

50.039 Theory and Practice of Deep Learning

W8-S3 Word Embedding Biases, Evaluation, Attention and Transformers

Matthieu De Mari



About this week (Week 8)

1. Why are **embeddings** an essential component of Neural Networks (NNs)?
2. Why are **good embeddings** difficult to produce?
3. What are the **conventional approaches to embeddings in NLP**?
What can we **learn from these approaches**?
4. What are the **typical issues with embeddings** and how do we address them?
5. **State-of-the-art** of current embedding problems, and **open questions** in research.

About this week (Week 8)

6. How do we evaluate the **quality/performance** of an embedding?
7. Can embeddings be **biased**?
8. Can we help the neural networks **identify the important parts of the context** to focus on?
9. What is **attention** in Neural Networks? What are **transformers** in Neural Networks?
10. What are the typical **uses for attention** these days?
11. What are the **limits of attention** and the **current research directions** on this topic?

In the last episode...

We have discussed the **embedding** problem, and its aspects in Natural Language Processing. Many approaches have been used to improve the quality of word embeddings.

- CBoW & SkipGram (2013),
- GloVe (2014),
- FastText (2018),
- ELMo (2018).

Question: What makes a word embedding better than another? How do we measure the performance of an embedding?

Measuring Embeddings Performance

Evaluating the performance of an embedding is a difficult task and still an open question in research [Schnabel2015].

- **Reason:** it is very difficult to come up with a performance **measure** to objectively evaluate the quality of a word embedding.

Overall, **two big families of approaches:**

- **Extrinsic methods**
- **Intrinsic Methods**

Measuring Embeddings Performance

Method #1 (Extrinsic Evaluation):

Methods of **extrinsic evaluation** are based on the ability of word embeddings to be used as the feature vectors of supervised machine learning algorithms.

The more able a word embedding is on a given (complex) task, the better it might be.

Typically, training a CBoW requires to predict a missing word, in the middle of a given sentence.

This is an easier task to train on, compared to predicting the possible context words for a given single word (SkipGram)

For this reason, we often assume that **SkipGram > CBoW**.

Measuring Embeddings Performance

In general, we train an embedding on a task and extract a layer from our Neural Network, corresponding to the embedding.

Extrinsic Evaluation: Evaluate the embedding by reusing it on another NLP benchmark task (using transfer learning).

Researchers have identified a set of typical NLP tasks and datasets on which we need word embeddings [Schnabel2015], e.g.

- Sentiment Analysis
- Translation
- Etc.

These have often been used as reference for measuring the performance of a given word embedding.

Measuring Embeddings Performance

Method #2 (Intrinsic Evaluation):

Methods of **intrinsic evaluation** are experiments in which word embeddings are compared with human judgments on words relations.

Typically, we will rely on a **human expert to judge** whether the word embedding is good or not.

Measuring Embeddings Performance

Definition (**cosine angle** between two vectors):

Researchers have tried to come up with an “**objective**” metric to compute the **similarity** between two word embeddings vectors.

For two vectors u, v , we define the **cosine angle** between both vectors as:

$$\cos(u, v) = \frac{\langle u, v \rangle}{\|u\| \|v\|}$$

The cosine angle formula verifies

$$\cos(u, v) = \begin{cases} 1 & \text{if } u = v \\ -1 & \text{if } u = -v \\ 0 & \text{if } u \text{ and } v \text{ are orthogonal} \end{cases}$$

A value close to 1 (resp. -1) indicate two vectors carrying a similar (resp. opposite) semantic meaning. A value 0 indicates two uncorrelated words.

Measuring Embeddings Performance

- **Rule #1: A “good” word embedding should model semantic proximity.**

One can check that word analogies are preserved by the word embedding, by e.g. looking for the top 10 words (u_1, \dots, u_{10}) with the largest cosine distance for any given word v .

These should make sense (somewhat)

- For instance (from the FastText tutorial <https://fasttext.cc/docs/en/unsupervised-tutorial.html>)

```
Query word? asparagus  
beetroot 0.812384  
tomato 0.806688  
horseradish 0.805928  
spinach 0.801483  
licorice 0.791697  
lingonberries 0.781507  
asparagales 0.780756  
lingonberry 0.778534  
celery 0.774529  
beets 0.773984
```

```
Query word? pidgey  
pidgeot 0.891801  
pidgeotto 0.885109  
pidge 0.884739  
pidgeon 0.787351  
pok 0.781068  
pikachu 0.758688  
charizard 0.749403  
squirtle 0.742582  
beedrill 0.741579  
charmeleon 0.733625
```

Measuring Embeddings Performance

- Rule #1: A “good” word embedding should model semantic proximity.

One can check that word

Are these top10 lists good or bad answers?

Your answer to this question will probably be very subjective and not a reliable performance evaluation metric...!



- For instance (from the FastText tutorial <https://fasttext.cc/docs/en/unsupervised-tutorial.html>)

```
Query word? asparagus  
beetroot 0.812384  
tomato 0.806688  
horseradish 0.805928  
spinach 0.801483  
licorice 0.791697  
lingonberries 0.781507  
asparagales 0.780756  
lingonberry 0.778534  
celery 0.774529  
beets 0.773984
```

```
Query word? pidgey  
pidgeot 0.891801  
pidgeotto 0.885109  
pidge 0.884739  
pidgeon 0.787351  
pok 0.781068  
pikachu 0.758688  
charizard 0.749403  
squirtle 0.742582  
beedrill 0.741579  
charmeleon 0.733625
```

Measuring Embeddings Performance

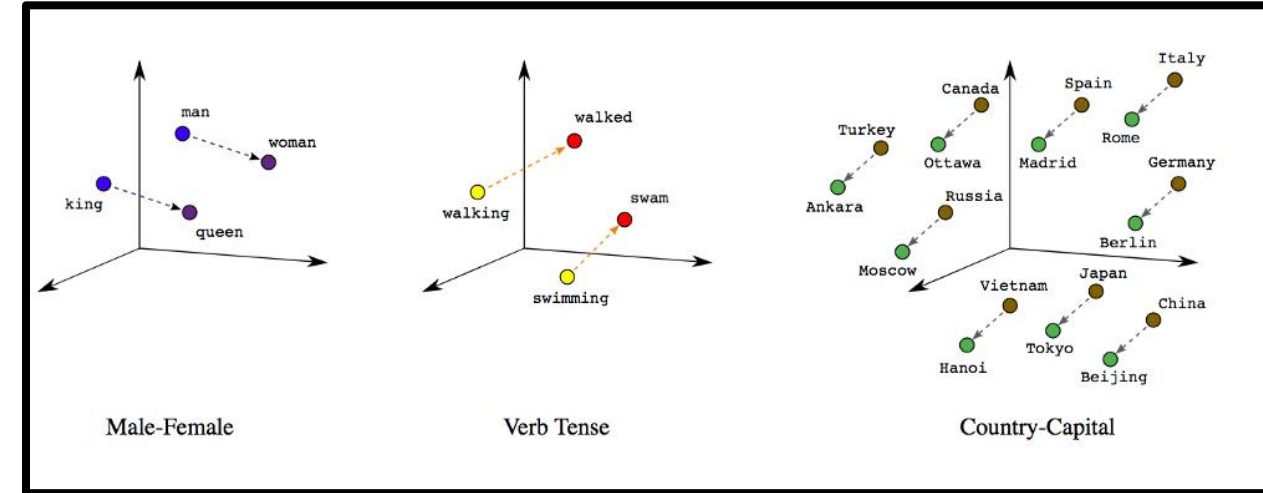
- **Rule #2: A “good” word embedding should preserve word analogies.**

Word analogies are sentences reading as “A is to B, what C is to D”, with A, B, C and D being four words.

Analogies are typically modeled by relations such as:

$$W_A - W_B = W_C - W_D$$

$$W_A = W_C - W_D + W_B$$



Measuring Embeddings Performance

- Rule #2: A “good” word embedding should preserve word analogies.

Word analogies are sentences

Is this top10 list containing good or bad answers?

Your answer to this question will (again!) probably be very subjective and not a reliable performance evaluation metric...!



- For instance (from the FastText tutorial <https://fasttext.cc/docs/en/unsupervised-tutorial.html>)

```
$ ./fasttext analogies result/fil9.bin
Pre-computing word vectors... done.
Query triplet (A - B + C)? berlin germany france
paris 0.896462
bourges 0.768954
louveciennes 0.765569
toulouse 0.761916
valenciennes 0.760251
montpellier 0.752747
strasbourg 0.744487
meudon 0.74143
bordeaux 0.740635
pigneaux 0.736122
```

Measuring Embeddings Performance

- Rule #2: A “good” word embedding should preserve word analogies.

Word analogies are sentences

Is this top10 list containing good or bad answers?

Your answer to this question will (again!) probably be very subjective and not a reliable performance evaluation metric...!

More specifically, because we are unsure **what the word embedding values represent in terms of semantics**.

There have been observed cases where the word embedding understands some “sort of relationship” but cannot properly distinguish between the different types of relationships, e.g.

$$W_{driver} - W_{car} = W_{king} - W_{queen}$$

Measuring Embeddings Performance

- **Rule #3: A “good” word embedding should be able to operate on variations on the words (color/colour, typos, plural forms, conjugation, etc.).**

Is this top10 list containing good or bad answers?

Your answer to this question will (again!) probably be very subjective and not a reliable performance evaluation metric...!

For instance (from the FastText tutorial

<https://fasttext.cc/docs/en/unsupervised-tutorial.html>)

```
Query word? accomodation
accomodations 0.96342
accommodation 0.942124
accommodations 0.915427
accommodative 0.847751
accommodating 0.794353
accomodated 0.740381
amenities 0.729746
catering 0.725975
accomodate 0.703177
hospitality 0.701426
```


Word embedding biases

Important: A quick note on word embeddings **biases**.

- As mentioned before, we value the quality of a word embedding by how good it manages to capture similarities and analogies between words, by extracting it from a given corpus.
- **Problem:** if the corpus is **biased** (sexist, racist, etc.), it will catch these **biases** as well! (See [Bolukbasi2016]).

$$w_{man_job} - w_{man} + w_{woman} = ?$$

Extreme *she* occupations

- | | | |
|-----------------|-----------------------|------------------------|
| 1. homemaker | 2. nurse | 3. receptionist |
| 4. librarian | 5. socialite | 6. hairdresser |
| 7. nanny | 8. bookkeeper | 9. stylist |
| 10. housekeeper | 11. interior designer | 12. guidance counselor |

Extreme *he* occupations

- | | | |
|----------------|-------------------|----------------|
| 1. maestro | 2. skipper | 3. protege |
| 4. philosopher | 5. captain | 6. architect |
| 7. financier | 8. warrior | 9. broadcaster |
| 10. magician | 11. fighter pilot | 12. boss |



Word embeddings evaluation

Question: What makes a word embedding better than another?

How do we measure the performance of an embedding?

How do we decide on the best word embedding to use?

Answer: Still an open question at the moment...

No perfect qualitative method for evaluating Word Embeddings.

More of a “consensus”, which changes over the years.

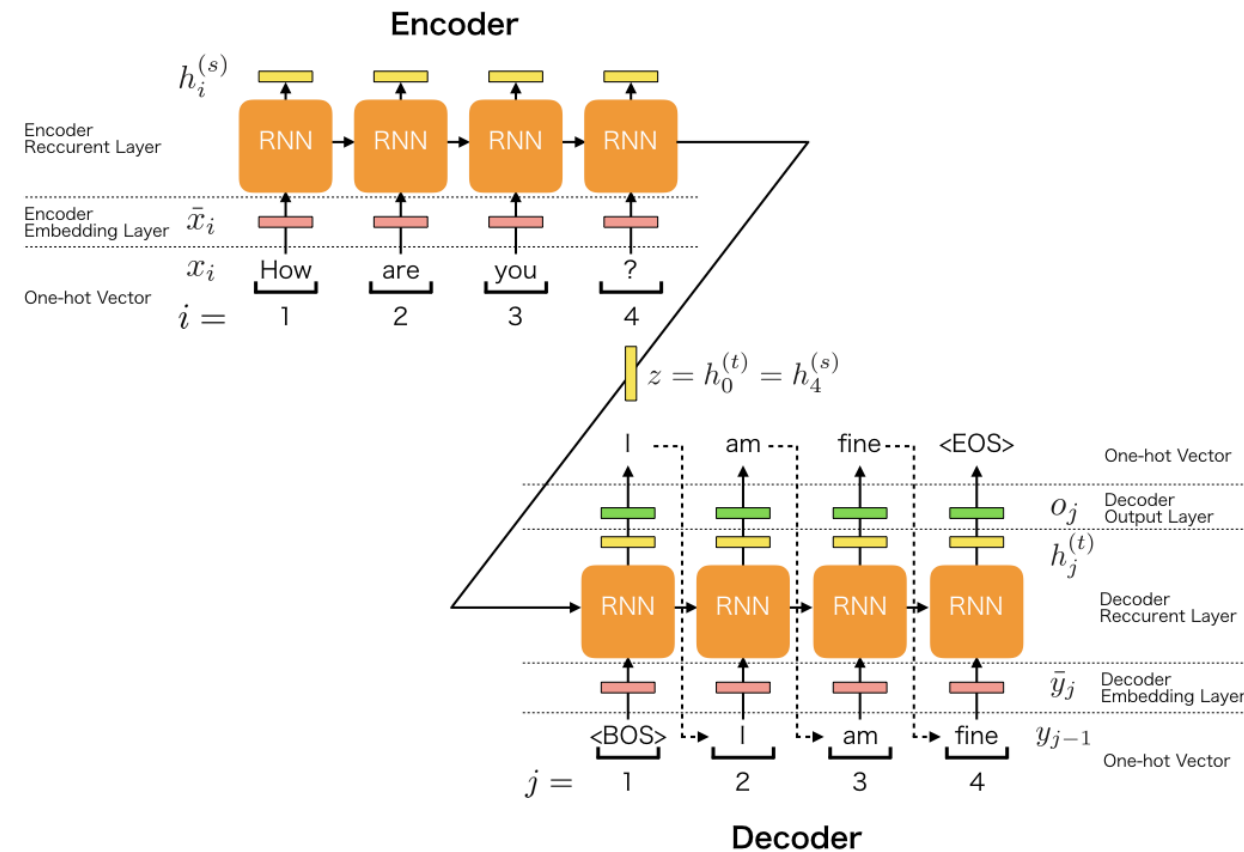
Still a lot of work needed to come up with a “**perfect**” **universal embedding** (if that is even possible?)

Sequence to sequence tasks

Definition (**Seq2Seq**):

A **Sequence-to-Sequence task (Seq2Seq)** is a family of machine learning tasks which turns an input sequence into another output sequence.

For instance, language to language translation (French sentence into an English sentence), answers to questions Chatbots, etc.



Sequence to sequence tasks

Definition (**Encoder-Decoder** architecture):

An **Encoder-Decoder** architecture typically consists of two parts:

- An **encoder**, which converts a sequence into an embedding vector, presumably encapsulating the **context** of the sequence.
- A **decoder**, which reads the embedding vector for **context** and constructs an output sequence.
- Both the encoder and the decoder will be modeled as RNNs as they both operate on sequences.

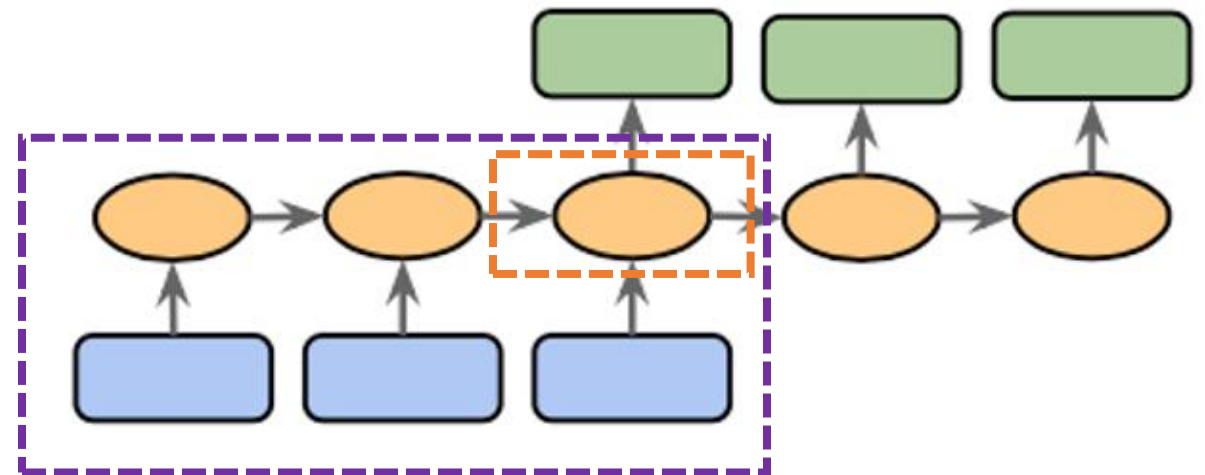
Encoder and Decoder models

Definition (**encoder and decoder models**):

In general, the first part of the model will focus on analysing the input, eventually producing a final memory vector output after having seen all inputs. **This is called an encoder.**

Can be done with a RNN of some sort, where the final memory state consists of the **encoding vector**.

E.g. read an English sentence, progressively encoding each word's meaning, to obtain the entire sentence meaning in a single encoding vector.



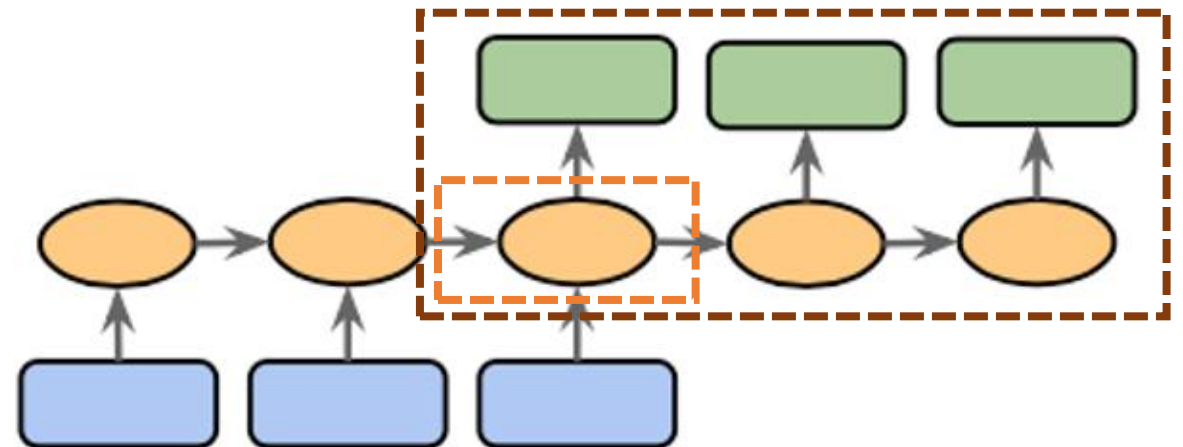
Encoder and Decoder models

Definition (**encoder and decoder models**):

In general, the second part of the model will focus on analysing the produced **encoding vector**, and producing a sequence of some sort as output. **This is called a decoder.**

Can be done with the same (or another) RNN of some sort, where the first memory state consists of the **encoding vector**.

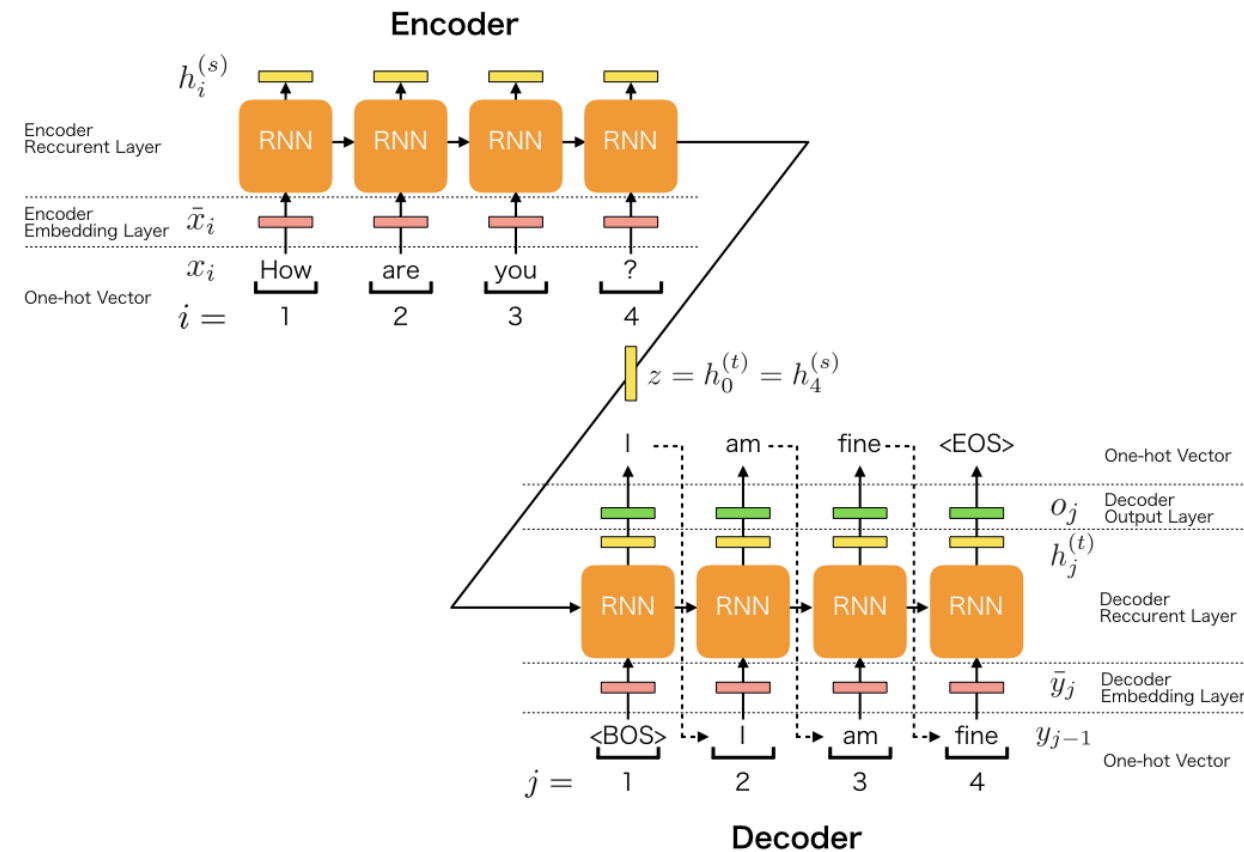
E.g. read an **encoding vector** describing an English sentence and produce an output sequence being the French translation.



Sequence to sequence tasks

In the classical Seq2Seq architecture,

- a **word embedding** is used for each word of the sentence,
- An (often bi-directional RNN) **encoder** is used to propagate and encode context among words in the sentence,
- A **decoder** RNN reads the context vector and produces a new sentence.



