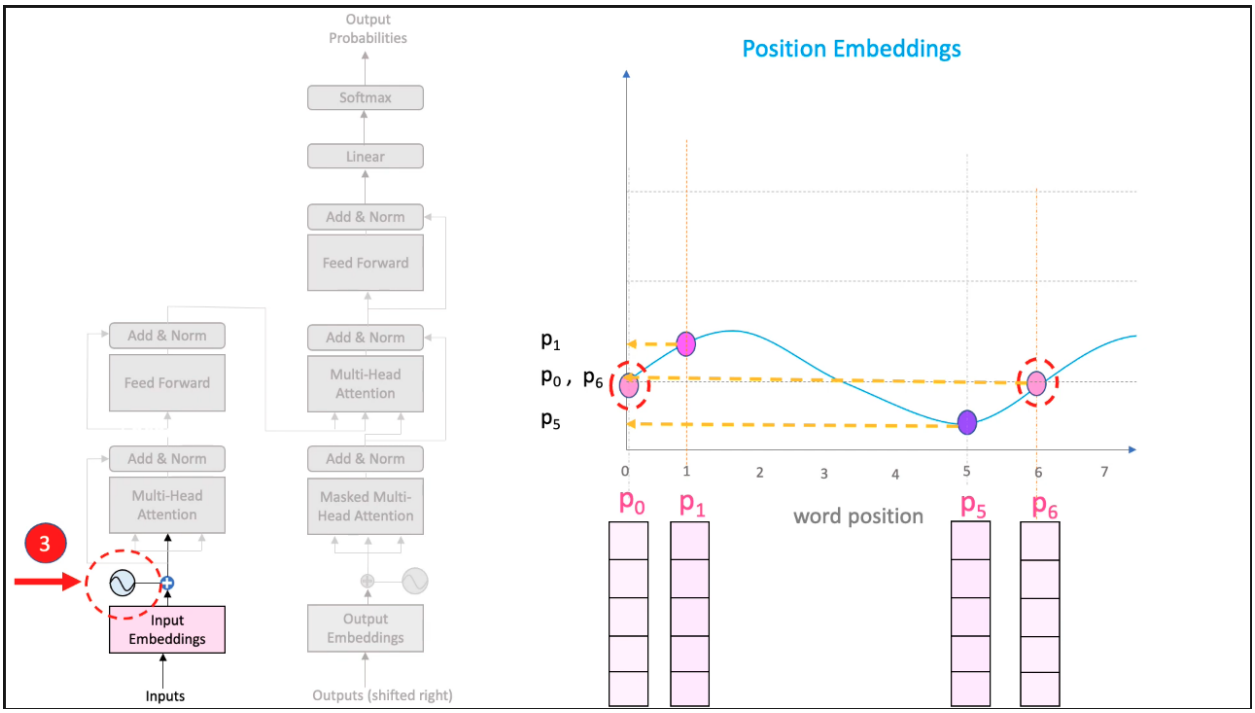
Positional Embeddings:

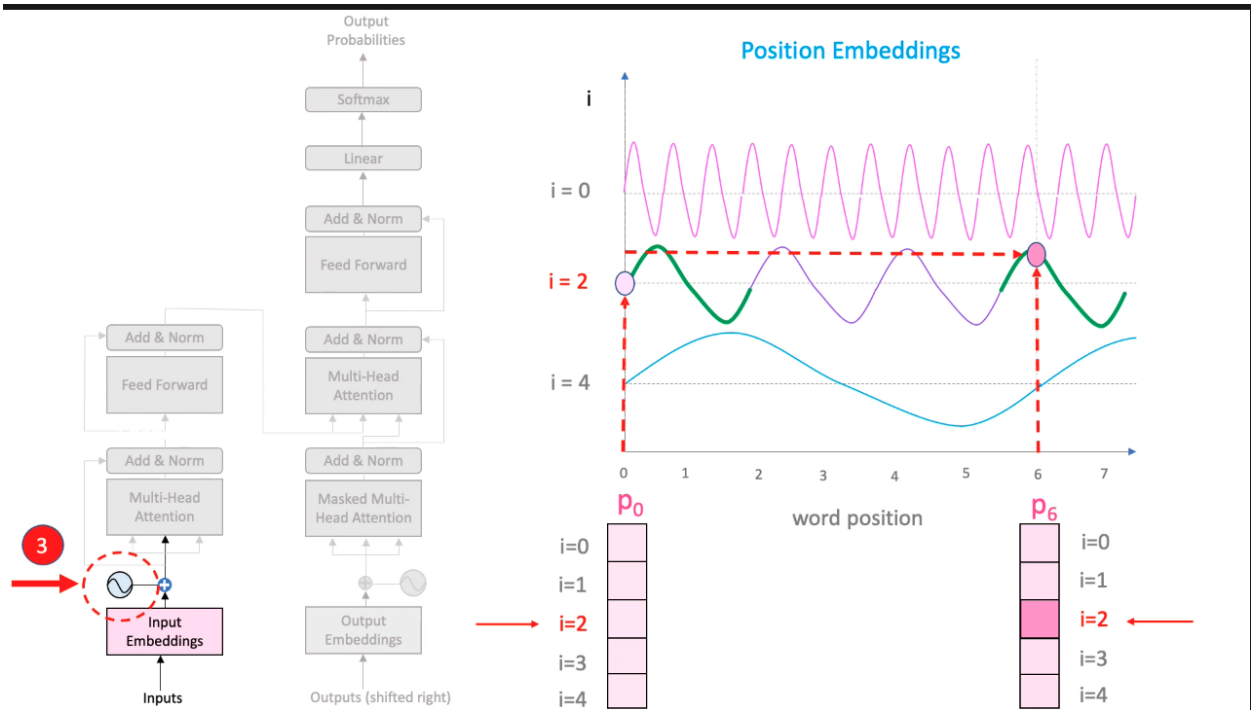
They contain information about the word order.