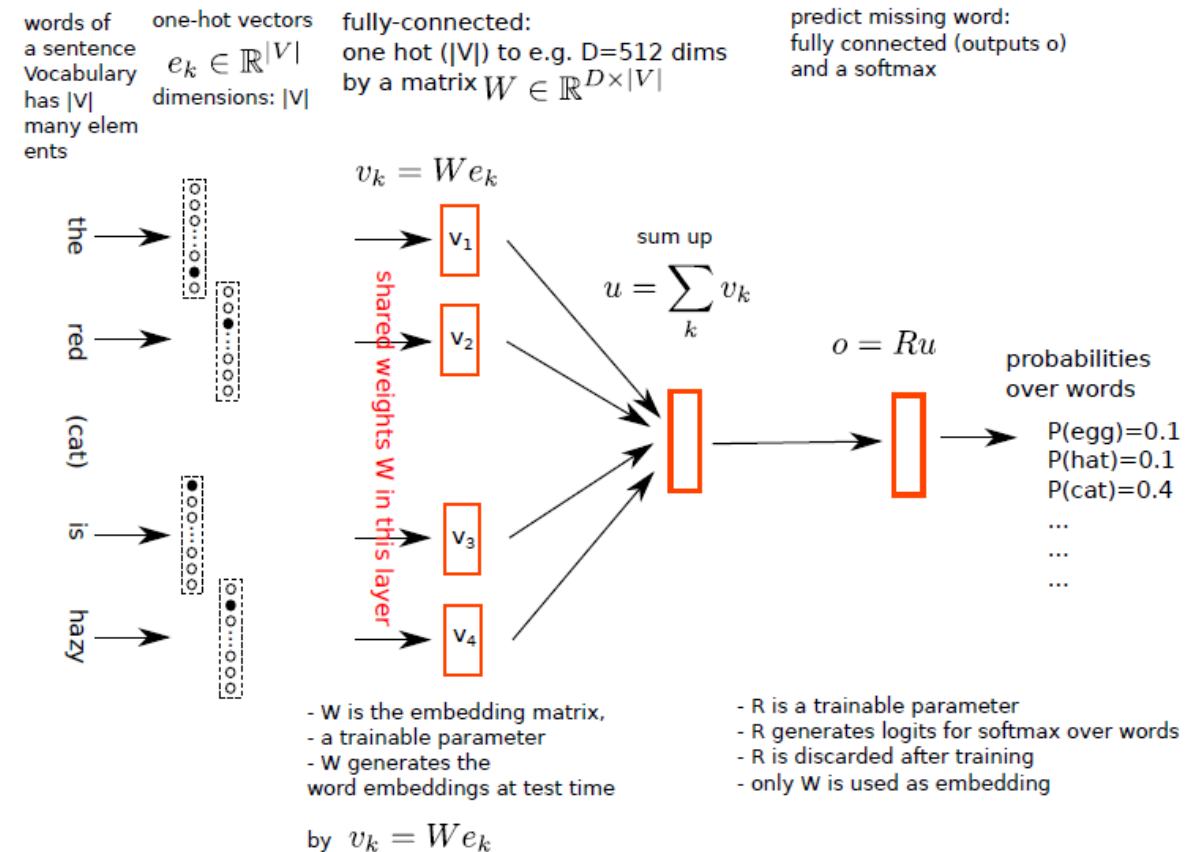
Sequence to sequence tasks

How is our context vector modeled again?

- **Averaging** the words embedding vectors for the words in the context (CBoW)

$$u = \sum_k v_k$$

Simple, but words might cancel each other out. Averaging effect if large number of words.



Sequence to sequence tasks

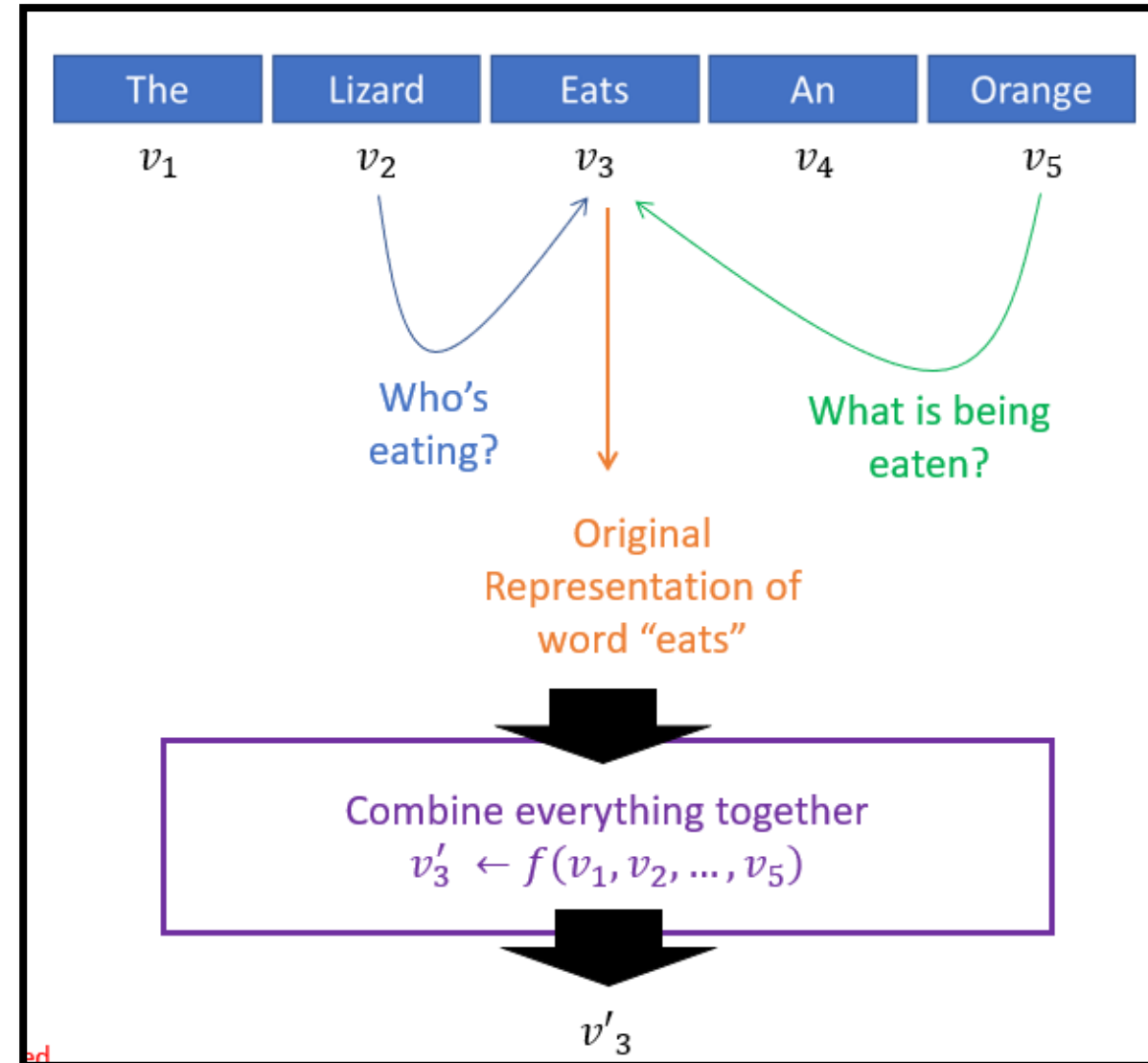
How is our context vector modeled again?

- **Concatenating** the hidden states of the bi-RNN gives surrounding context to word w_k (ELMo)

$$h_k = [\overleftarrow{h}_k, \overrightarrow{h}_k]$$

Preserves and propagates meaning of all surrounding words.

Question, though: What is the memory span of RNNs anyway?



Sequence to sequence tasks

How is our context vector modeled again?

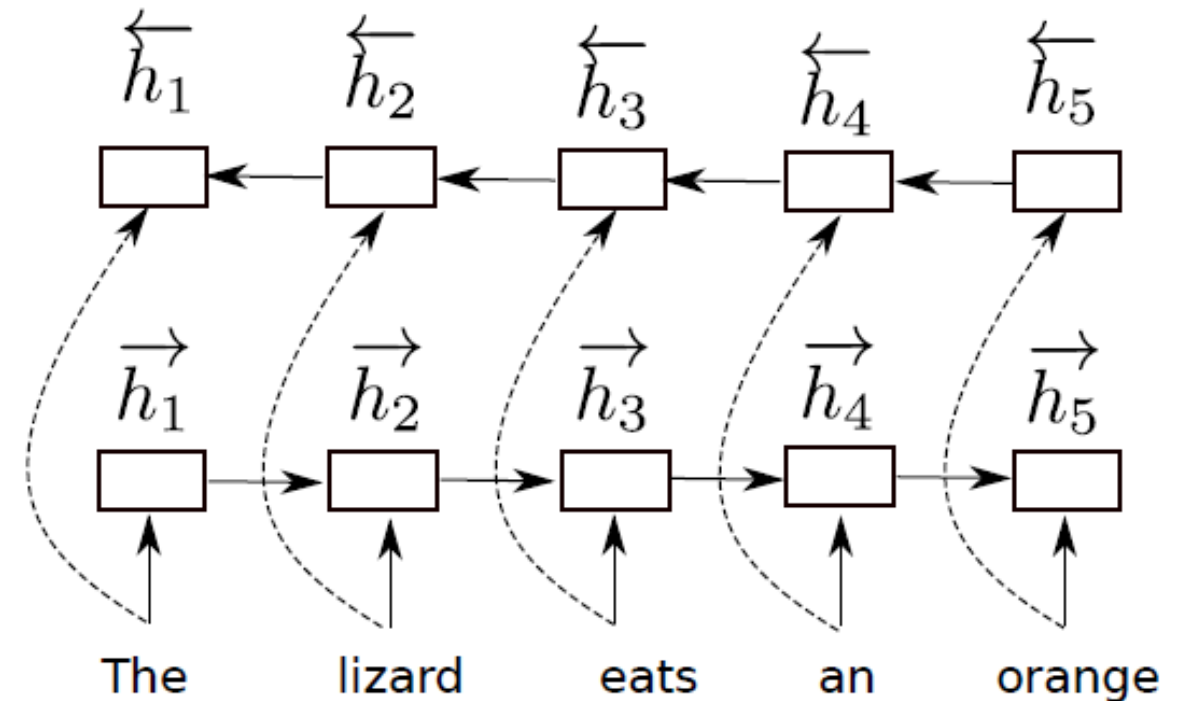
- **Concatenating** the hidden states of the bi-RNN gives surrounding context to word w_k (ELMo)

$$h_k = [\overleftarrow{h}_k, \overrightarrow{h}_k]$$

Preserves and propagates meaning of all surrounding words.

Question, though: What is the memory span of RNNs anyway?

$$h_k = [\overleftarrow{h}_k, \overrightarrow{h}_k]$$



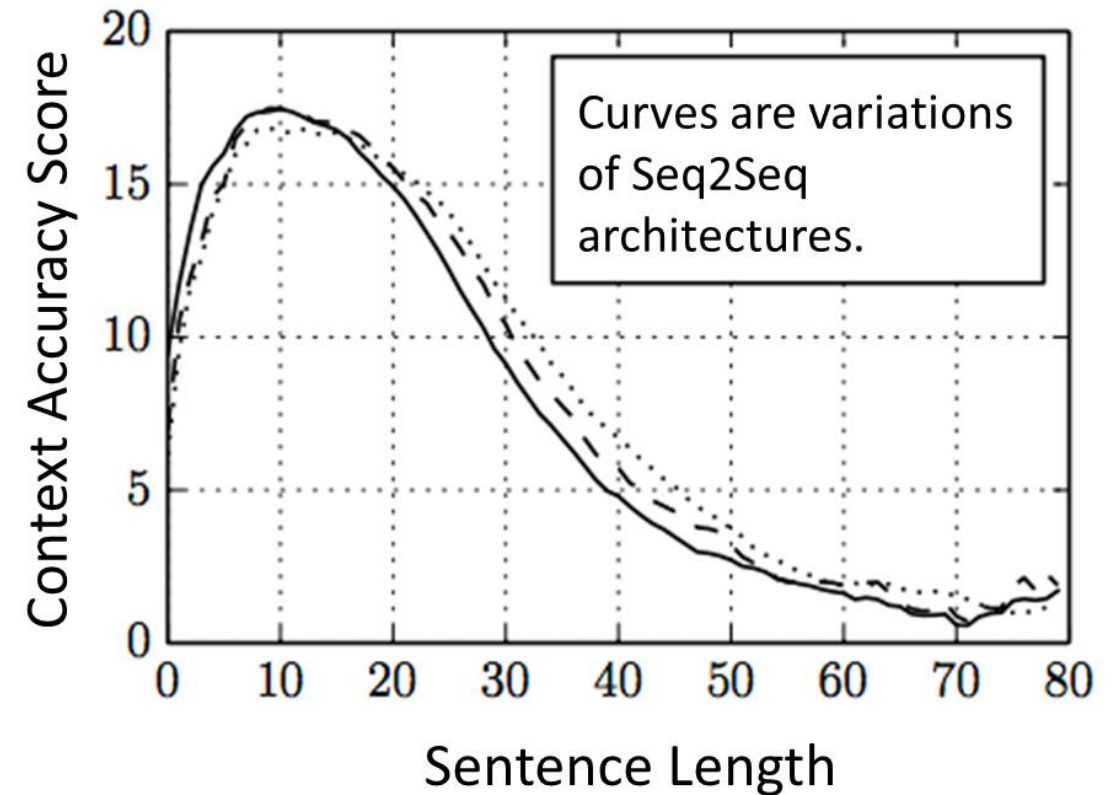
Sequence to sequence tasks

Question: What is the memory span of RNNs anyway?

Vanishing effect in memory between the successive hidden LSTM states if large number of words in sentence [Vaswani2017].

Your RNN will have forgotten about word #1 by the time you reach word #70.

→ Loss of context!



From: <https://www.kaggle.com/sarthakvajpayee/attention-mechanism-japanese-english/notebook>

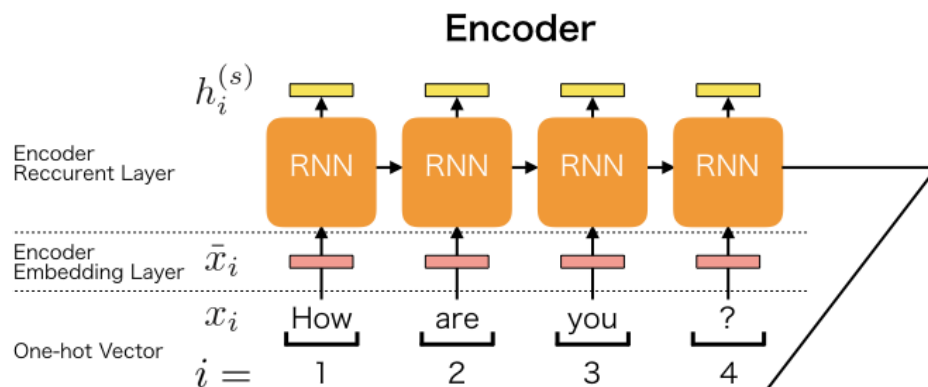
Sequence to sequence tasks

What is the reason for the loss of context meaning in large sentences?

Words that are far away from each other will have their context lost by the hidden state of the LSTM RNN.

- For instance,

“Paris was a universe whole and entire unto herself, hollowed and fashioned by history; so she seemed in this age of Napoleon III with her towering buildings, her massive cathedrals, her grand boulevards, and ancient winding medieval streets—as vast and indestructible as nature itself. All was embraced by her, by her volatile and enchanted populace thronging the galleries, the theaters, the cafes, giving birth over and over to genius and sanctity, philosophy and war, frivolity and the finest art; so it seemed that if all the world outside her were to sink into darkness, what was fine, what was beautiful, what was essential might there still come to its finest flower.”



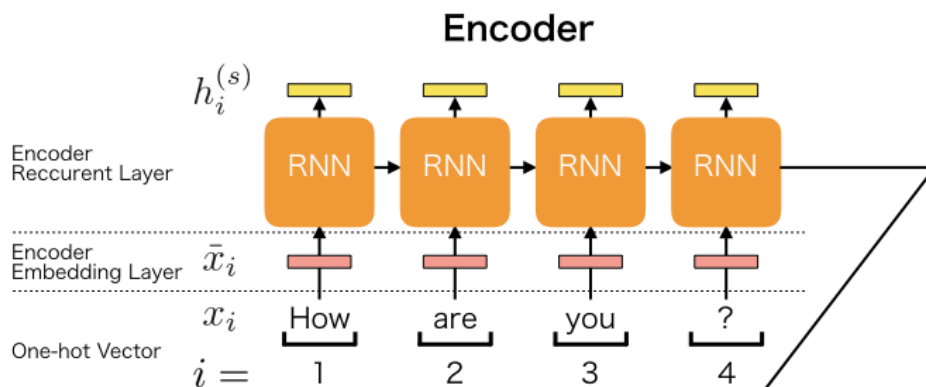
Sequence to sequence tasks

What is the reason for the loss of context meaning in large sentences?

Words that are far away from each other will have their context lost by the hidden state of the LSTM RNN.

- For instance,

*“Paris was a universe whole and entire unto herself, hollowed and fashioned by history; so she seemed in this age of Napoleon III with her towering buildings, her massive cathedrals, her grand boulevards, and ancient winding medieval streets—as vast and indestructible as nature itself. All was embraced by her, by her volatile and enchanted **populace** thronging the galleries, the theaters, the cafes, giving birth over and over to genius and sanctity, philosophy and war, frivolity and the finest art; so it seemed that if all the world outside her were to sink into darkness, what was fine, what was beautiful, what was essential might there still come to its finest flower.”*



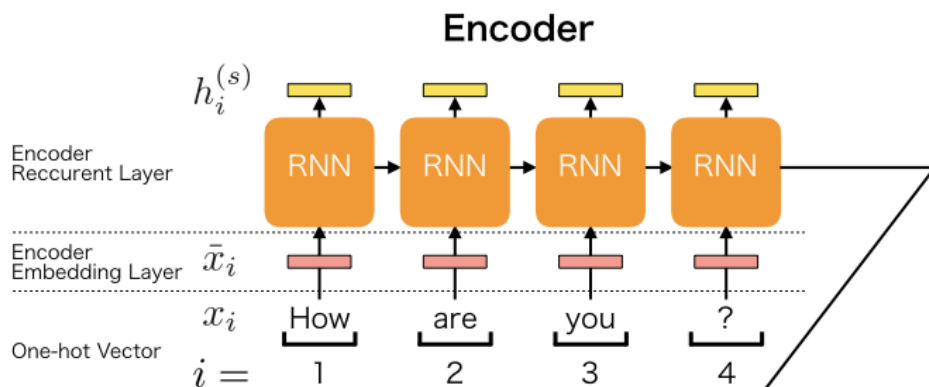
Sequence to sequence tasks

What is the reason for the loss of context meaning in large sentences?

Words that are far away from each other will have their context lost by the hidden state of the LSTM RNN.

- For instance,

*“**Paris** was a universe whole and entire unto herself, hollowed and fashioned by history; so she seemed in this age of Napoleon III with her towering buildings, her massive cathedrals, her grand boulevards, and ancient winding medieval streets—as vast and indestructible as nature itself. All was embraced by her, by her volatile and enchanted **populace** thronging the galleries, the theaters, the cafes, giving birth over and over to genius and sanctity, philosophy and war, frivolity and the finest art; so it seemed that if all the world outside her were to sink into darkness, what was fine, what was beautiful, what was essential might there still come to its finest flower.”*



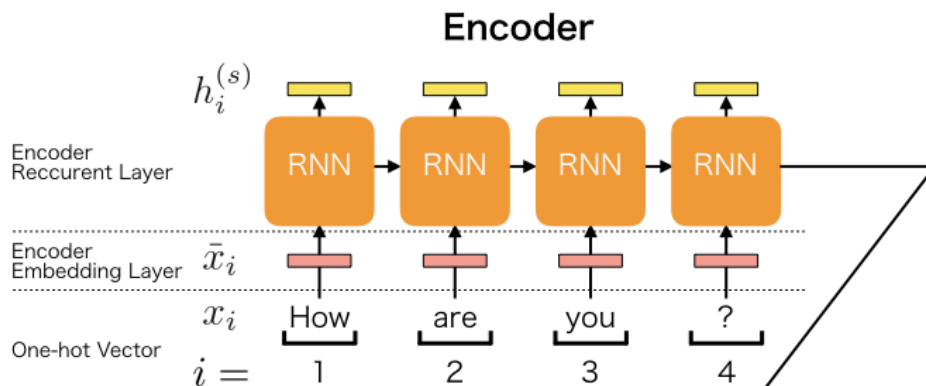
Sequence to sequence tasks

What is the reason for the loss of context meaning in large sentences?

Words that are far away from each other will have their context lost by the hidden state of the LSTM RNN.

Question: Can we train our Neural Networks to get a better grasp of the important words that matter in the sequence and how to connect them, even when far away from each other?

- Yes!
- That is exactly what the **attention layer** does!



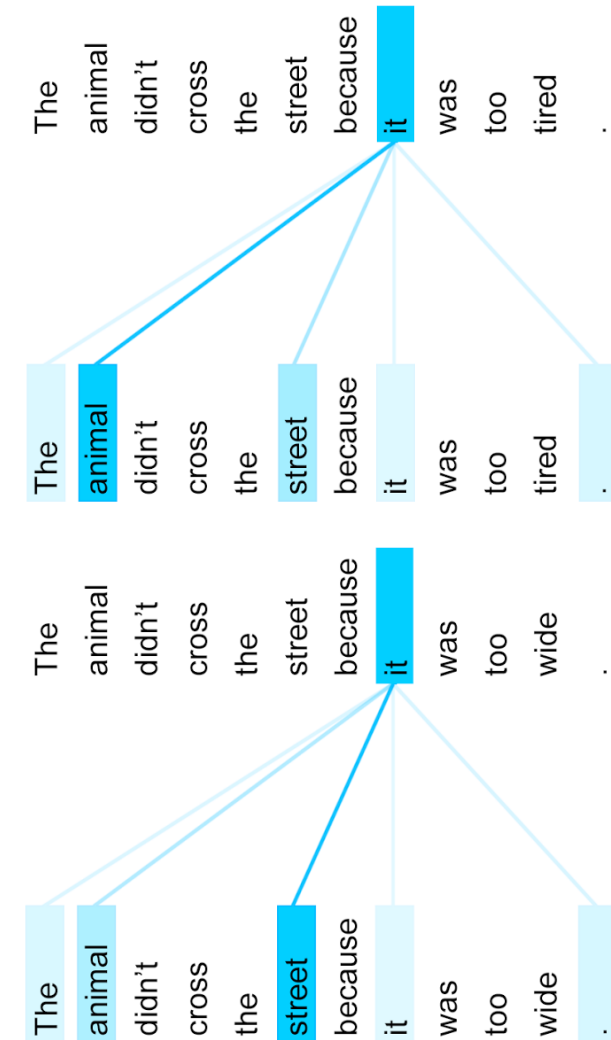
Attention: definition and motivation

Definition (**attention**):

In the context of neural networks, **attention** (as introduced in [Vaswani2017, Devlin2019]) is a technique that mimics cognitive attention.

The effect **enhances the important parts of the input data** and **fades out the rest**.

The thought being that **the network should devote more computing power on that small but important part** of the data.



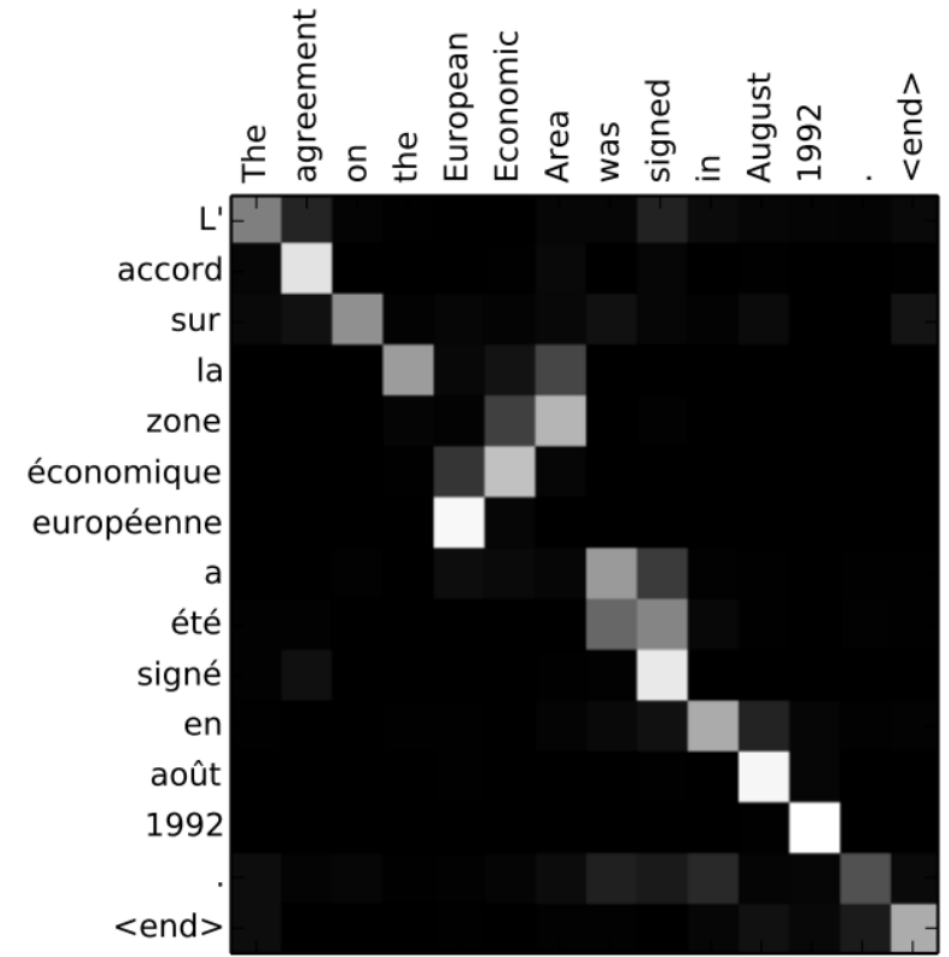
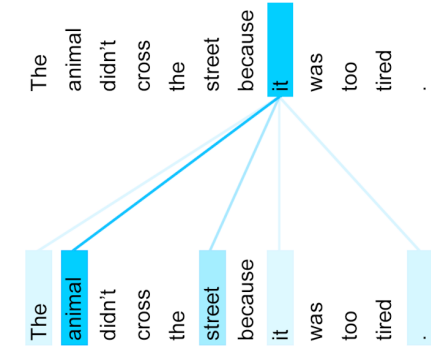
Attention coefficients

Definition (**attention coefficients**):

Attention coefficients identify which elements of the input sequence $x = (x_1, \dots, x_N)$ is relevant to compute an element y_k in the output sequence $y = (y_1, \dots, y_M)$.

$$a_{i,j} = \text{importance of } x_i \text{ wrt. } y_j$$

For instance, attention helps with shenanigans such as adjective-noun placement in English and French.



Attention coefficients

Definition (**attention coefficients**):

Attention coefficients identify which elements of the input sequence $x = (x_1, \dots, x_N)$ is relevant to compute an element y_k in the output sequence $y = (y_1, \dots, y_M)$.

$$a_{i,j} = \text{importance of } x_i \text{ wrt. } y_j$$

For instance, attention helps with shenanigans such as adjective-noun placement in English and French.

*“**Paris** was a universe whole and entire unto herself, hollowed and fashioned by history; so she seemed in this age of Napoleon III with her towering buildings, her massive cathedrals, her grand boulevards, and ancient winding medieval streets—as vast and indestructible as nature itself. All was embraced by her, by her volatile and enchanted **populace** thronging the galleries, the theaters, the cafes, giving birth over and over to genius and sanctity, philosophy and war, frivolity and the finest art; so it seemed that if all the world outside her were to sink into darkness, what was fine, what was beautiful, what was essential might there still come to its finest flower.”*

Attention: definition and motivation

Definition (**transformer**):

A **transformer** network make extensive use of **attention mechanisms** and more specifically **attention layers** to achieve their expressive power.

Transformers are currently considered as one of the most promising directions for Computer Vision, and Natural Language Processing.



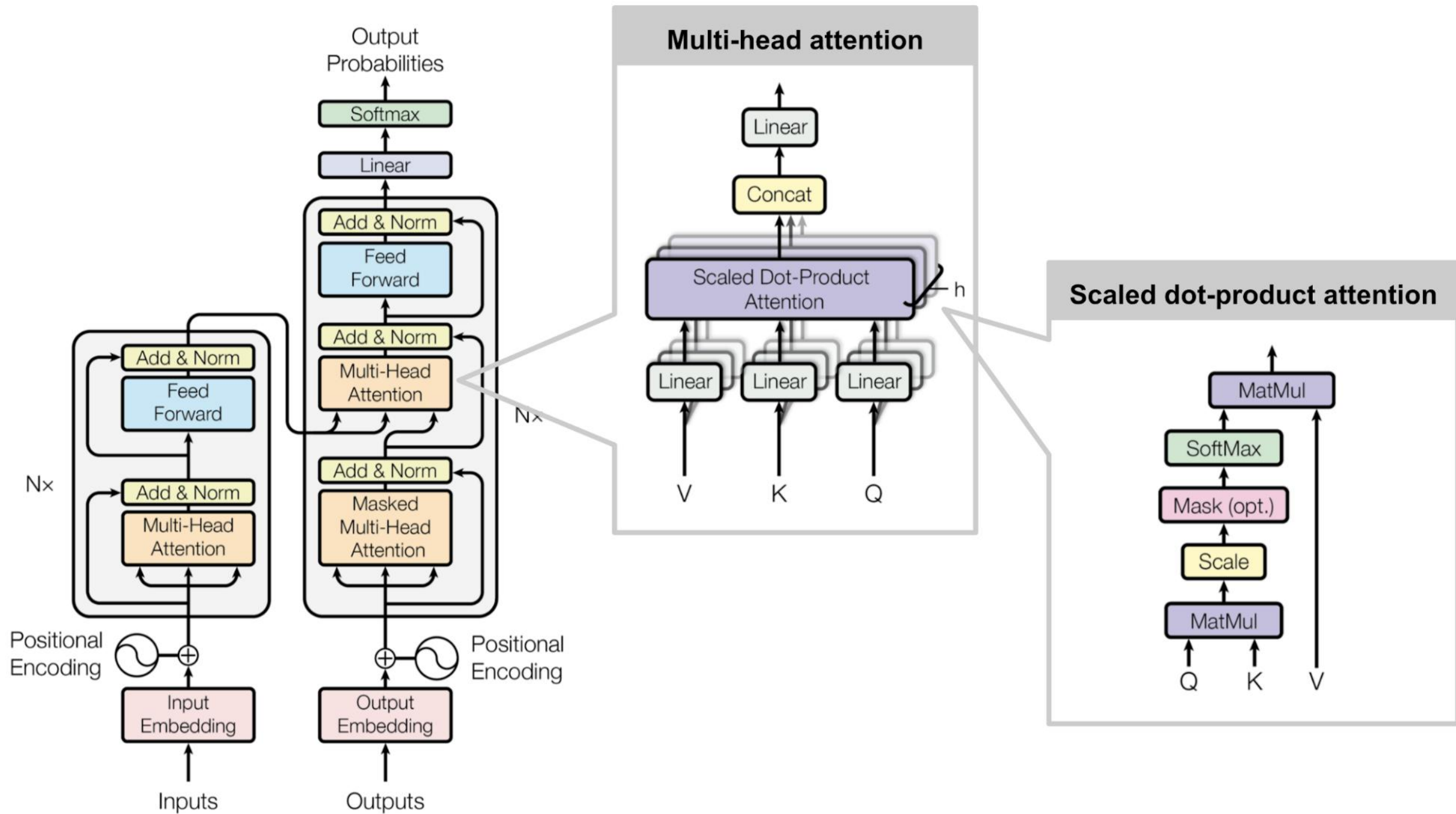
Don't you even think about it.
Seriously, don't.

The attention layer

Definition (the **transformer architecture):**

But first, a disclaimer from our beloved SUTD sponsor because transformers are, by far, the most difficult architecture we will see this term.





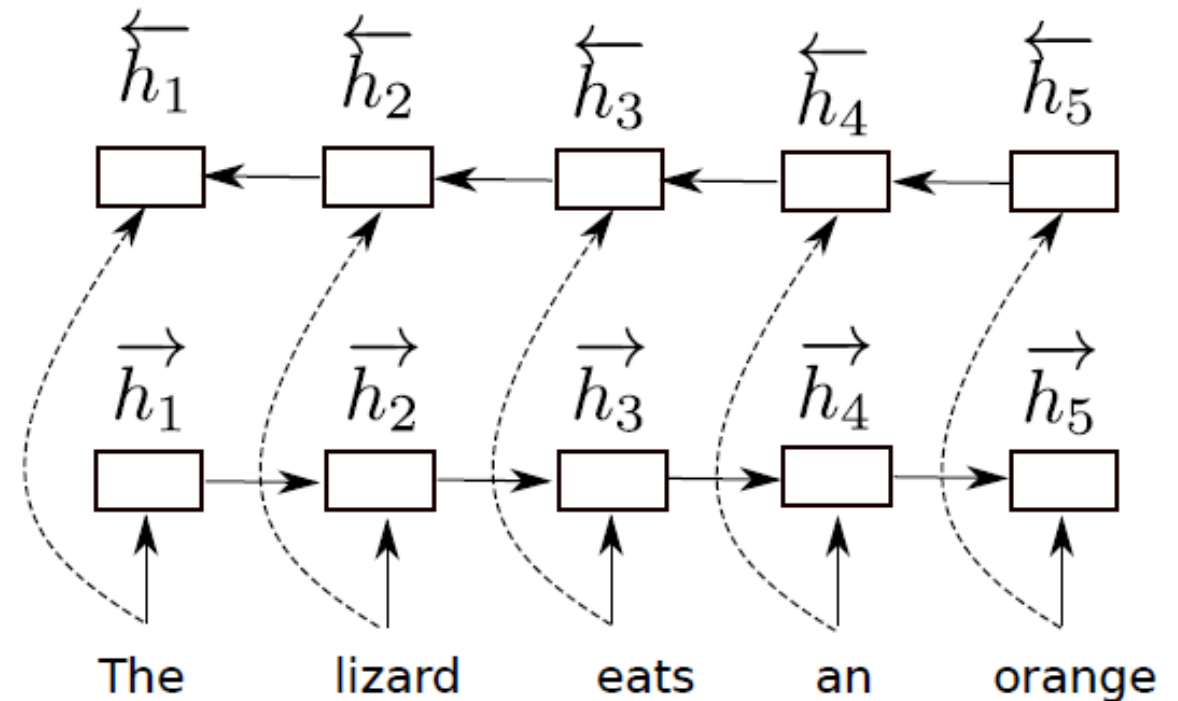
The attention layer

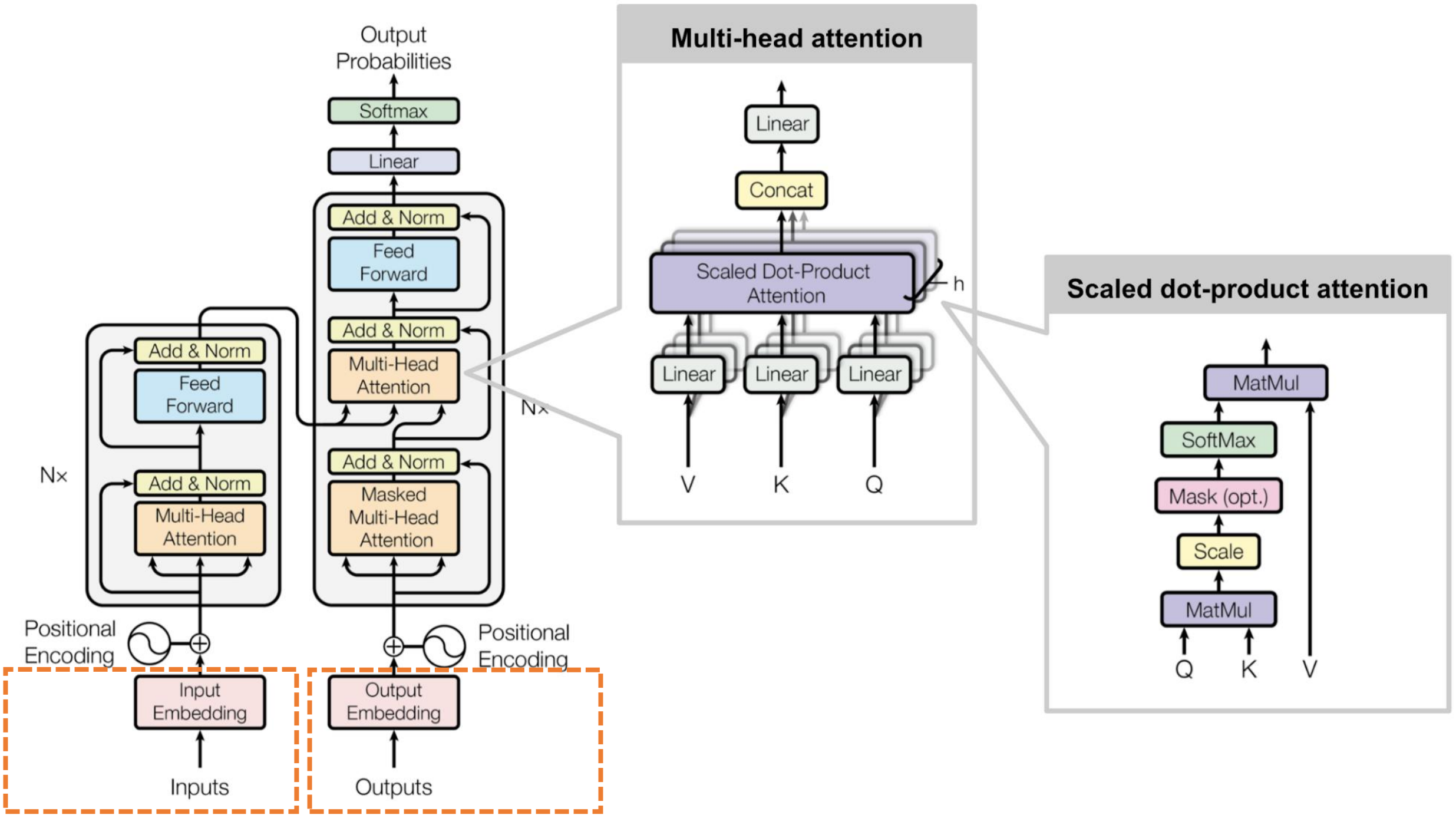
Definition (the **transformer architecture**):

1. It starts with an input embedding, which is used to propagate context between words of an input sequence.

Use a first **embedding vector** for any word w_k , using either another word embedding model (e.g. using ELMo), or by implementing a similar architecture.

$$h_k = \left[\overleftarrow{h}_k, \overrightarrow{h}_k \right]$$





The attention layer

Definition (the **transformer architecture**):

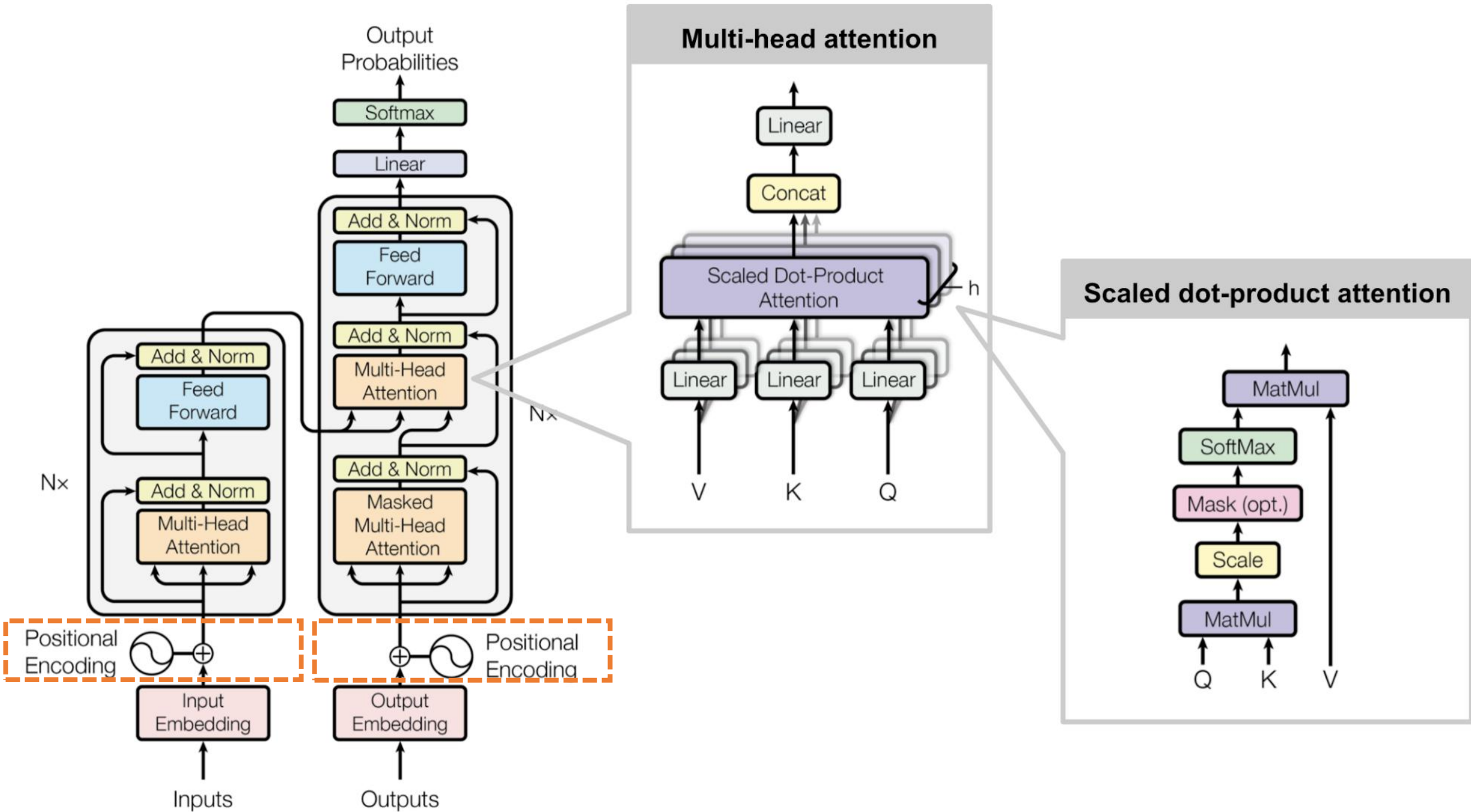
2. If you observe, the self-attention computations we will define later have no notion of the ordering of words among the sequences.

This is where **positional encoding** comes into play.

(However, out-of-scope for this class, will be skipped).

Core idea: Some languages have different positioning of words in sentences (Japanese tends to put verbs at the end of sentence, French tends to place adjectives after nouns, etc.)

To elicit this notion of positioning of words in the sequence, positional encoding can be added to the regular input embeddings.



The attention layer

Basically, we want a matrix A that contains coefficients $a_{i,j} \in [0,1]$, telling the proportion of the meaning of word v_j that needs to be transferred to word v_i to produce a more relevant embedding v'_i .

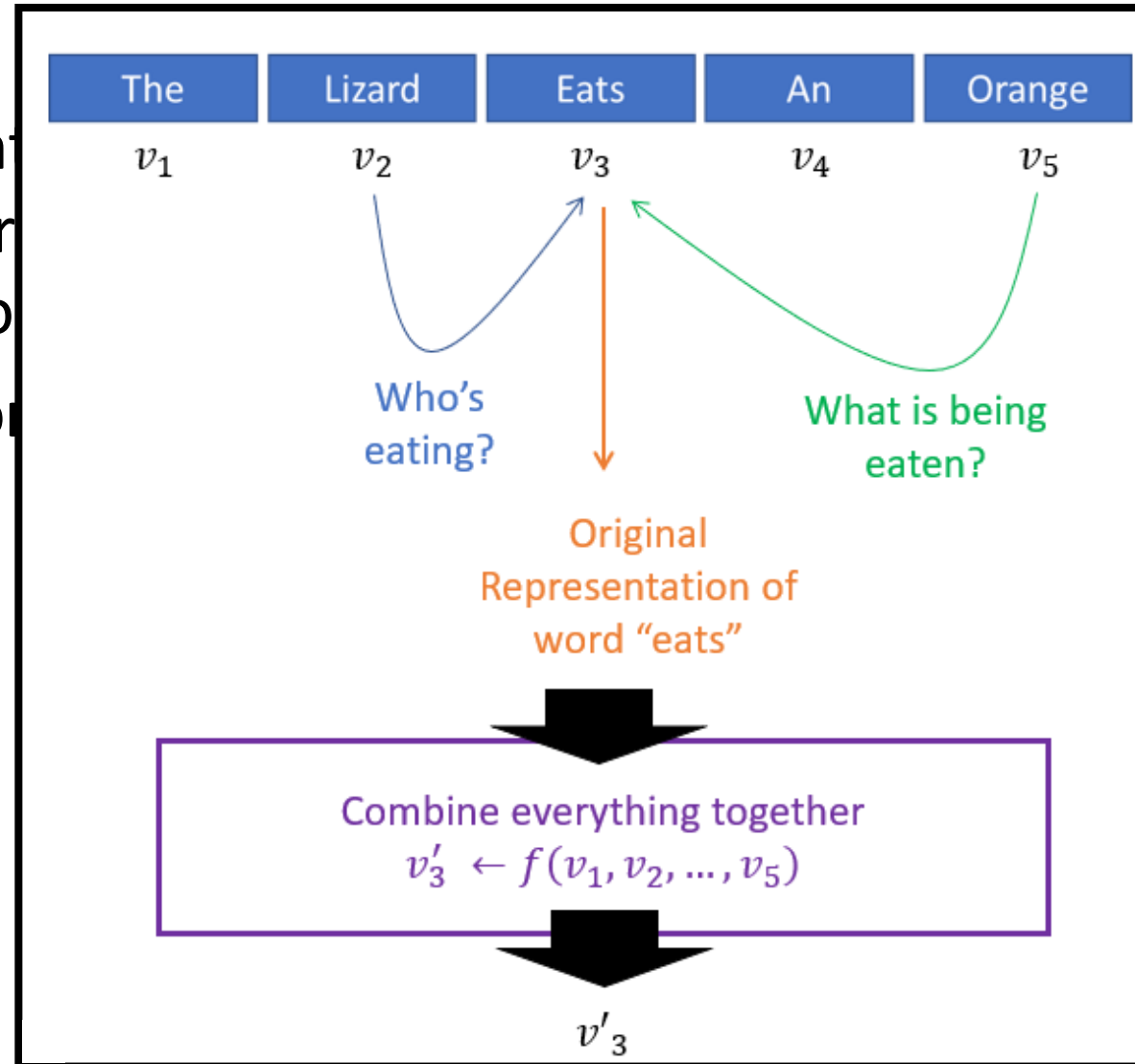
- **Before attention layer:** embeddings for word i is v_i
- **After attention layer:** new updated embeddings for word i is v'_i

$$v'_i = \sum_j a_{i,j} v_j$$

The attention layer

Basically, we want
telling the proportion
transferred to word

- **Before attention**
- **After attention**



The attention layer

Definition (the multi-head self-attention layer a.k.a. the **attention layer):**

3. Getting vectors V, K, Q

$q_i \in Q$ is the query, i.e. the word I am currently looking at.