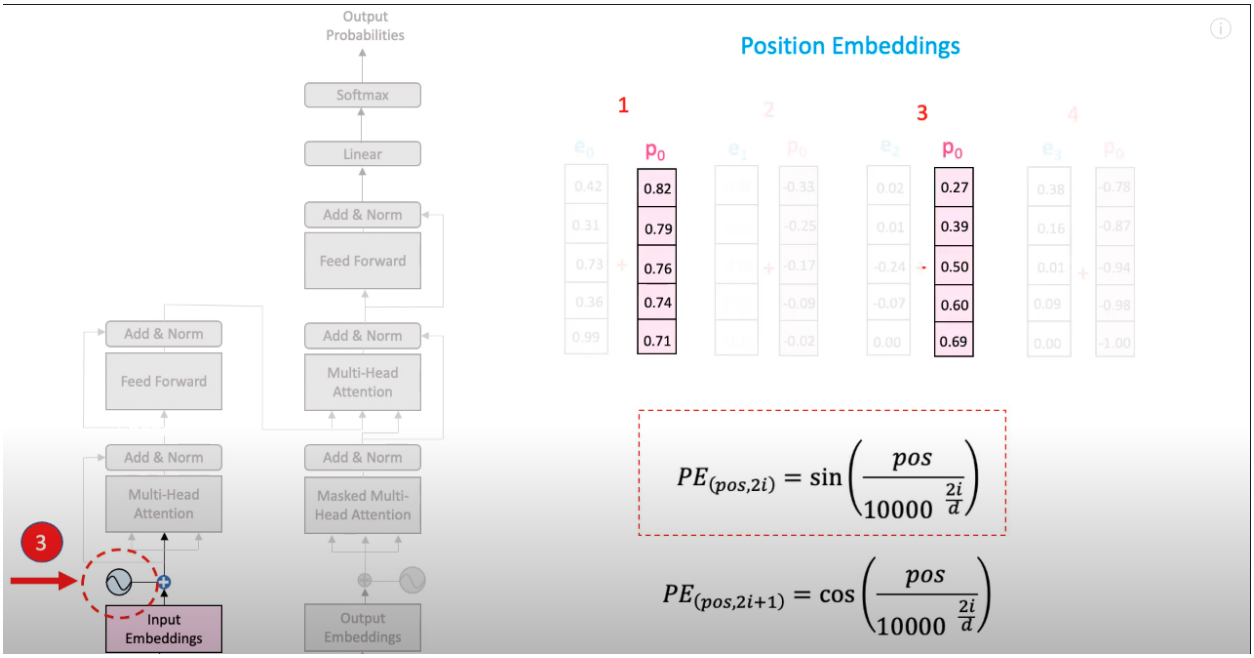
1. Visual Guide to Transformer Neural Networks - (Episode 1) Position Embeddings
2. **Transformer Architecture: The Positional Encoding (Blog)**
3. **The Annotated Transformer (blog with paper and accompanying PyTorch code)**