V and K refer to values and keys.

For simplicity, we will assume that

$$V = K = Q = \text{words in seq.}$$

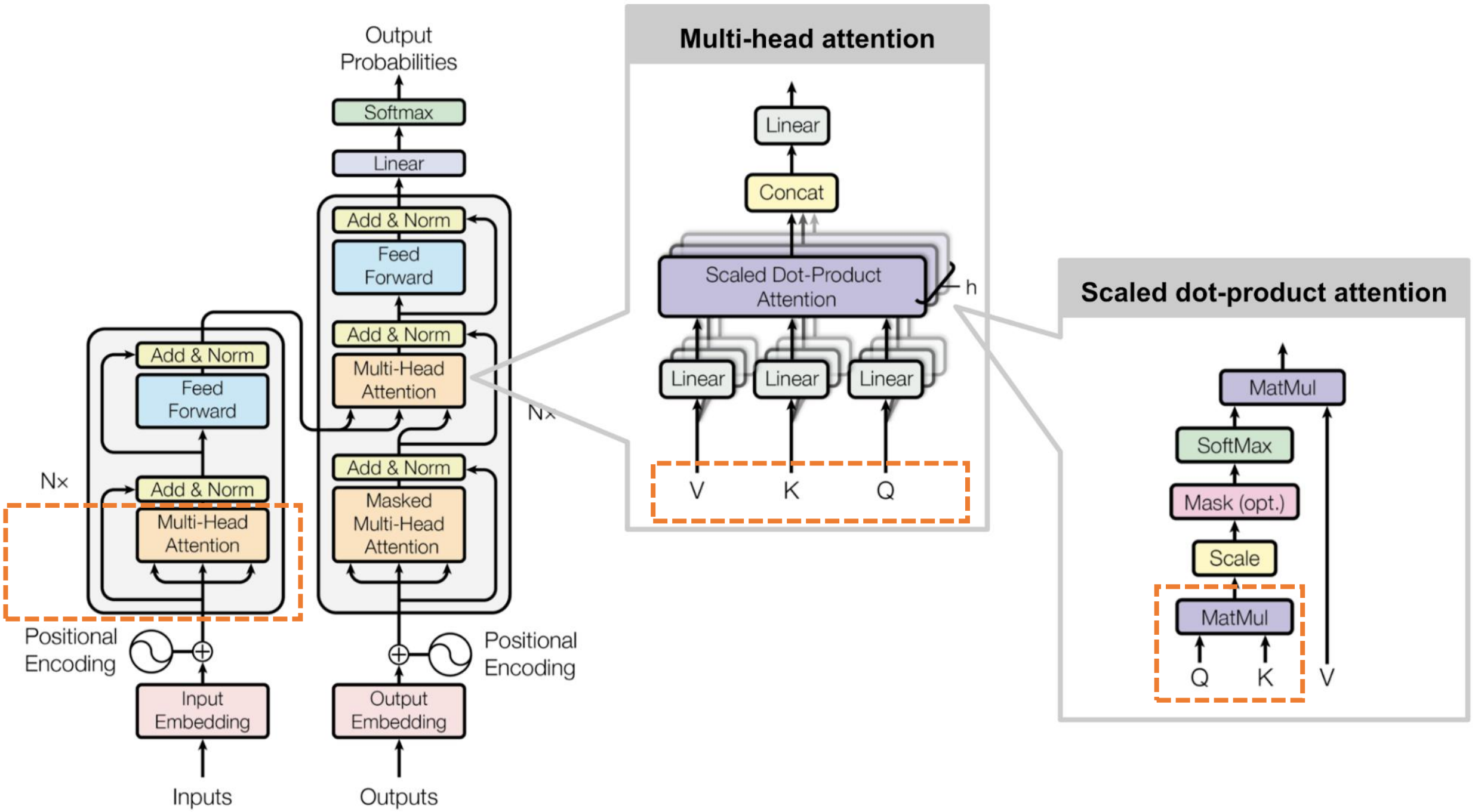
Let us consider the sentence

“Paris is the capital city of France.”

When I multiply

- the context vector of the word “city” (i.e. one query word q_i),
- with any other word of the sentence, e.g. “Paris” (as one key word k_j),
- we obtain a similarity score $s_{i,j}$, as in the dot-product/cosine angle formula from earlier.
- Repeat for all pairs of query/key words.

$$S = QK^T$$



The attention layer

Applying a softmax gives you a distribution over these similarities

Definition (the multi-head self-attention layer a.k.a. the **attention layer):**

3. Getting vectors V, K, Q

Q is the query, i.e. the words I am currently looking at.

V and K refer to values and keys.

In general, we also like to normalize the QK^T multiplication, as in the cosine angle.

For simplicity, we will assume that

$$V = K = Q = \text{words in seq.}$$

$$S = \frac{QK^T}{d}$$
$$S_2 = \text{softmax}(S)$$

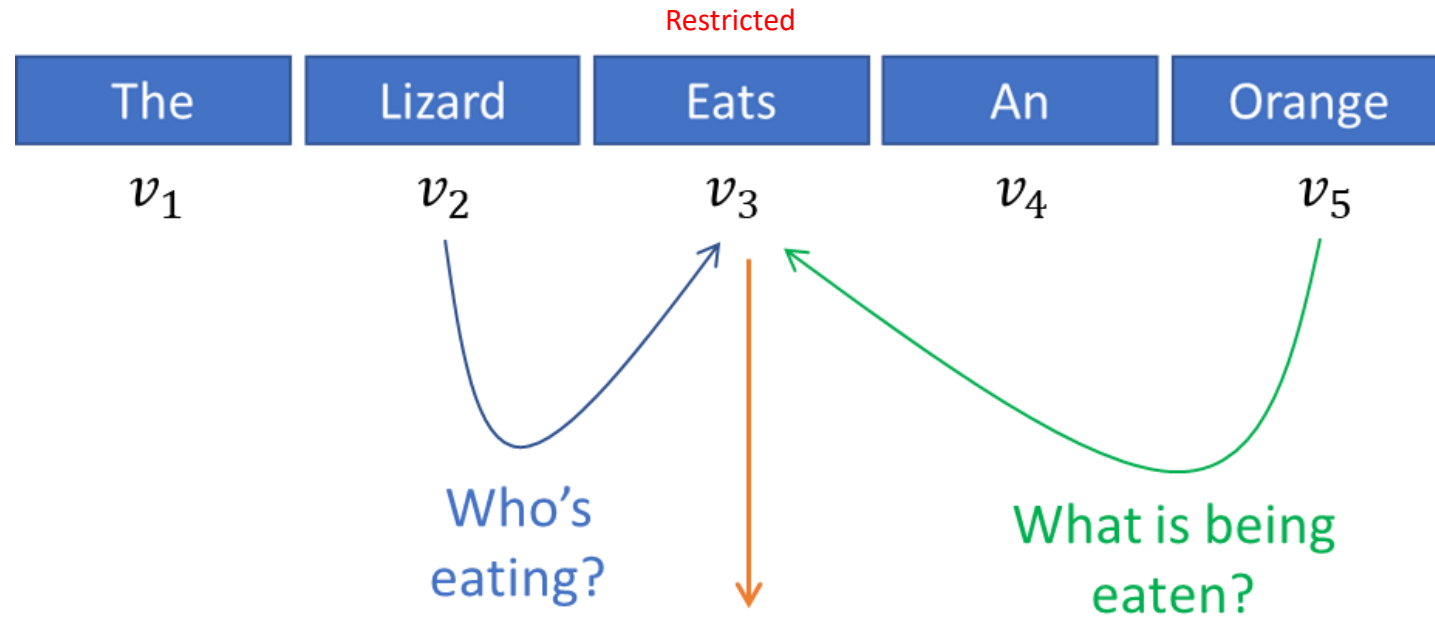
The attention layer

Basically, we obtained a matrix A (or S_2 here) that contains coefficients $a_{i,j} \in [0,1]$, telling the proportion of the meaning of word v_j that needs to be transferred to word v_i to produce a more relevant embedding v'_i .

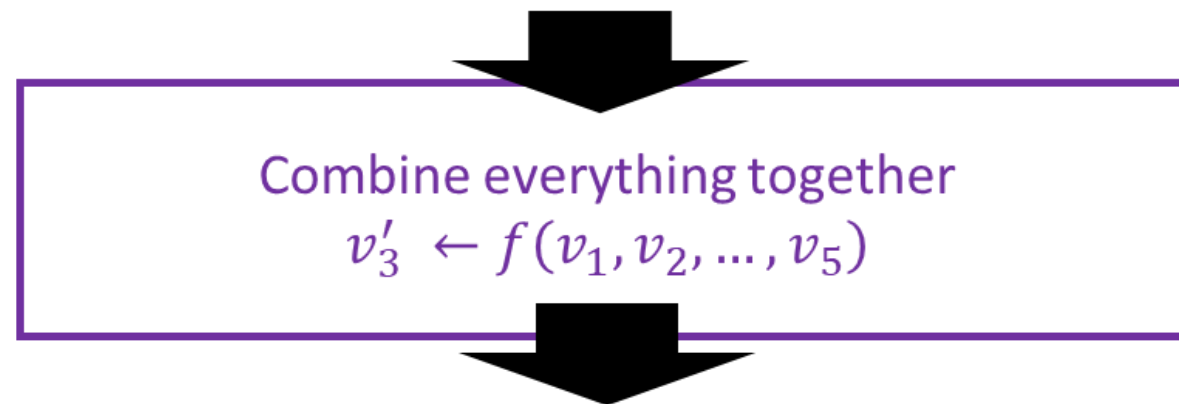
Intuitively, it consists of all dot products between all pairs of vectors, normalized, somewhat similar to the cosine angle from earlier?

- **Before attention layer:** embeddings for word i is v_i
- **After attention layer:** new updated embeddings for word i is v'_i

$$v'_i = \sum_j a_{i,j} v_j$$



Original
Representation of
word "eats"



Restricted

The attention layer

Definition (the multi-head self-attention layer a.k.a. the **attention layer):**

3. Getting vectors V, K, Q

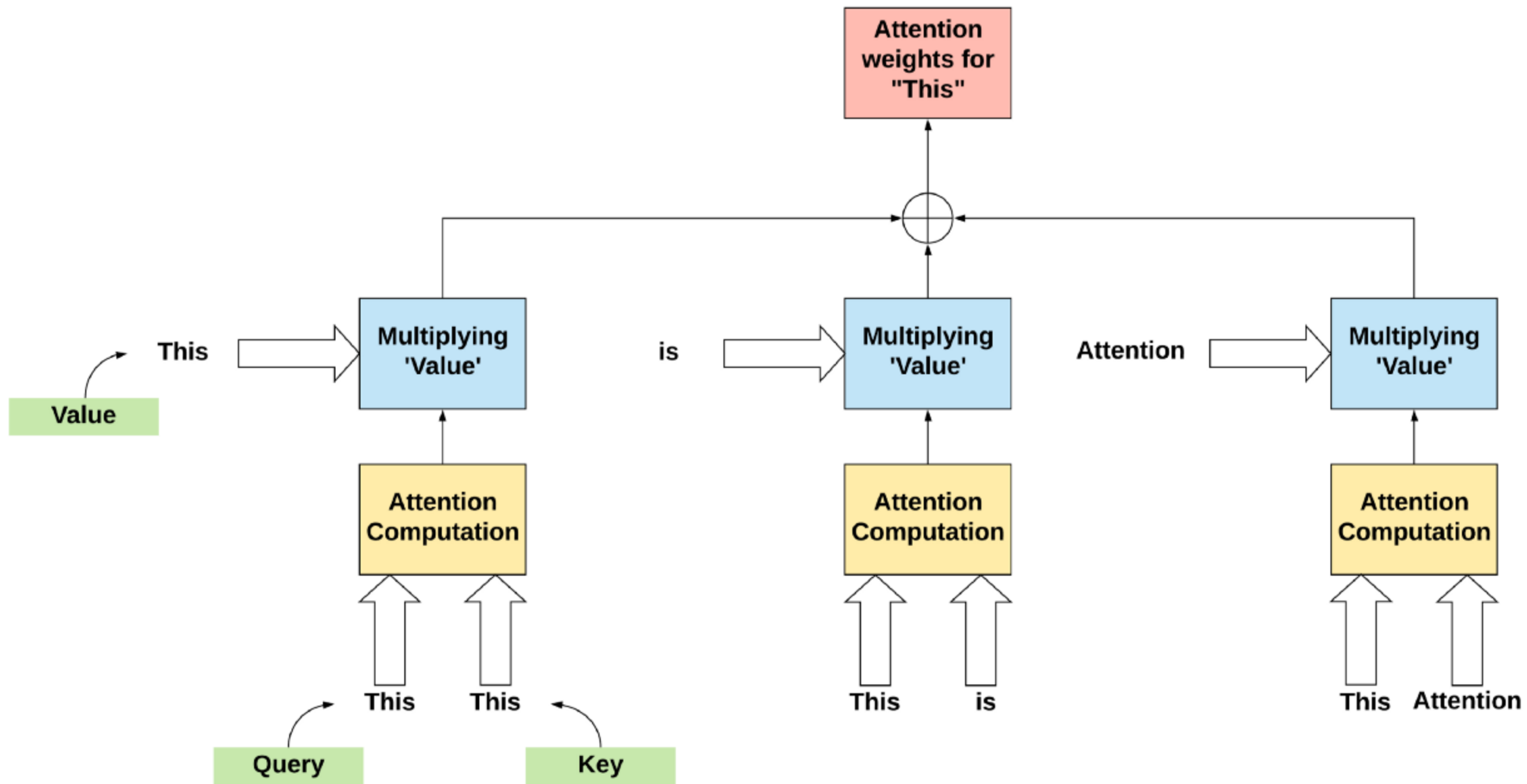
Q is the query, i.e. the word I am currently looking at.

V and K refer to values and keys, and for simplicity will first consist of all the words you have in your sentence.

Multiply this vector of distribution with the words in V , and you obtain a matrix of attention V' !

$$V' = \text{softmax}\left(\frac{QK^T}{d}\right)V$$

This matrix V' is an updated version of V , which encapsulates the inter-dependencies between the words of your sentence.



The attention layer

Definition (the multi-head self-attention layer a.k.a. the **attention layer):**

3. Getting vectors V, K, Q

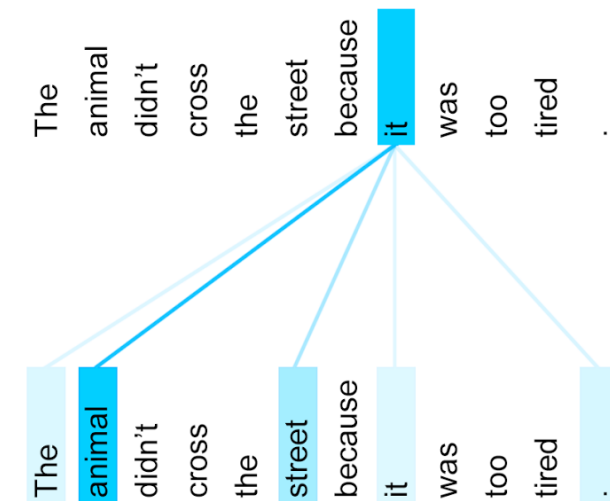
Q is the query, i.e. the words I am currently looking at.

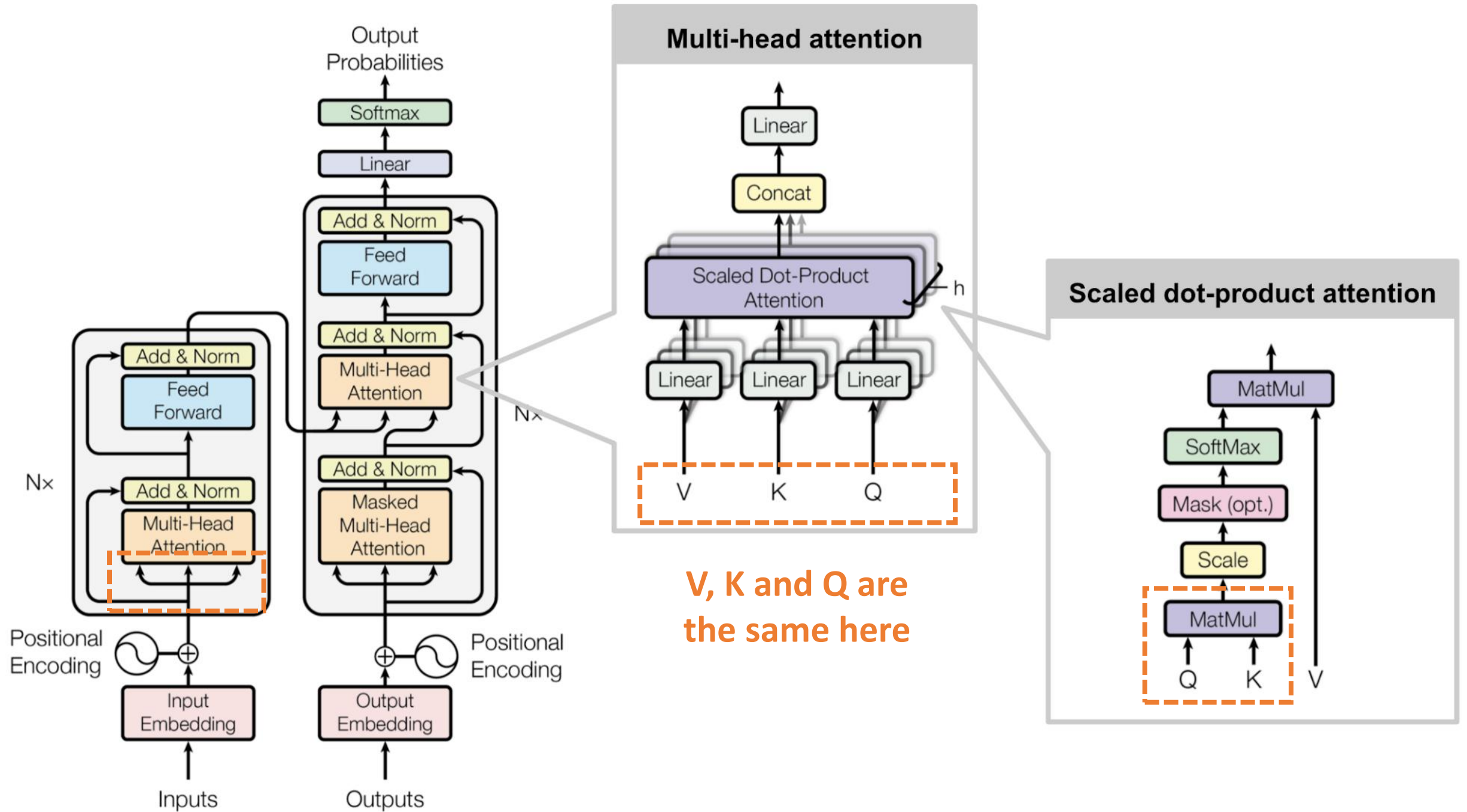
V and K refer to values and keys.

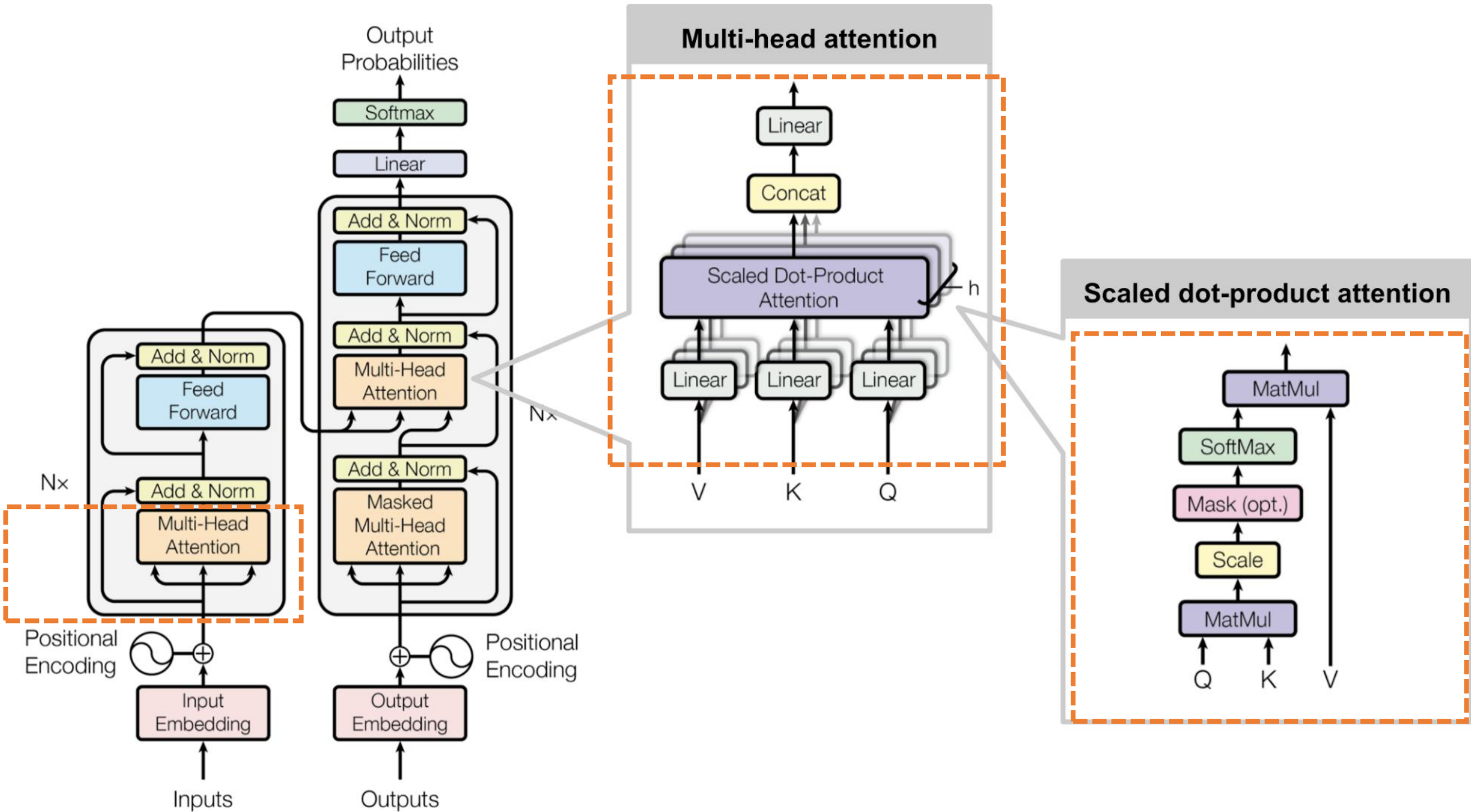
For simplicity, we will assume that

$$V = K = Q = \text{words in seq.}$$

This calculation is often referred to as **self-attention**, as it gives inter-dependencies between the words of a single sentence.





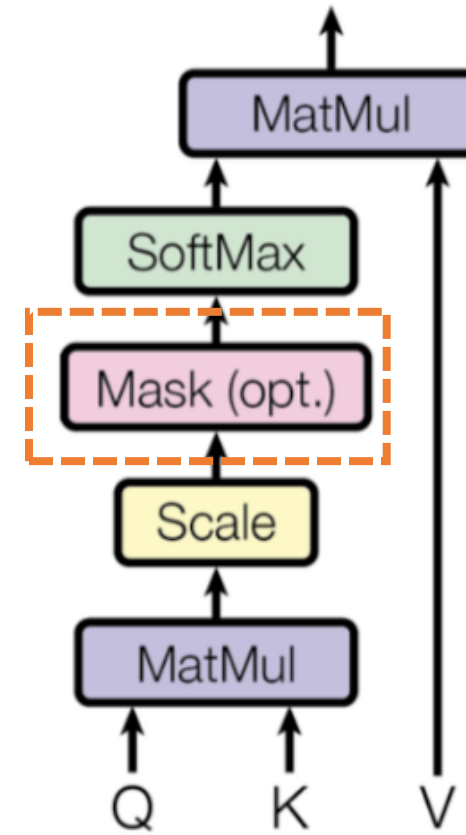


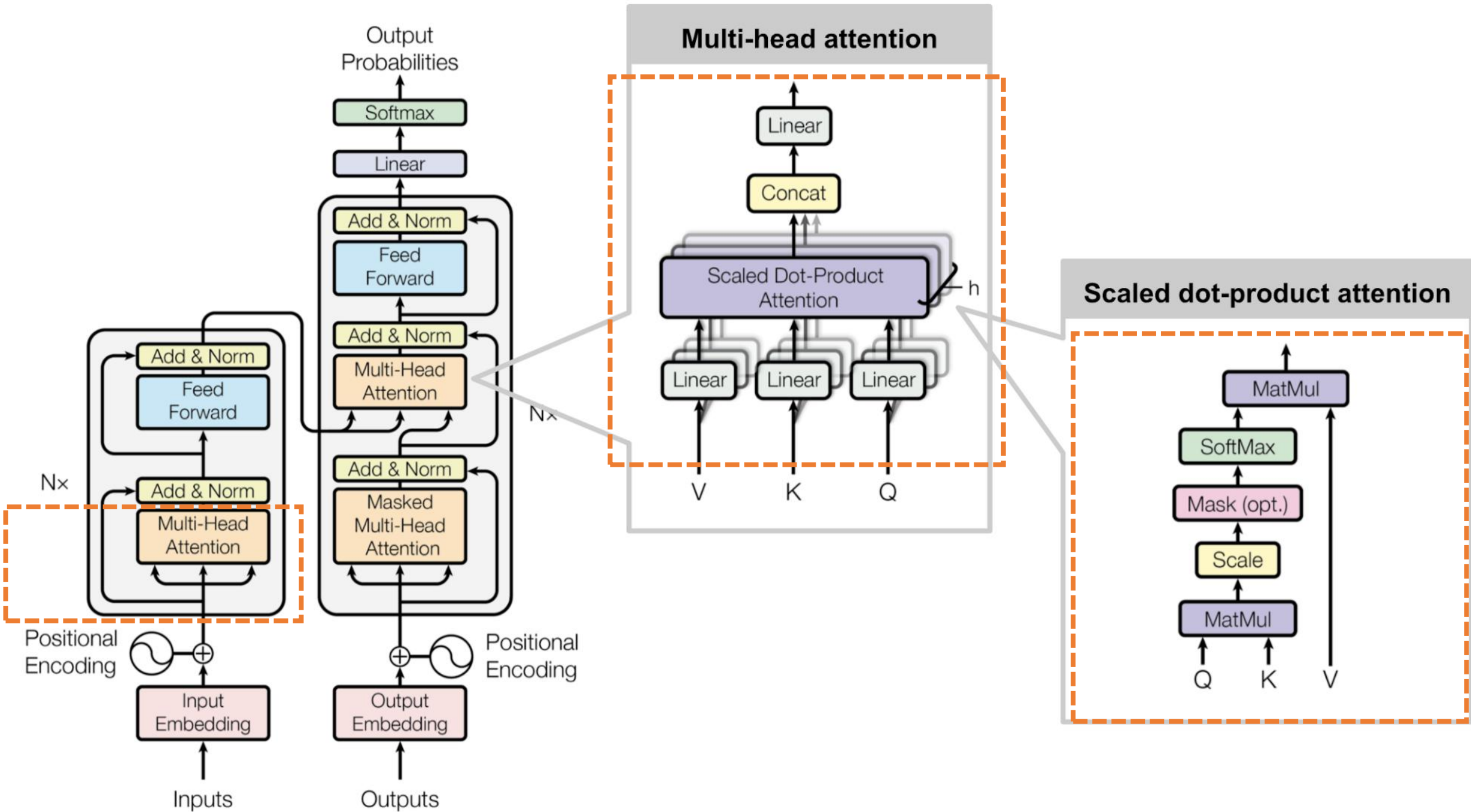
The attention layer

Definition (the multi-head self-attention layer a.k.a. the **attention layer):**

4. **Masking** is only used in the decoder part (and it will be discussed later).

Scaled dot-product attention





```

1  class SelfAttention(nn.Module):
2      """
3      Description: SelfAttention layer Class, describing the attention layers to
4      be used in the transformer.
5      Attributes list:
6      - d_model: An integer, defining the dimension of the attention layer
7      - output_size: An integer, defining the dimension of the output
8      for the attention layer.
9      - dropout_rate: A float value between 0 and 1, corresponding to the Dropout rate
10     used in the Dropout layers of the Feed Forward layers.
11     - mask: A boolean. If set to True, a triangular mask will be applied.
12     """
13
14     def __init__(self, d_model, output_size, dropout_rate = 0.3, mask = None):
15         """
16         Init Method for attention layer, mostly defining attributes.
17         """
18         super().__init__()
19         self.query = nn.Linear(d_model, output_size)
20         self.key = nn.Linear(d_model, output_size)
21         self.value = nn.Linear(d_model, output_size)
22         self.dropout = nn.Dropout(dropout_rate)
23         self.mask = mask

```



```
24
25 def forward(self, q, k, v):
26     """
27     Forward pass for the attention layer.
28     """
29
30     # Compute query, key and value parameters
31     q_shape = q.shape[0]
32     y_len = q.shape[1]
33     seq_len = k.shape[1]
34     query_out = self.query(q)
35     key_out = self.key(k)
36     value_out = self.value(v)
37     key_dim = key_out.size(-1)
38     transposed_key_out = key_out.transpose(1, 2)
39
40     # Calculate attention scores
41     scores = torch.bmm(query_out, transposed_key_out) / np.sqrt(key_dim)
42
```

(masking part to be discussed
later)

```
56
57     # Final touches
58     weights = F.softmax(scores, dim = -1)
59     out = torch.bmm(weights, value_out)
60     return out
```



```

1  class MultiHeadAttentionLayer(nn.Module):
2      """
3      Description: MultiHeadAttention layer Class, describing the attention layers to
4      be used in the transformer.
5      Attributes list:
6      - d_model: An integer, defining the dimension of the attention layer
7      - num_heads: An integer, defining the number of attention heads
8      to be used in both the encoder and decoder layers.
9      - dropout_rate: A float value between 0 and 1, corresponding to the Dropout rate
10     used in the Dropout layers of the Feed Forward layers.
11     - mask: A boolean. If set to True, a triangular mask will be applied.
12     """
13
14     def __init__(self, d_model, num_heads, dropout_rate, mask = None):
15         """
16         Init method, mostly defining attributes.
17         """
18         super().__init__()
19         self.d_model = d_model
20         self.num_heads = num_heads
21         self.dropout_rate = dropout_rate
22         self.attention_output_size = d_model/num_heads
23         layers = [SelfAttention(d_model, self.attention_output_size, dropout_rate, mask) \
24                     for i in range(num_heads)]
25         self.attention_layers = nn.Module(layers, )
26         self.final_layer = nn.Linear(d_model, d_model)

```

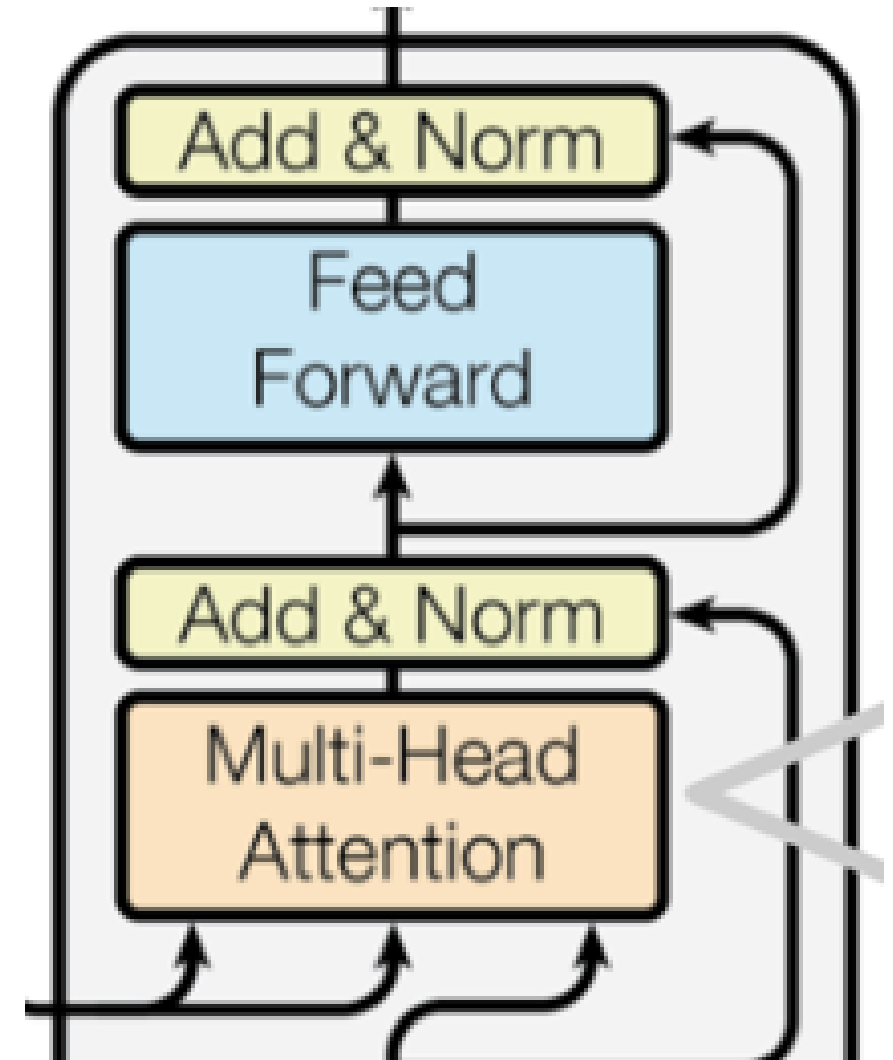
```
27
28     def forward(self, q, k, v):
29         """
30         Forward pass, simply going through each attention layer.
31         Applying one linear for final touch.
32         """
33         x = torch.cat([layer(q, k, v) for layer in self.attention_layers], dim = -1)
34         x = self.final_layer(x)
35         return x
```

The attention layer

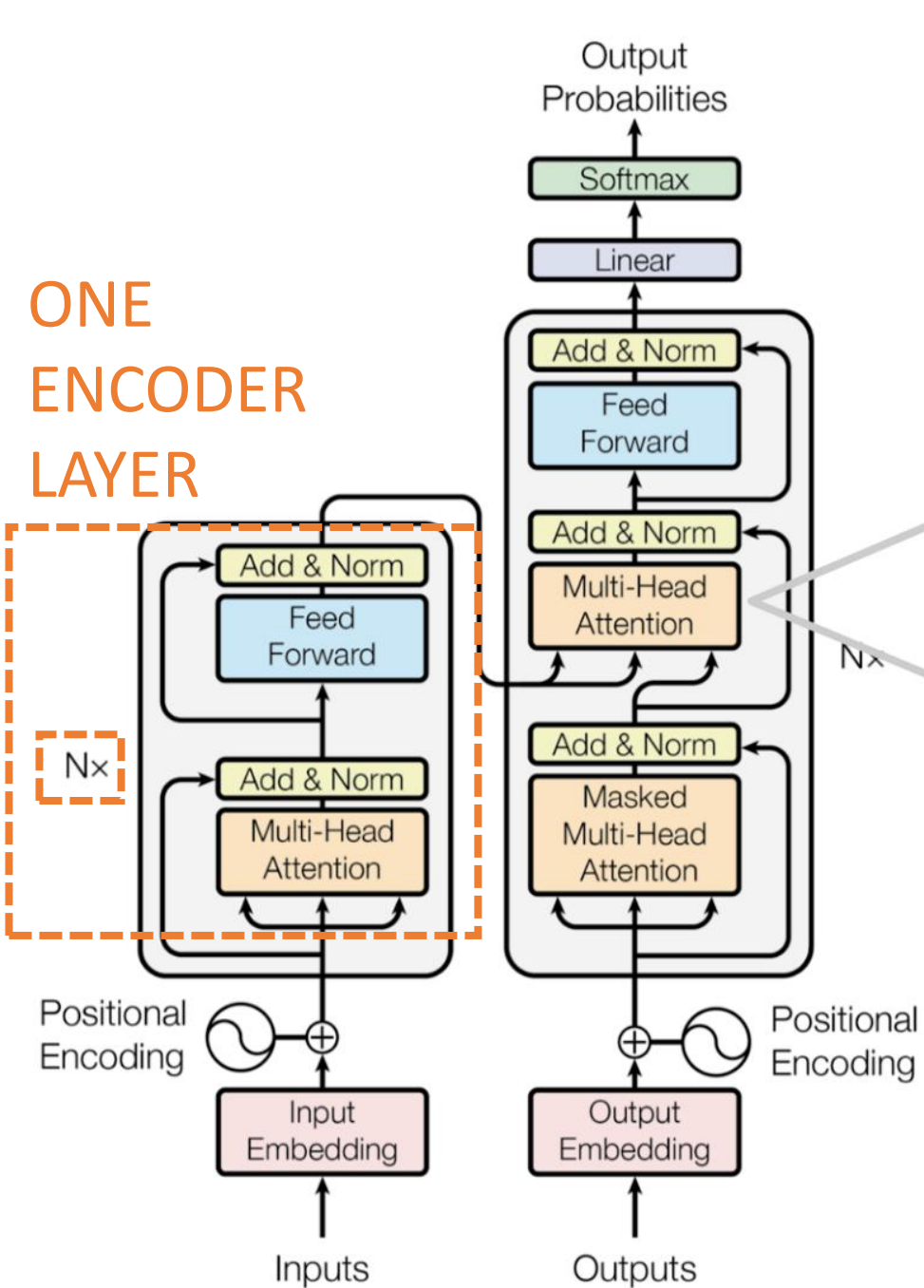
Definition (the multi-head self-attention layer a.k.a. the **attention layer):**

5. Feedforward is a set of linear/fully connected layers, whose main purpose is to allow for smooth transitions and reshape the attention vectors so that it is acceptable by the next encoder or decoder layer.

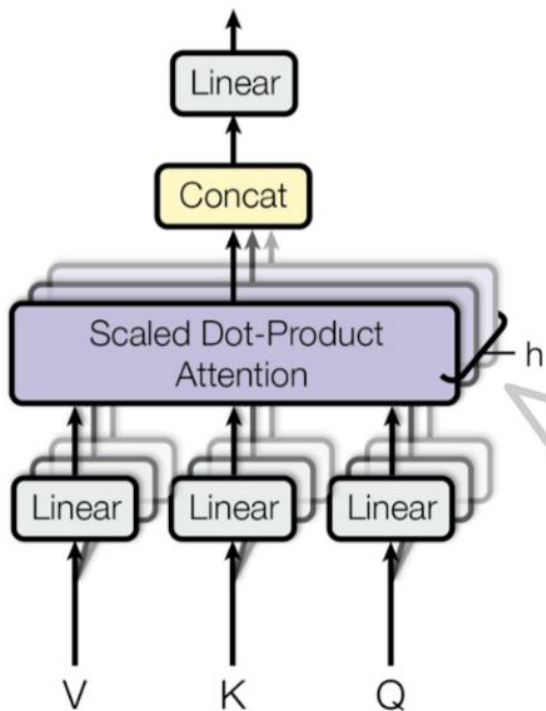
Also uses a skip/residual connection as in ResNet!