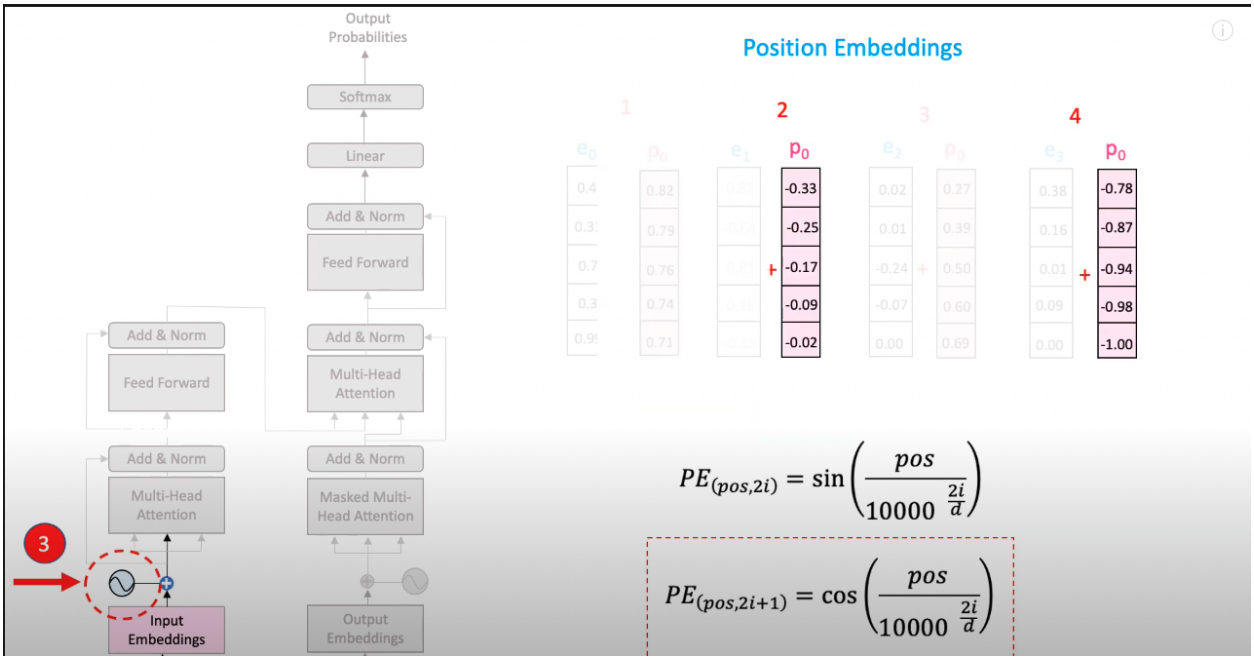
Visual Guide to Transformer Neural Networks - (Episode 1) Position Embeddings Notes




Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
3
Input Embeddings
Output Embeddings
Inputs
Outputs (shifted right)
Position Embeddings
p1
p0 , p6
p5
0 1 2 3 4 5 6 7
word position
p0 p1 p5 p6


Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
3
Input Embeddings
Output Embeddings
Inputs
Outputs (shifted right)
Position Embeddings
i
i = 0
i = 2
i = 4
0 1 2 3 4 5 6 7
word position
p0
p6
i=0
i=1
i=2
i=3
i=4





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, $\overrightarrow{p_t} \in \mathbb{R}^d$ be its corresponding encoding, and d be the encoding dimension (where $d \equiv_2 0$)
Then $f : \mathbb{N} \to \mathbb{R}^d$ will be the function that produces the output vector $\overrightarrow{p_t}$ and it is defined as follows:

$$\overrightarrow{p_t}^{(i)} = f(t)^{(i)} := \begin{cases} \sin(\omega_k . t), & \text{if } i = 2k \\ \cos(\omega_k . t), & \text{if } i = 2k+1 \end{cases}$$

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:

$$\psi'(w_t) = \psi(w_t) + \overrightarrow{p_t}$$

To make this summation possible, we keep the positional embedding's dimension equal to the word embeddings' dimension i.e.
$d_{\text{word embedding}} = d_{\text{postional embedding}}$

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, PEpos+k can be represented as a linear function of PEpos.

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?**

Next-Steps:

To better the understanding on self-attention itself lets go through these videos:

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=0PjHri8tc1c~~
2. ~~Next better your understanding on Multi-Head attention~~
   ~~a. https://www.youtube.com/watch?v=A1cUVxscNq8~~
3. Some other Transformer Video :
   ~~a. https://www.youtube.com/watch?v=dichIcUZfOw~~
   ~~b. https://www.youtube.com/watch?v=mMa2PmYJlCo~~
4. Stanford Lecture 9 - Self- Attention and Transformers **[Later, during revision]**
5. Full Stack DL—> Transformers **[Later, during revision]**
6. **Go through DL FC text-book and implement positional embedding in Keras.**
7. Understand the sequence to sequence models.
8. Understand the decoder part of the transformers.
9. Build better intuition for why self-attention, positional encoding works ?