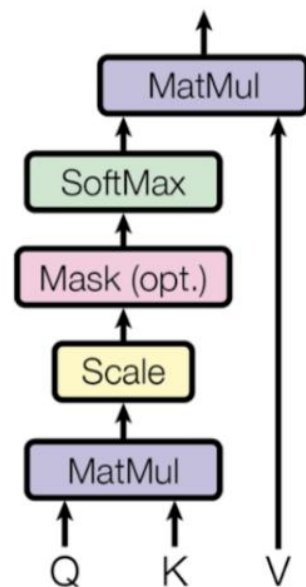
ONE ENCODER LAYER



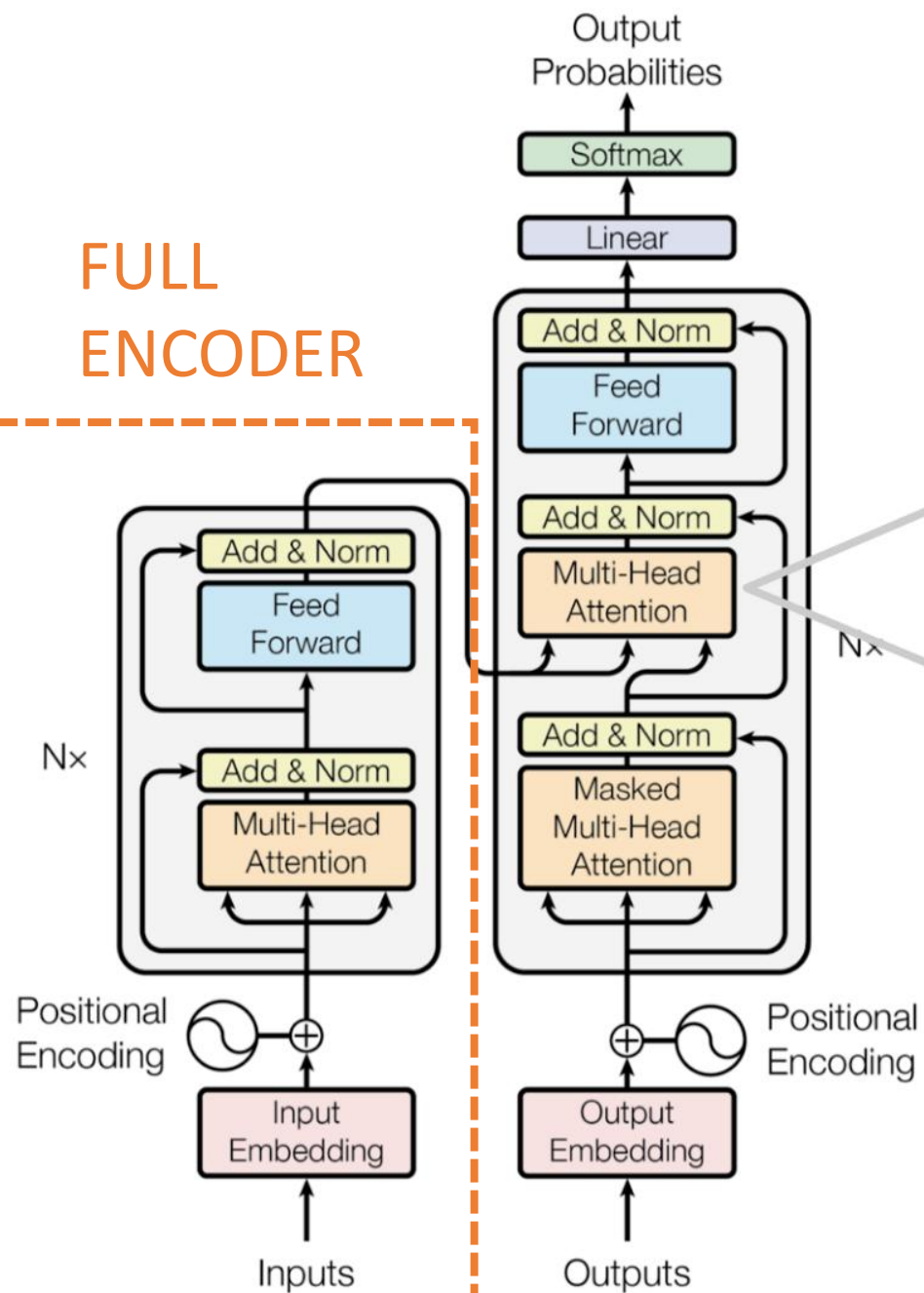
Multi-head attention



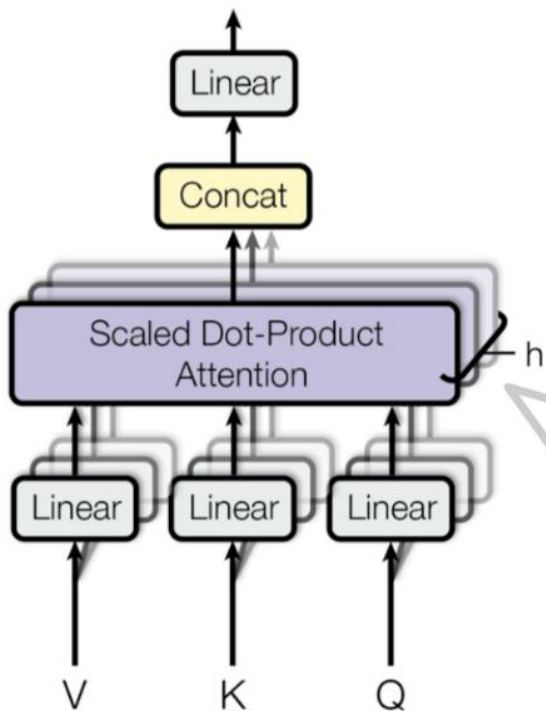
Scaled dot-product attention



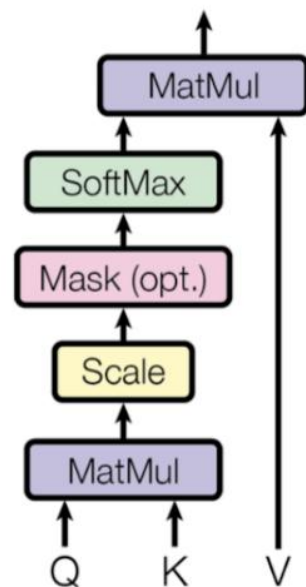
FULL ENCODER



Multi-head attention



Scaled dot-product attention



```
1 class EncoderLayer(nn.Module):
2     """
3     Description: EncoderLayer Class, describing the encoder layers to
4     be used in the Encoder part of the transformer.
5     Attributes list:
6     - d_model: An integer, defining the dimension of the encoder layer.
7     - num_heads: An integer, defining the number of attention heads
8     to be used in both the encoder and decoder layers.
9     - d_inner: An integer, defining the number of neurons in the Feed Forward layers.
10    - drouput_rate: A float value between 0 and 1, corresponding to the Dropout rate
11    used in the Dropout layers of the Feed Forward layers.
12    """
13
```

```

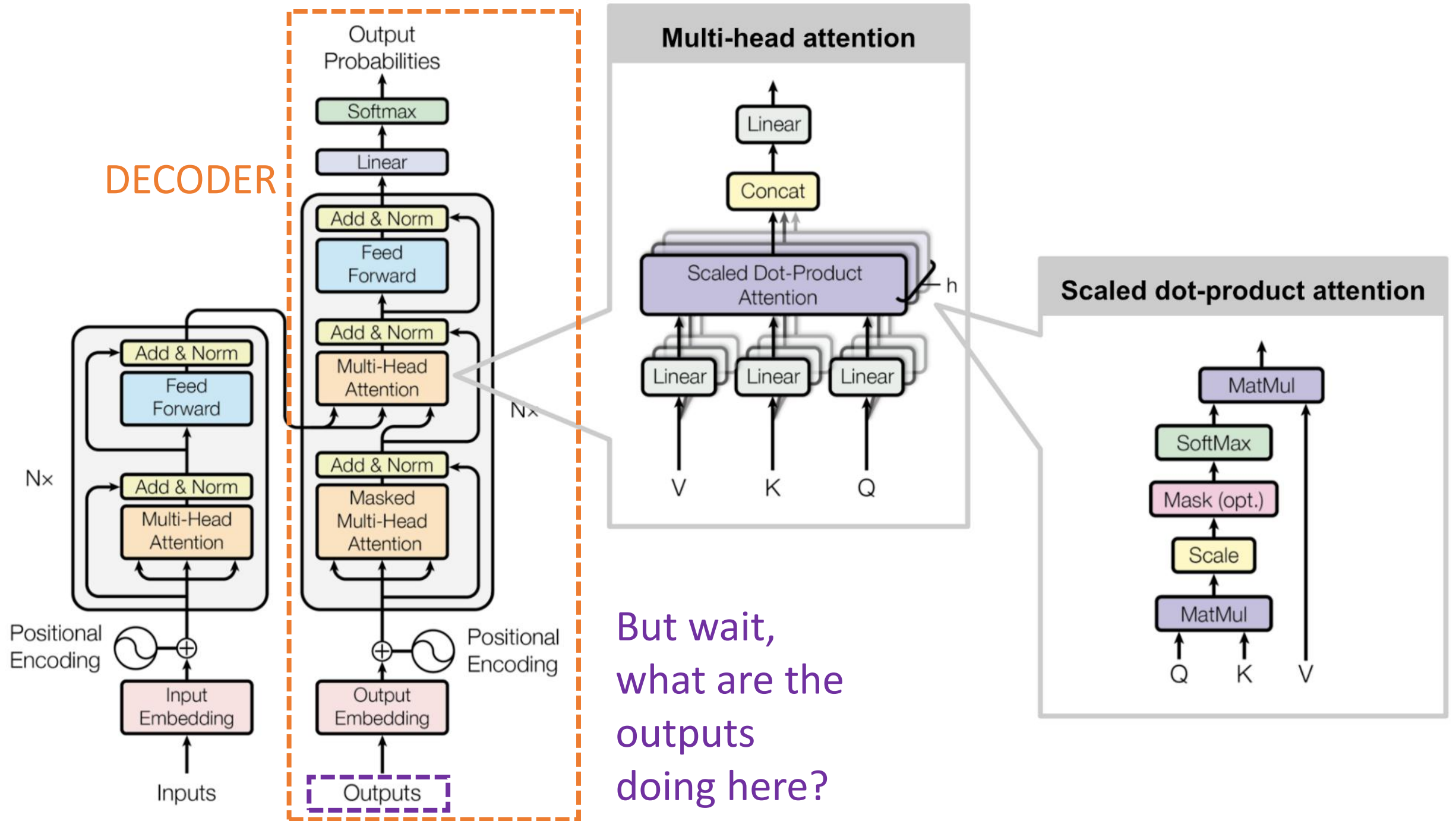
13
14 def __init__(self, d_model, num_heads, d_inner = 2048, dropout_rate = 0.3):
15     """
16     Init Method, which adds up
17     - one MultiHeadAttentionLayer, with an expanded mask,
18     - one Feed Forward layer,
19     - and Normalization Layers after each one of them.
20     """
21     super().__init__()
22     # First Multi Head Attention Layer
23     self.attention = MultiHeadAttentionLayer(d_model, num_heads, dropout_rate)
24     # Feed Forward Network Layer
25     self.ffn = nn.Sequential(nn.Linear(d_model, d_inner),
26                               nn.ReLU(inplace = True),
27                               nn.Dropout(dropout_rate),
28                               nn.Linear(d_inner, d_model),
29                               nn.Dropout(dropout_rate),)
30     # Normalization Layers
31     # (Roughly identical to BatchNorm)
32     self.attention_normalization = nn.LayerNorm(d_model)
33     self.ffn_normalization = nn.LayerNorm(d_model)

```

```
34
35     def forward(self, x):
36         """
37         Forward pass for the encoder layer.
38         """
39         # Attention Layer forward pass and normalization
40         out = x + self.attention(q = x,
41                                   k = x,
42                                   v = x)
43         out = self.attention_normalization(out)
44         # Feed Forward Network Layer forward pass and normalization
45         out = out + self.ffn(out)
46         out = self.ffn_normalization(out)
47         return out
```



```
1 class Encoder(nn.Module):
2     """
3     Description: Encoder Class, describing the Encoding part of the transformer.
4     Attributes list:
5     - d_model: An integer, defining the dimension of the encoder layers.
6     - num_heads: An integer, defining the number of attention heads
7     to be used in both the encoder and decoder layers.
8     - num_encoders: An integer, defining the number of encoder layers
9     to be used in the transformer.
10    """
11
12    def __init__(self, d_model, num_heads, num_encoders):
13        """
14        Init Method, which adds up num_encoders EncoderLayers in a row.
15        """
16        super().__init__()
17        layers_list = [EncoderLayer(d_model, num_heads) for i in range(num_encoders)]
18        self.enc_layers = nn.ModuleList(layers_list, )
19
20    def forward(self, x):
21        """
22        Forward pass for the Encoder, simply repeating the EncoderLayers in a row.
23        """
24        out = x
25        for layers in self.enc_layers:
26            out = layer(out)
27        return out
```



The attention layer

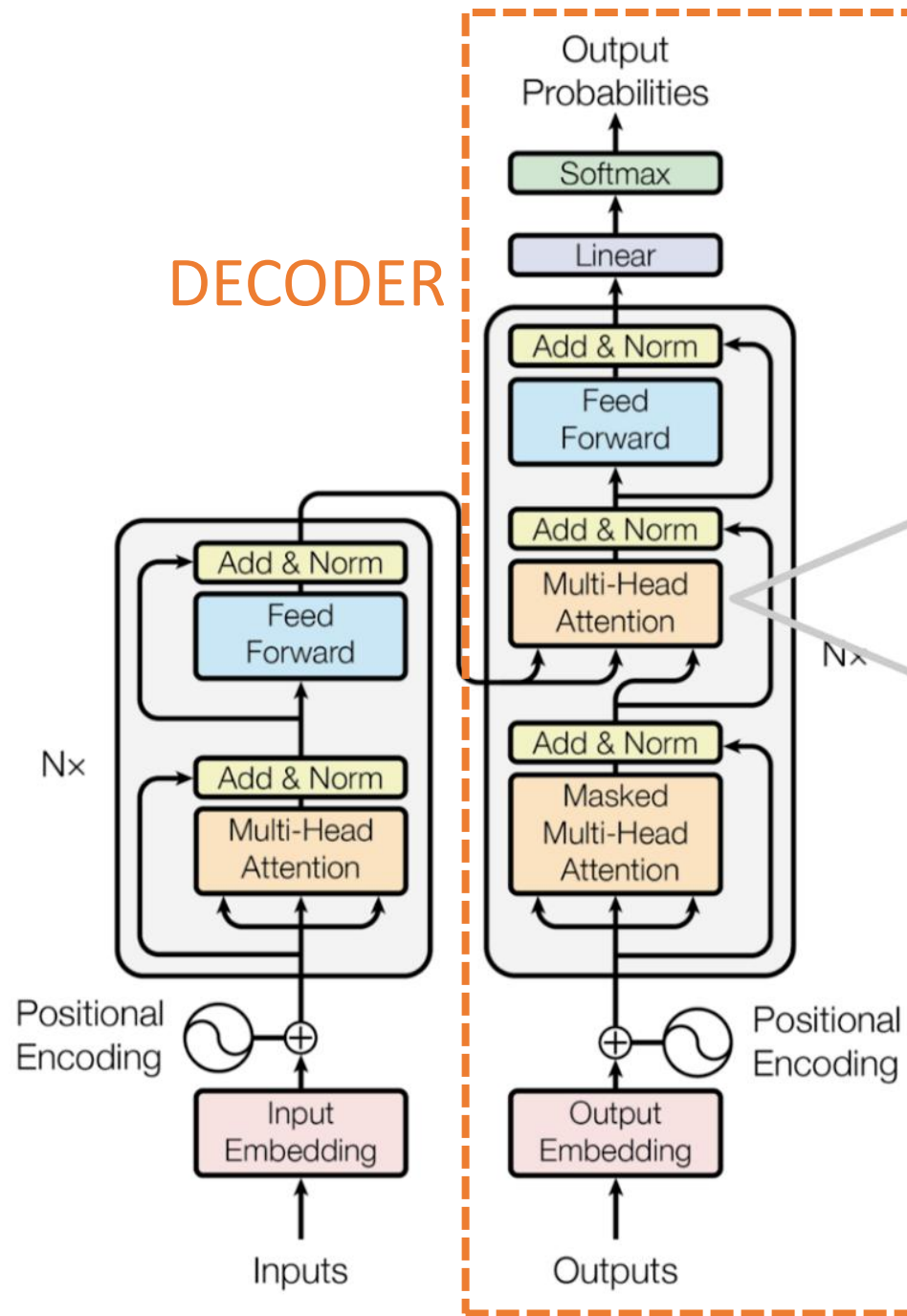
Important question: Why are the outputs used both as inputs and targets for the decoder model?

Our objective is to train a good word embedding (our encoder), by using an encoder-decoder architecture on some NLP task.

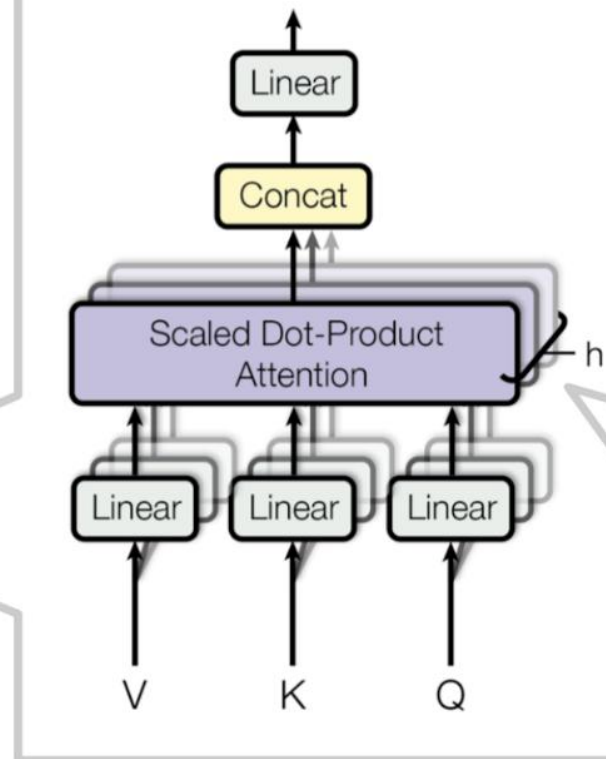
At the end of the day, the performance of this Seq2Seq model is irrelevant, as we only care about the embedding (as in SkipGram).

So, who cares if we “cheat” and give the answer to the decoder model?

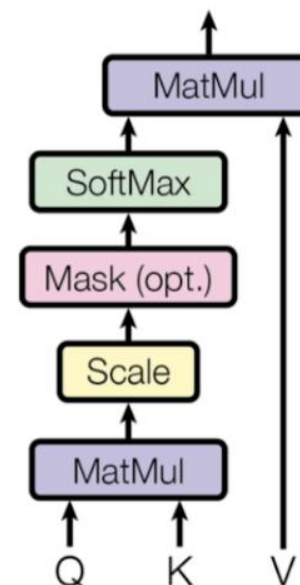
DECODER



Multi-head attention



Scaled dot-product attention



The attention layer

Definition (the multi-head self-attention layer a.k.a. the **attention layer):**

6. Same multi-head block as before but with **masking** this time!

Why masking?

While generating target sequences at the decoder, since the transformer uses self-attention, it tends to include all the words from the decoder inputs. But, practically, this is incorrect and too much “cheating” for the network.

The transformer must not use the current or future output to predict the next one, so the output sequence must be partially masked to prevent this reverse information flow.

Masking Multi-Head Attention ensures this.

The attention layer

Definition (the multi-head self

attention

attention

6. S Encoder

b
t

v

Decoder

The → The big red dog
big → The big red dog
red → The big red dog
dog → The big red dog

Le → Le gros chien rouge
gros → Le gros chien rouge
chien → Le gros chien rouge
rouge → Le gros chien rouge

Masked Input

es at
er
clude

Only
word
of

sures

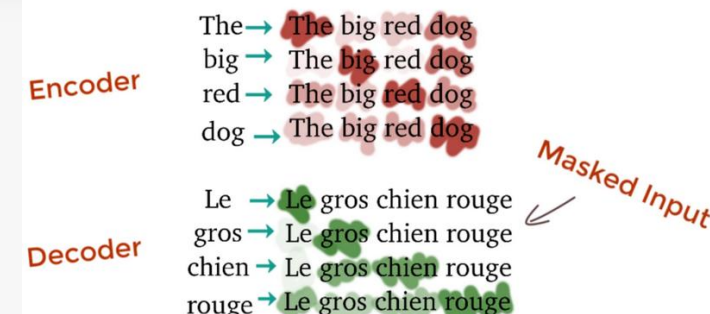
(Extra step in the attention layer class, forward method!)

Note: I played around with a few different masks, just for fun. Paper suggests to use the triu (upper triangular) mask.

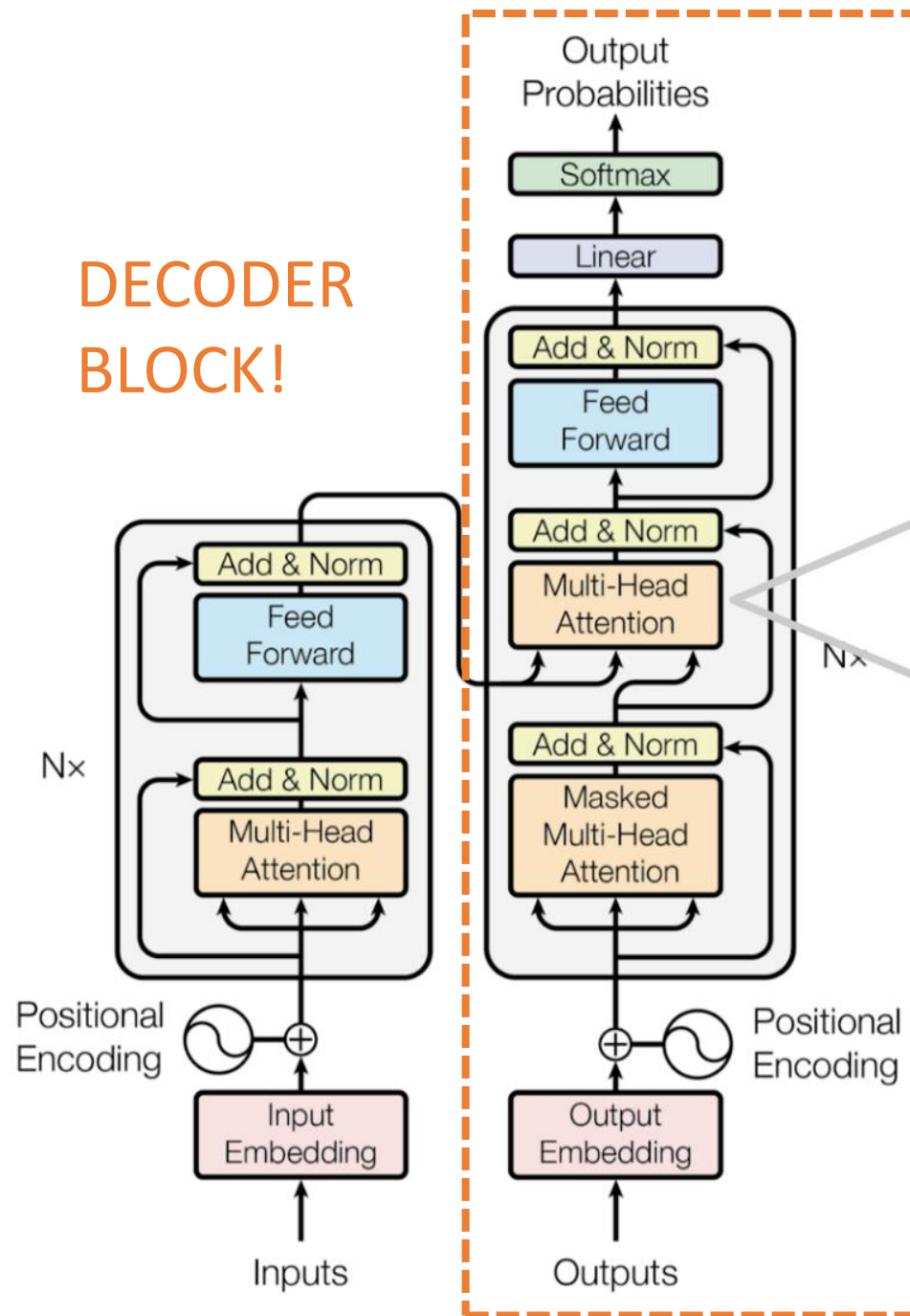
```

36 value_out = self.value(v)
37 key_dim = key.size(-1)
38 transposed_key_out = key.transpose(1,2)
39
40 # Calculate attention scores
41 scores = torch.bmm(query, transposed_key_out)/np.sqrt(key_dim)
42
43 # Apply masks if needed
44 if self.mask is not None:
45     mask = torch.triu(torch.ones(seq_len, seq_len), diagonal=1).to(device)
46     mask = mask.unsqueeze(0).repeat(q_shape, 1, 1)
47     scores = scores.masked_fill(mask == 0, float('-inf'))
48
49 # Final touches
50 weights = F.softmax(scores, dim = -1)
51 out = torch.bmm(weights, value_out)
52 return out

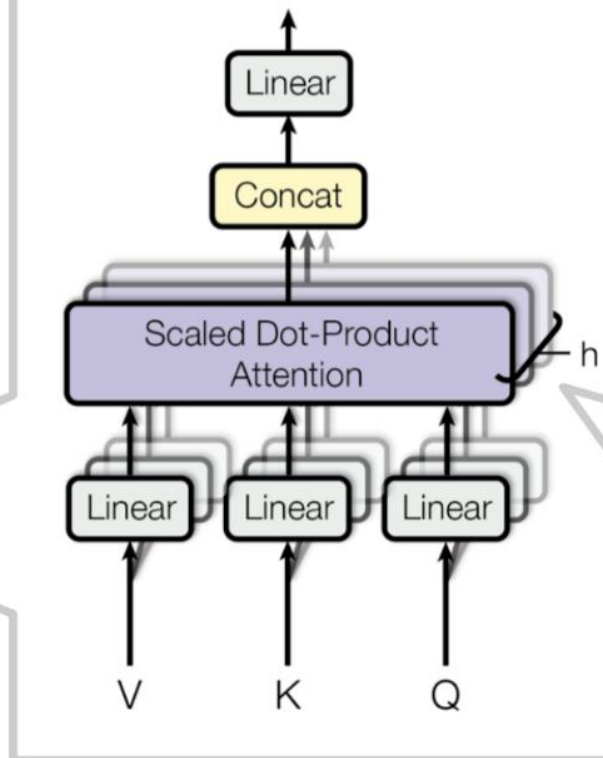
```



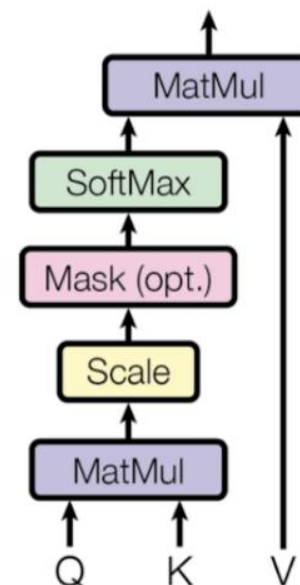
DECODER BLOCK!



Multi-head attention



Scaled dot-product attention

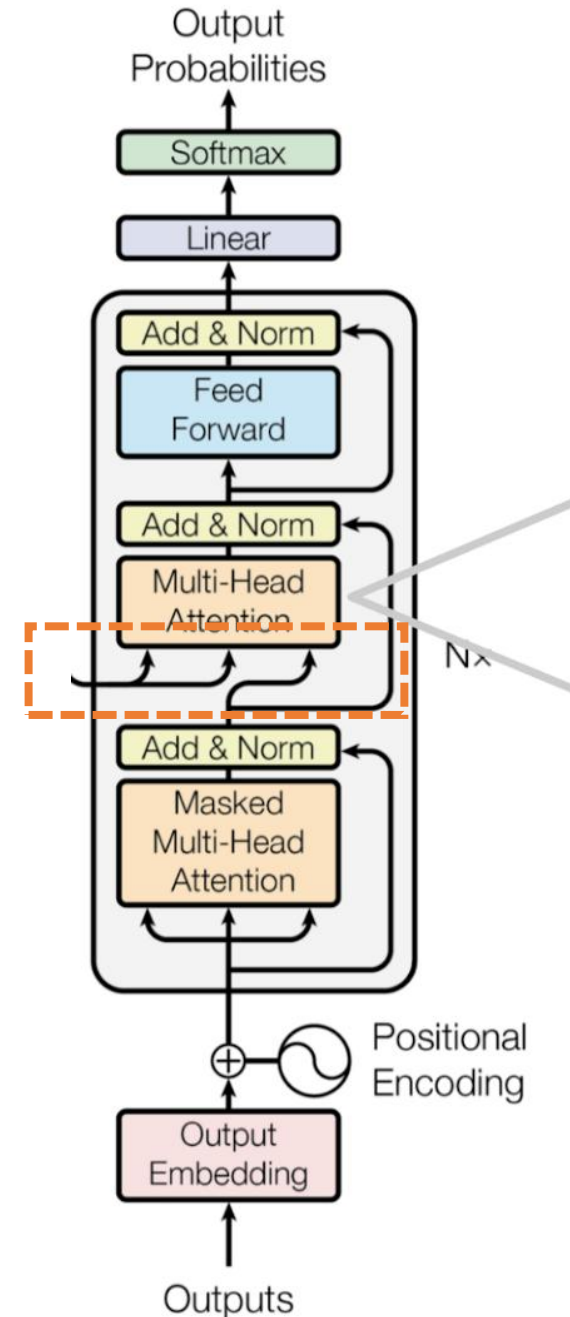


The attention layer

Definition (the multi-head self-attention layer a.k.a. the **attention layer):**

- The second attention block is special as it uses V and K vectors from the encoder part and vectors Q from the decoder part.

$$A = \text{softmax} \left(\frac{QK^T}{d} \right) V$$

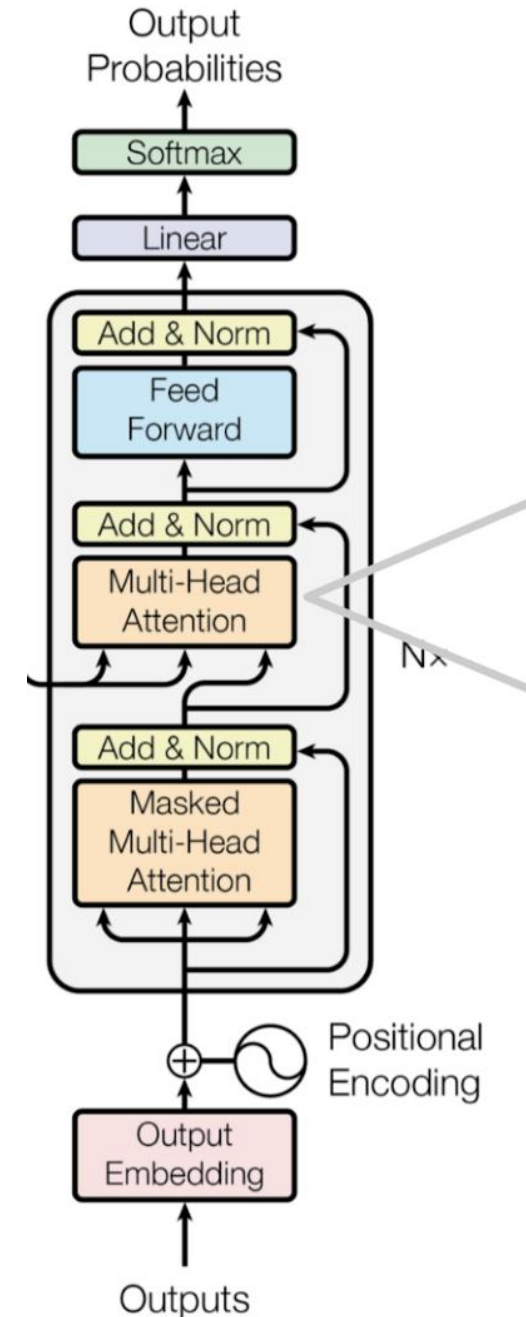


The attention layer

Definition (the multi-head self-attention layer a.k.a. the **attention layer):**

8. Repeat the attention block (multi-head attention + add/norm + feed forward) a few times.

Finish with a fully connected and softmax giving probabilities for the words in each position of the output sequence!



```
1 class DecoderLayer(nn.Module):
2     """
3     Description: DecoderLayer Class, describing the decoder layers to
4     be used in the Decoder part of the transformer.
5     Attributes list:
6     - d_model: An integer, defining the dimension of the encoder layers.
7     - num_heads: An integer, defining the number of attention heads
8     to be used in the decoder layer.
9     - d_inner: An integer, defining the number of neurons in the Feed Forward layers.
10    - dropout_rate: A float value between 0 and 1, corresponding to the Dropout rate
11    used in the Dropout layers of the Feed Forward layers.
12    """
```

```

14     def __init__(self, d_model, num_heads, d_inner = 2048, dropout_rate = 0.3):
15         """
16         Init Method, which adds up
17         - one MultiHeadAttentionLayer, with a triangular mask,
18         - one MultiHeadAttentionLayer, with no mask,
19         - one Feed Forward layer,
20         - and Normalization Layers after each one of them.
21         """
22         super().__init__()
23         # Masked First Attention Layer
24         self.masked_attention = MultiHeadAttentionLayer(d_model, \
25                                                         num_heads, \
26                                                         dropout_rate, \
27                                                         mask = True)
28         # Subsequent Attention Layer
29         self.subsequent_attention = MultiHeadAttentionLayer(d_model, \
30                                                             num_heads, \
31                                                             dropout_rate)
32         # Feed Forward Network Layer
33         self.ffn = nn.Sequential(nn.Linear(d_model, d_inner),
34                                   nn.ReLU(inplace = True),
35                                   nn.Dropout(dropout_rate),
36                                   nn.Linear(d_inner, d_model),
37                                   nn.Dropout(dropout_rate),)
38         # Normalization Layers
39         # (Roughly identical to BatchNorm)
40         self.masked_attention_normalization = nn.LayerNorm(d_model)
41         self.subsequent_attention_normalization = nn.LayerNorm(d_model)
42         self.ffn_normalization = nn.LayerNorm(d_model)

```

```
44 def forward(self, y, x):
45     """
46     Forward pass for the decoder layer.
47     """
48     # Masked First Attention Layer
49     out = y
50     out = out + self.masked_attention(q = out, k = out, v = out)
51     out = self.masked_attention_normalization(out)
52     # Subsequent Attention Layer
53     out = out + self.subsequent_attention(q = out, k = x, v = x)
54     out = self.subsequent_attention_normalization(out)
55     # Feed Forward Network Layer
56     out = out + self.ffn(out)
57     out = self.ffn_normalization(out)
58     return out
```

```

1  class Decoder(nn.Module):
2      """
3      Description: Encoder Class, describing the Encoding part of the transformer.
4      Attributes list:
5      - d_model: An integer, defining the dimension of the decoder layers.
6      - num_heads: An integer, defining the number of attention heads
7      to be used in both the decoder layers.
8      - num_decoders: An integer, defining the number of decoder layers
9      to be used in the transformer.
10     """
11
12     def __init__(self, d_model, num_heads, num_decoders):
13         """
14         Init Method, which adds up num_decoders DecoderLayers in a row.
15         """
16         super().__init__()
17         layers_list = [DecoderLayer(d_model, num_heads) for i in range(num_decoders)]
18         self.dec_layers = nn.ModuleList(layers_list, )
19
20     def forward(self, y, enc):
21         """
22         Forward pass for the Decoder, simply repeating the DeccoderLayers in a row.
23         """
24         out = y
25         for layers in self.dec_layers:
26             out = layer(out, enc)
27         return out

```

The attention layer

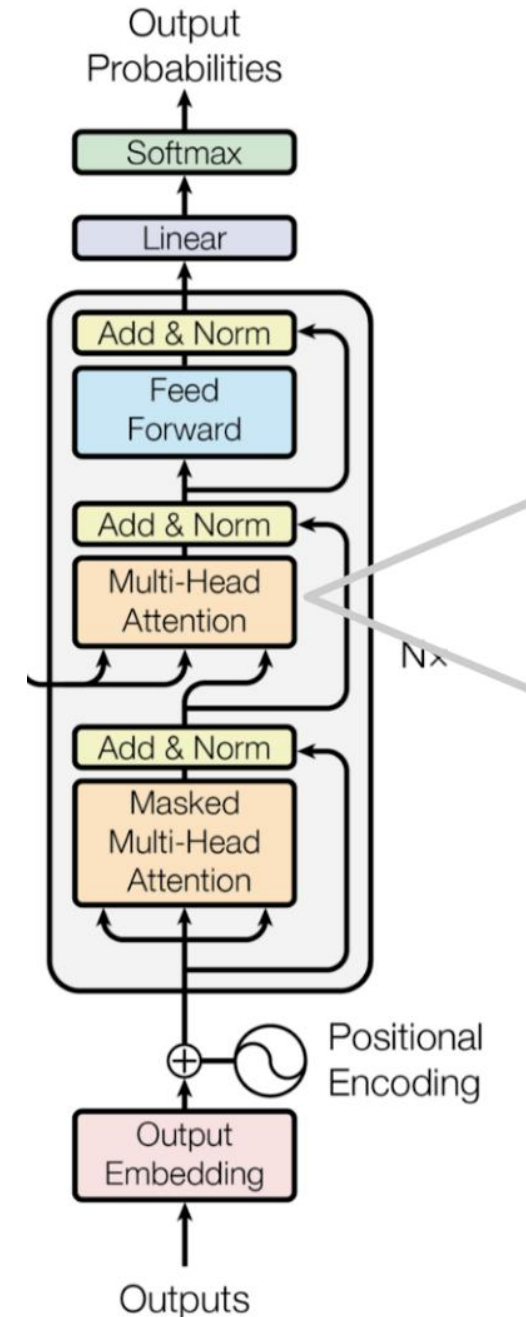
Definition (the multi-head self-attention layer a.k.a. the **attention layer):**

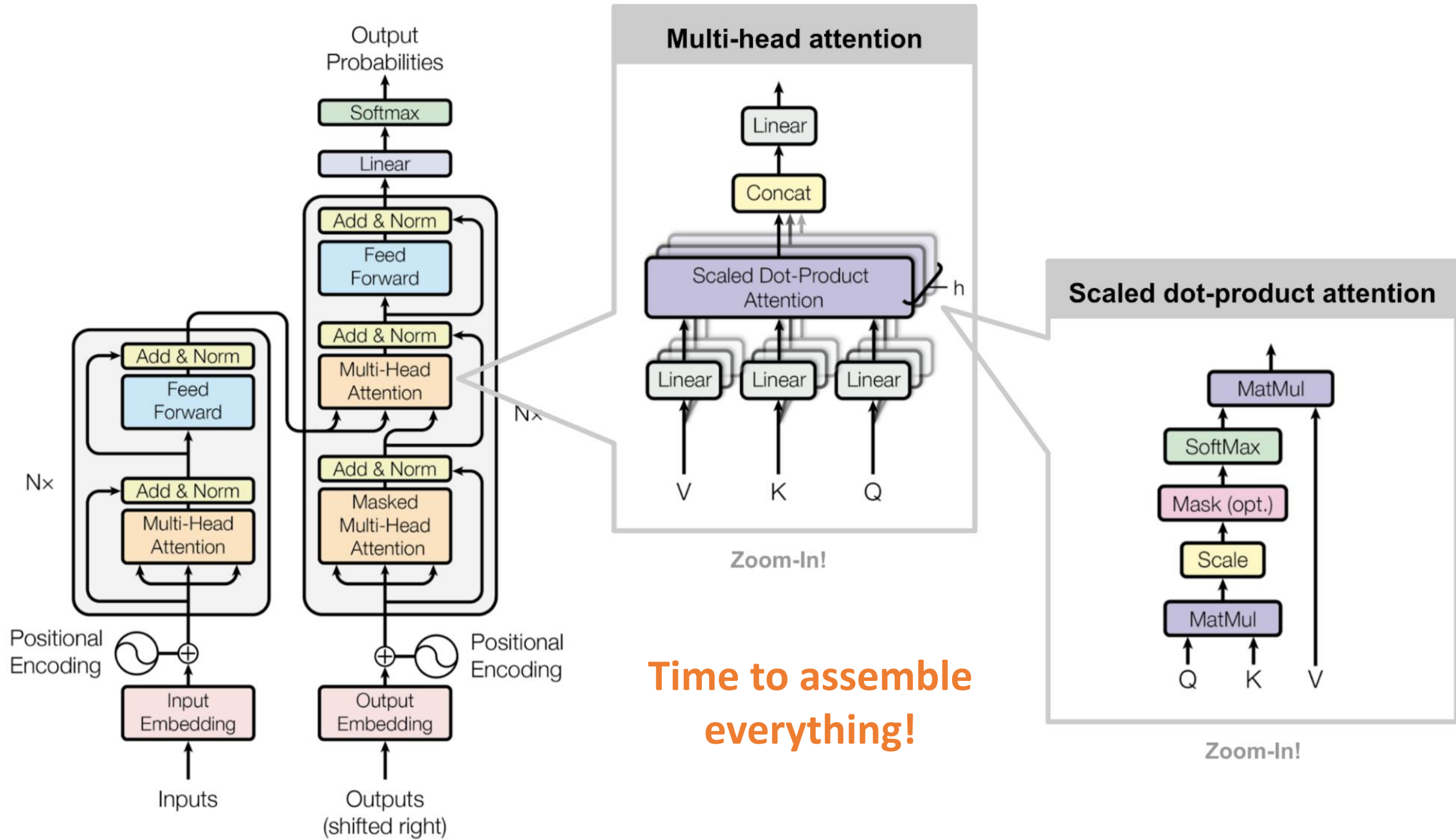
8. Train it with a dataset consisting of several pairs (x, y) of sentences.

For instance, chatbot with x being question prompts and y being expected answers.

Or translation with x and y being sentences in two different languages.

Etc.






```

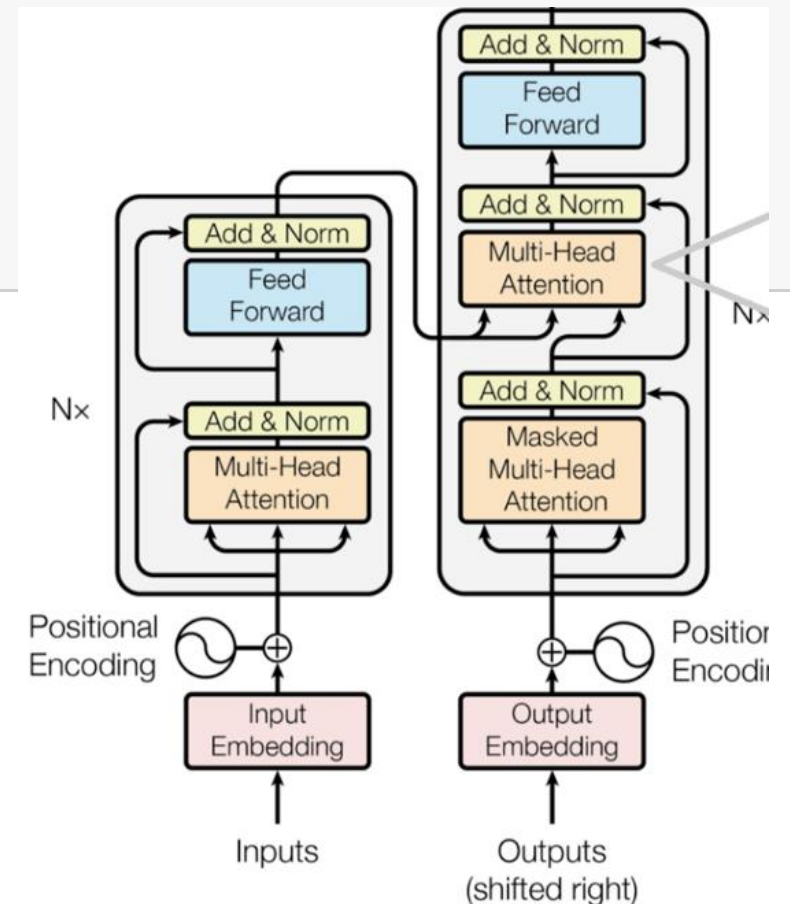
1  class Transformer(nn.Module):
2      """
3      Description: Full Transformer Class, combining the Encoder and Decoder parts.
4      Attributes list:
5      - d_model: An integer, defining the dimension of the encoder
6      and decoder layers.
7      - num_heads: An integer, defining the number of attention heads
8      to be used in both the encoder and decoder layers.
9      - num_encoders: An integer, defining the number of encoder layers
10     to be used in the transformer.
11     - num_decoders: An integer, defining the number of decoder layers
12     to be used in the transformer.
13     """
14
15     def __init__(self, d_model = 512, num_heads = 8, num_encoders = 6, num_decoders = 6):
16         """
17         Init Method, establishing the Decoder and Encoder parts as attributes of the transfo
18         Later on, we will extract the encoder part for our Embeddings.
19         """
20         super().__init__()
21         self.encoder = Encoder(d_model, num_heads, num_encoders)
22         self.decoder = Decoder(d_model, num_heads, num_decoders)
23

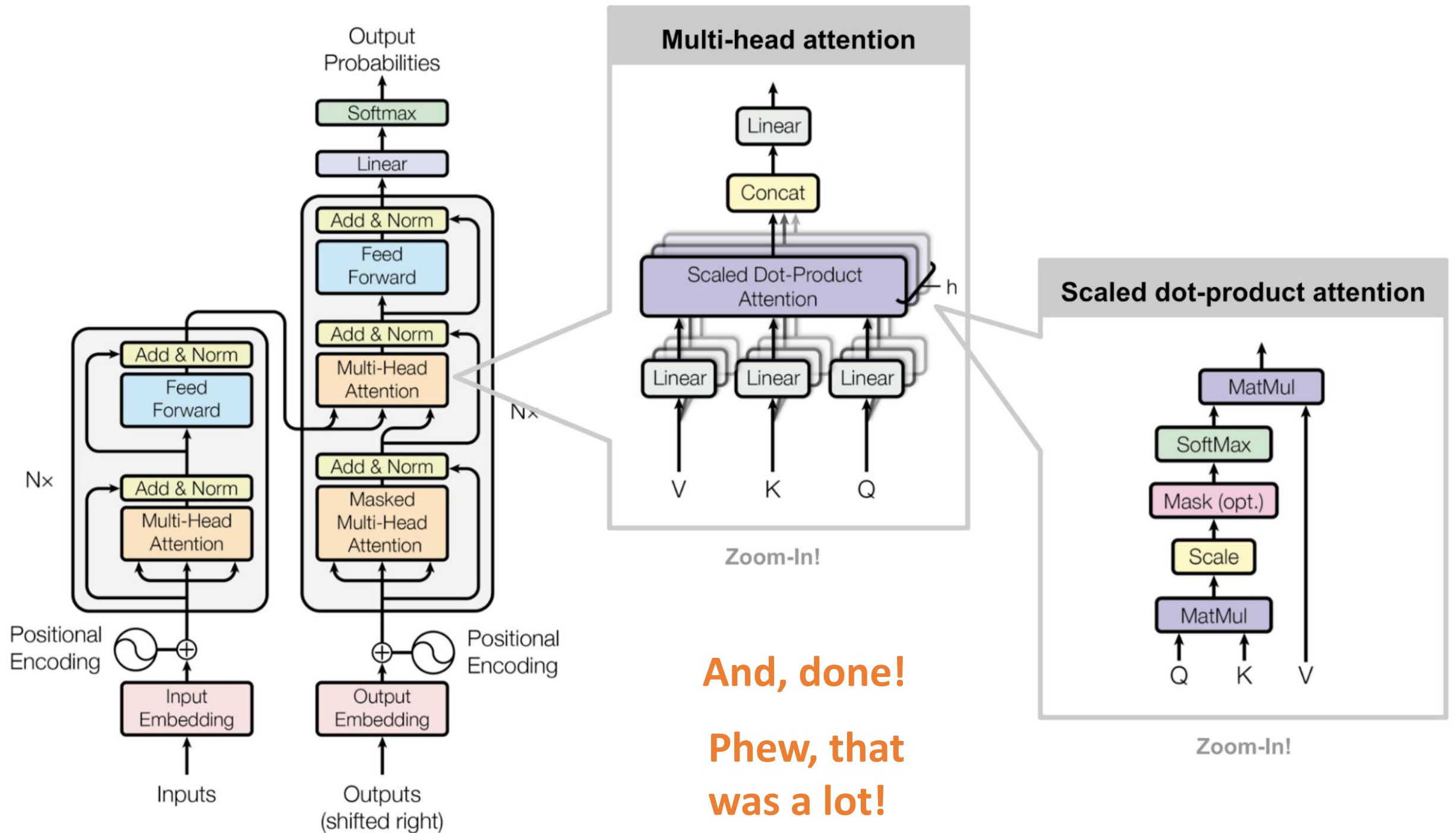
```

```

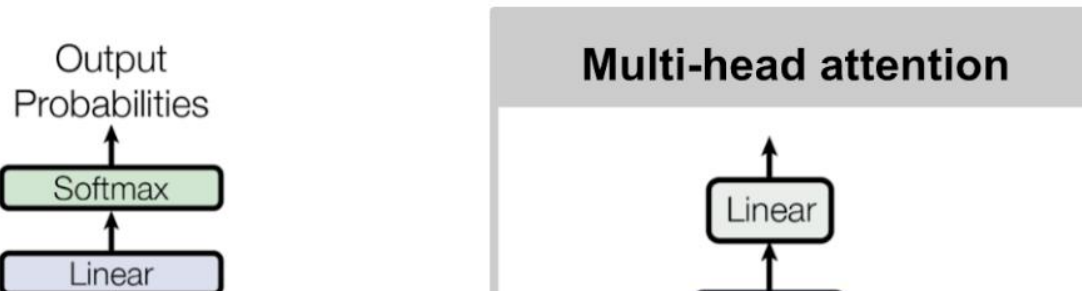
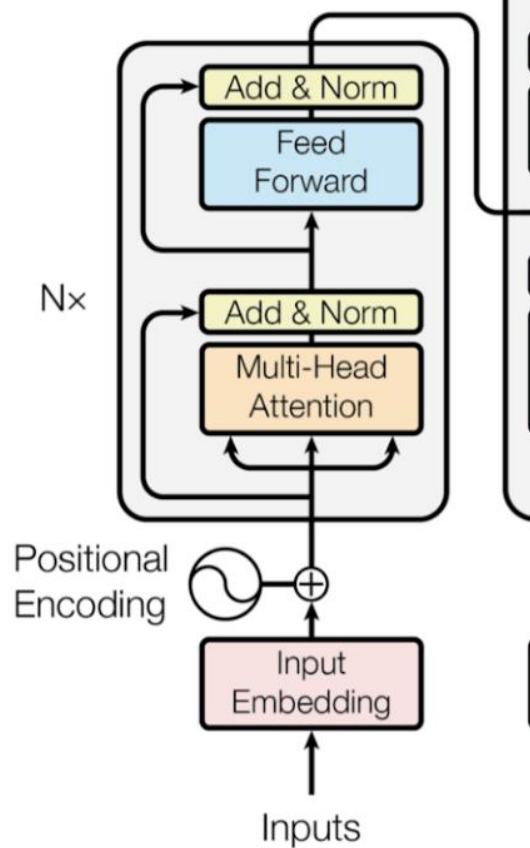
24
25 def forward(self, x, y):
26     """
27     Forward pass for transformer.
28     - Will encode the input with the Encoder part first.
29     - Then use the Decoder, combining the encoded input and its target,
30     along with masks.
31     """
32     enc_out = self.encoder(x)
33     dec_out = self.decoder(y, enc_out)
34     return dec_out

```





Well, at the end of the day, it is “just” matrix multiplications...



Credit fast.ai

```
class LSTMCell(Module):
    def __init__(self, ni, nh):
        self.forget_gate = nn.Linear(ni + nh, nh)
        self.input_gate = nn.Linear(ni + nh, nh)
        self.cell_gate = nn.Linear(ni + nh, nh)
        self.output_gate = nn.Linear(ni + nh, nh)

    def forward(self, input, state):
        h, c = state
        h = torch.stack([h, input], dim=1)
        forget = torch.sigmoid(self.forget_gate(h))
        c = c * forget
        inp = torch.sigmoid(self.input_gate(h))
        cell = torch.tanh(self.cell_gate(h))
        c = c + inp * cell
        out = torch.sigmoid(self.output_gate(h))
        h = outgate * torch.tanh(c)
        return h, (h, c)
```

WAIT SO YOU'RE TELLING ME
IT'S ALL MATRIX MULTIPLICATIONS?

ALWAYS HAS BEEN

Great news!

The 2022 release of PyTorch **COMES WITH TRANSFORMERS CLASSES!**

<https://pytorch.org/docs/stable/generated/torch.nn.Transformer.html>

TRANSFORMER

```
CLASS torch.nn.Transformer(d_model=512, nhead=8, num_encoder_layers=6,  
num_decoder_layers=6, dim_feedforward=2048, dropout=0.1, activation=<function relu>,  
custom_encoder=None, custom_decoder=None, layer_norm_eps=1e-05, batch_first=False,  
norm_first=False, device=None, dtype=None) [SOURCE]
```

A transformer model. User is able to modify the attributes as needed. The architecture is based on the paper “Attention Is All You Need”. Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In Advances in Neural Information Processing Systems, pages 6000-6010. Users can build the BERT(<https://arxiv.org/abs/1810.04805>) model with corresponding parameters.

Applications for attention/transformers

In NLP: Sentences are typical examples of sequences where computing attention coefficients make sense as a way to help with contextualization.

- Other applications for attention exist, typically with images, where attention coefficients can help understand how pixels interact with each other! Technically attention layers could “replace” CNNs!
- Or how pixels could affect a caption decision (image and video captioning, handwritten recognition, etc.)
- Will also be used in graphs next week!

The BERT word embedding

- Why discuss transformers during this (long) lecture on embeddings?
- Well, let us assume we train a transformer for a meaningless linguistics task...
- Can we extract the encoder part, as we did with all the unsupervised approaches to word embedding (CBoW, SkipGram, FastText, ELMo)?
- **Yes! And you obtain one of the current best word embeddings, BERT and GPT! [Devlin2019]!**
- Python package:
<https://pypi.org/project/bert-embedding/>



Also, GPT-n embeddings

- On another note, OpenAI has also produced an important embedding, also relying on transformers, called **GPT-3** [Brown2020][GPT-3].
- **GPT**, being the acronym for **Generative Pretrained Transformer**.
- Similar to BERT, it is also considered one of the current state-of-the-art embeddings.



GPT3 and ChatGPT

- In fact, GPT3 (and even 4 these days!) is the **encoder model** that is used in ChatGPT to process the text prompt you have entered!
- This **encoding vector** is later passed to a **decoder**, whose objective is to produce an answer to your text prompt, in the form of a text sequence!
- Moving closer and closer to understanding ChatGPT fully, but **more concepts are needed for the decoder part!**
- (Something for later!)

A pointer on GPT-5?

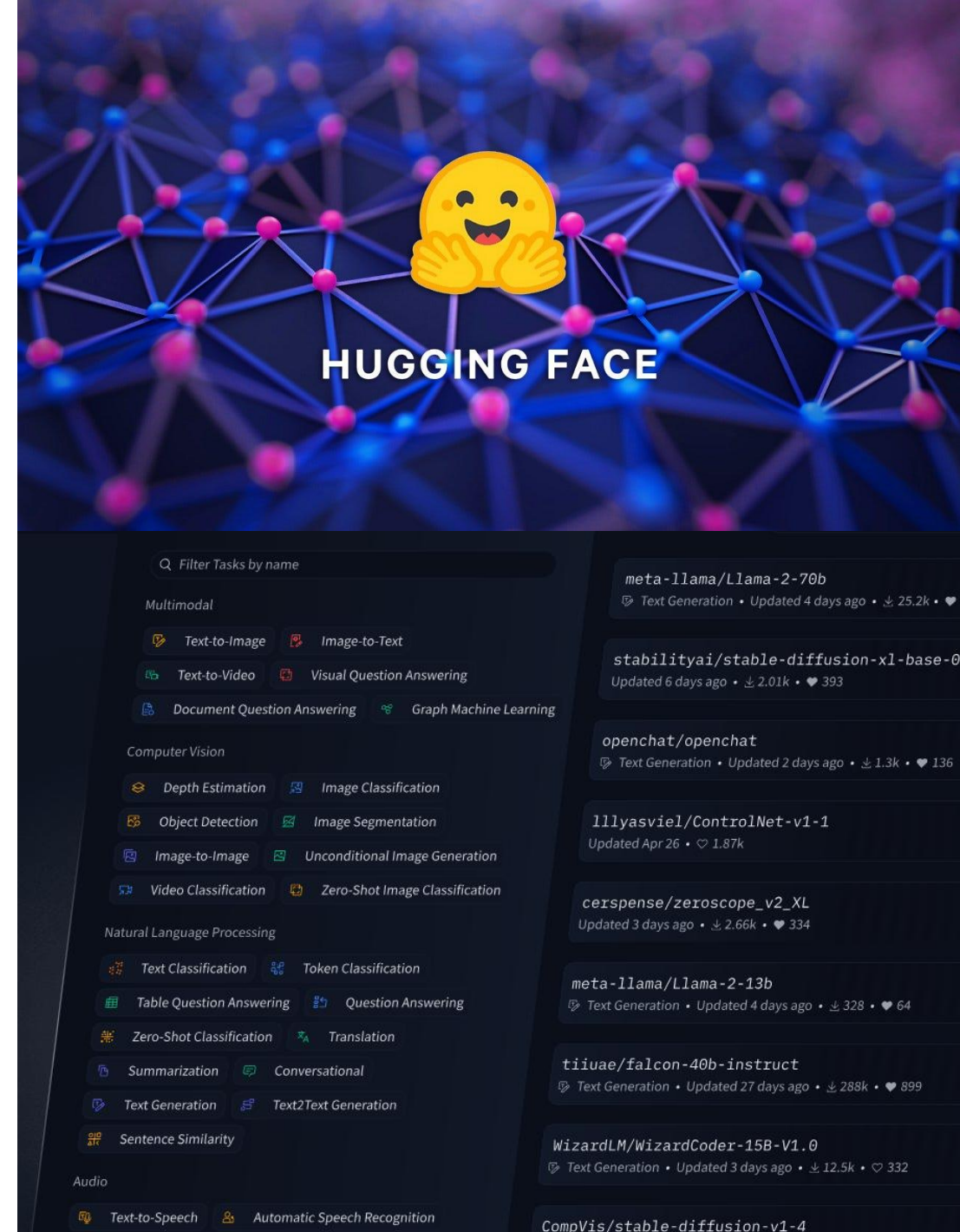


Transformers on HuggingFace

Definition (**Hugging Face**):

Hugging Face is a machine learning company that provides tools, models, and libraries designed to make natural language processing (NLP) and other machine learning tasks more accessible.

Widely used in both academia and industry to simplify the process of working with models, especially those based on transformers.



Implementing transformers on HuggingFace

In Notebook 5, we show how to implement a BERT word embedding using HuggingFace.

It consists of downloading and using a pre-trained model.

(Often advised if you need a word embedding, training one is tough)

It includes a **tokenizer** and a word embedding model.

```
# Create and load pre-trained BERT model and tokenizer  
# (Ignore the warning, if any.)  
# (This might take a while if you run it for the first time,  
# as we need to download the model from huggingface and it is 400MB!)  
model_name = 'bert-base-uncased'  
tokenizer = BertTokenizer.from_pretrained(model_name)  
model = BertModel.from_pretrained(model_name)
```

```
loading file vocab.txt from cache at C:\Users\matth\.cache\huggingface  
ab.txt  
loading file added_tokens.json from cache at None  
loading file special_tokens_map.json from cache at None  
loading file tokenizer_config.json from cache at C:\Users\matth\.cache  
7f723594\tokenizer_config.json  
loading file tokenizer.json from cache at C:\Users\matth\.cache\huggin  
4\tokenizer.json  
loading file chat_template.jinja from cache at None
```

About tokenizers and models

Definition (**Tokenizer**):

IN NLP, a **tokenizer** is responsible for preprocessing text input so it can be understood by the model. The tokenizer will typically:

- Split the input text into smaller units called "tokens." This breaks down words into tokens or word pieces (e.g., "unhappiness" will decompose into the tokens ["un", "happy", "ness"]).
- Add special tokens such as separators (to separate sentences, convert special symbols or mark the end of a sequence).
- Convert tokens into numerical IDs based on the model's vocabulary.

Tokenizers will be explored more in detail in the Term 7 NLP course.

Implementing transformers on HuggingFace

Later on, we can use our pre-trained BERT model to produce word embeddings.

We can also confirm that the BERT word embeddings preserve semantic similarity using the cosine similarity function!

(Feel free to use it for projects!)

```
# Define the words to get embeddings for
words = ["cat", "kitten", "university"]

# Process each word individually
for word in words:
    # Tokenize first
    inputs = tokenizer(word, return_tensors = "pt", add_special_tokens = False)
    # Compute embedding second
    with torch.no_grad():
        outputs = model(**inputs)
    # Extract the embedding for the word (BERT's output includes embeddings for all tokens)
    embedding = outputs.last_hidden_state.mean(dim = 1)
    embeddings.append(embedding.squeeze().numpy())
```

```
# Helper function to check similarity between the embeddings
```

```
def cosine_similarity(vec1, vec2):
    return np.dot(vec1, vec2)/(norm(vec1)*norm(vec2))
```

```
# Compute the cosine similarities between some pairs of words to check embedding similarities
similarity_cat_kitten = cosine_similarity(embeddings[0], embeddings[1])
similarity_cat_university = cosine_similarity(embeddings[0], embeddings[2])
similarity_kitten_university = cosine_similarity(embeddings[1], embeddings[2])
print(f"Cosine similarity between 'cat' and 'kitten': {similarity_cat_kitten:.4f}")
print(f"Cosine similarity between 'cat' and 'university': {similarity_cat_university:.4f}")
print(f"Cosine similarity between 'kitten' and 'university': {similarity_kitten_university:.4f}")
```

```
Cosine similarity between 'cat' and 'kitten': 0.8756
Cosine similarity between 'cat' and 'university': 0.4423
Cosine similarity between 'kitten' and 'university': 0.4332
```

Conclusion (W9S3)

We have seen a few approaches to embeddings.

- Train an AI to figure out embeddings?
 - Basic approaches.
- Out of dictionary entries?
 - Use ELMo or FastText!

(Alternative supervised methods?)

- *(Supervised: InferSent)*
- *(Multi-task: Google's Universal Embedding.)*

A few more problems are still open at the moment for these word embeddings.

- Can these embedding be biased? Yes, unfortunately, active research on this.
- Can we help the network in computing context? Yes, with attention and transformers!
- Best word embedding at the moment?
Probably GPT4!

Learn more about these topics

Out of class, for those of you who are curious

- [Schnabel2015] **Schnabel** et al., “Evaluation methods for unsupervised word embeddings”, 2015.
<https://www.aclweb.org/anthology/D15-1036/>
- [Bolukbasi2016] **Bolukbasi** et al., “Man is to Computer Programmer as Woman is to Homemaker? Debiasing Word Embeddings”, 2016.
<https://arxiv.org/abs/1607.06520>
- [Vaswani2017] **Vaswani** et al., “Attention Is All You Need”, 2017.
<https://arxiv.org/abs/1706.03762>

Learn more about these topics

Out of class, for those of you who are curious

- [Conneau2018] **Conneau** et al., “Supervised Learning of Universal Sentence Representations from Natural Language Inference Data”, 2018.
<https://arxiv.org/abs/1705.02364>
- [Cer2019] **Cer** et al., “Universal Sentence Encoder”, 2019.
<https://arxiv.org/abs/1803.11175>
- [Wang2018] Wang et al., “GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding”
<https://arxiv.org/abs/1804.07461>

Learn more about these topics

Out of class, for those of you who are curious

- [Devlin2019] **Devlin** et al., “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”, 2019.
<https://arxiv.org/abs/1810.04805>
- [Brown2020] Brown et al. (20 of them or so), “Language Models are Few-Shot Learners”, 2020.
<https://arxiv.org/abs/2005.14165>

Learn more about these topics

Out of class, for those of you who are curious

- [Dong2021] Dong et al., “Attention is Not All You Need: Pure Attention Loses Rank Doubly Exponentially with Depth”, 2021.
<https://arxiv.org/abs/2103.03404>
- [Bello2021] Bello et al., “LambdaNetworks: Modeling Long-Range Interactions Without Attention”, 2021.
<https://arxiv.org/abs/2102.08602>

Learn more about these topics

Out of class, for those of you who are curious

- [GPT-3] Read more about GPT-3 and OpenAI, here:
<https://www.vox.com/future-perfect/21355768/gpt-3-ai-openai-turing-test-language>
<https://www.technologyreview.com/2020/07/20/1005454/openai-machine-learning-language-generator-gpt-3-nlp/>
- [Forbes-GPT4] Read more about the GPT-4 model, next big update and future candidate for the state-of-the-art of language models, in this article, here (but more importantly, track it and do not miss it):
<https://www.forbes.com/sites/bernardmarr/2023/02/24/gpt-4-is-coming--what-we-know-so-far/>

Learn more about these topics

Out of class, for those of you who are curious

- [GPT4OAI] The GPT-4 announcement from Open AI (made today, as of March 15th 2023!)

<https://openai.com/research/gpt-4>

- [Cohere] A great article about word and sentence embeddings with cool visualization.

<https://txt.cohere.ai/sentence-word-embeddings/>

Learn more about these topics

Tracking important names (Track their works and follow them on Scholar, Twitter, or whatever works for you!)

- **Alexis Conneau: Senior Researcher at Google.**
<https://scholar.google.fr/citations?user=45KfCpgAAAAJ&hl=fr>
- **Daniel Cer: Researcher at Google and Lecturer at UC Berkeley.**
He has a really good course about NLP and Embeddings.
<https://scholar.google.com/citations?user=BrT1NW8AAAAJ&hl=en>
<https://www.ischool.berkeley.edu/user/4406>
<https://www.ischool.berkeley.edu/courses/datasci/266>

Learn more about these topics

Tracking important names (Track their works and follow them on Scholar, Twitter, or whatever works for you!)

- **Tobias Schnabel**: Senior researcher at Microsoft.
https://scholar.google.com/citations?user=o_i419UAAAAJ&hl=en
<https://www.microsoft.com/en-us/research/people/toschnab>
- **Tolga Bolukbasi**: Senior researcher at Google Brain.
<https://scholar.google.com/citations?user=3rF9gtAAAAAJ&hl=en>
- **Anish Vaswani**: Researcher at Google Brain. Highest cited paper in DL.
<https://scholar.google.com/citations?user=oR9sCGYAAAAJ&hl=en>
- **Jacob Devlin**: Researcher at Google.
<https://research.google/people/106320/>

Learn more about these topics

Tracking important names (Track their works and follow them on Scholar, Twitter, or whatever works for you!)

- **Greg Brockman: CEO of OpenAI.**
<https://gregbrockman.com/>
<https://dblp.org/pid/74/10715.html>
- **Sam Altman: Former CEO of OpenAI**, still involved in research. Very active on Twitter and technically, the social media face of OpenAI?
<https://blog.samaltman.com/>

Learn more about these topics

Tracking important names (Track their works and follow them on Scholar, Twitter, or whatever works for you!)

- **Alec Radford**: Former research scientist at **OpenAI** and was one of the primary authors of the GPT. Also worked on Dall-E.
<https://scholar.google.com/citations?user=dOad5HoAAAAJ&hl=en>
<https://openai.com/blog/authors/alec/>
- **OpenAI**: The company behind ChatGPT. Already mentioned it on W4S2.
<https://openai.com/>

Supervised and combined learning?

- Word2Vec, GloVe, FastText and ELMo approaches are commonly referred to as **unsupervised (or semi-supervised) approaches to embedding**, both relying on the **distributional hypothesis** we mentioned on the previous lecture.
- Unsupervised representation learning of sentences had been the norm for quite some time.
- **More advanced versions of unsupervised approaches exist!**
- However, the last few years have seen a shift toward **supervised** and **multi-task learning schemes** with a number of notably interesting proposals in late 2017/early 2018.

From unsupervised to combined learning

For a long time, there was a general consensus (see [Wang2018] for details) in the field that

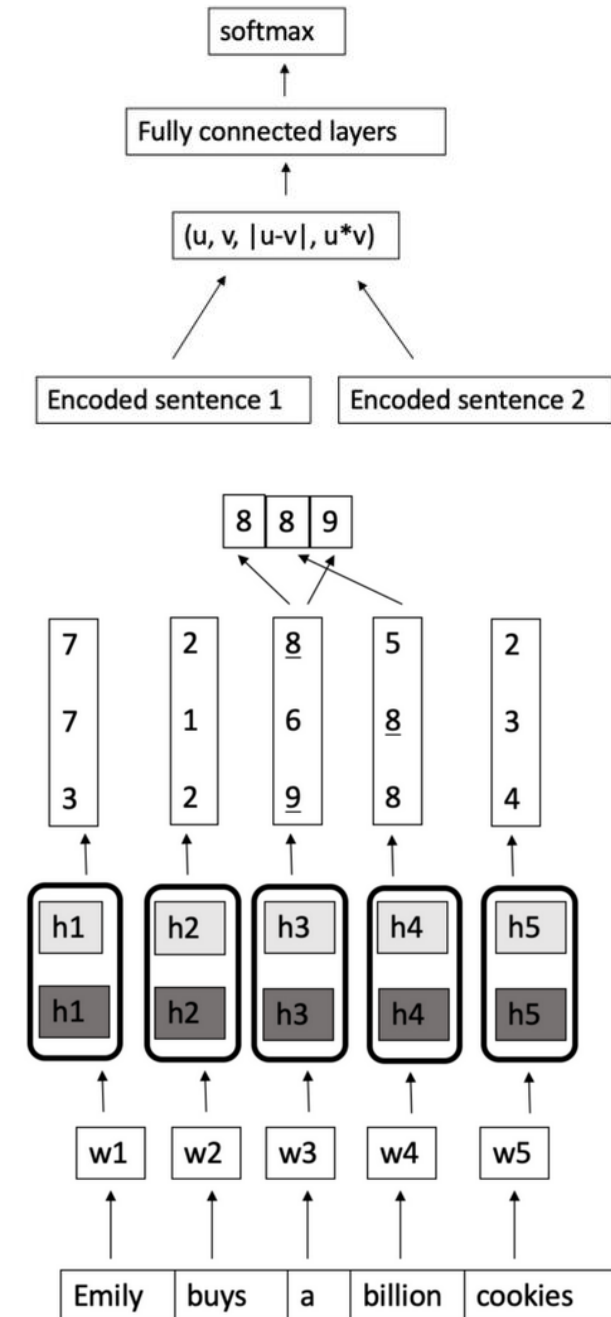
- the simple approach of **simply averaging a sentence's word vectors** (as in CBoW approach) gives a strong baseline for context, which is good enough for many downstream tasks.
- **supervised** learning of sentence/word embeddings was costly and gave lower-quality embeddings than unsupervised/semi-supervised approaches.

But these assumptions have recently been overturned, in part following the publication of the InferSent ([Conneau2018]) results.

Counter-example: InferSent (out-of-class)

Simple key takeaways

- The InferSent model (in [Conneau2018]) is an example of a supervised classification model, with three classes: Entailment, Neutral, Contradiction.
- It uses the same forward-backward LSTM architecture as ELMo, and attempts to compare two sentences and classify the pair.
- Feature representation extracted from model showed good performance.



From unsupervised to combined learning

- Following these recent results, showing that supervised approaches are not necessarily bad...
- The current direction in research for word embeddings suggests to combine both supervised and unsupervised learning techniques. This is commonly referred to as a **multi-task learning based embedding**.
- The most notable work in this direction would be **Google's Universal Sentence and Word Encoder** (in [Cer2019]). Kept rather hidden, have only seen a pre-release on Tensorflow:
https://www.tensorflow.org/hub/tutorials/semantic_similarity_with_tf_hub_universal_encoder
- (But nothing for PyTorch yet?)

On the limits of attention?

- Active research following the BERT and GPT embeddings, wondering what might be the next big revolution? ([Devlin2019])
- In the paper [Dong2021], a research team from Google and EPFL proposed a novel approach that sheds light on the operation and inductive biases of self-attention/transformer networks and finds that pure attention performance decays in rank doubly exponentially with respect to depth.
- Recent works suggest that a new type of transformer layer, called **Lambda layers** could become the next hype for the future of NLP embeddings [Bello2021]?
- Or simply GPT-4? ([Forbes-GPT4]).