Language Technology

Chapter 15: Self-Attention and Transformers

Pierre Nugues

Pierre.Nugues@cs.lth.se

September 26 and October 3rd, 2024

Transformers

Transformers:

- An architecture proposed in 2017 based on the concept of **attention**
- Consists of a smart pipeline of matrices
- They can learn complex lexical relations

Using Transformers

Goals of transformers:

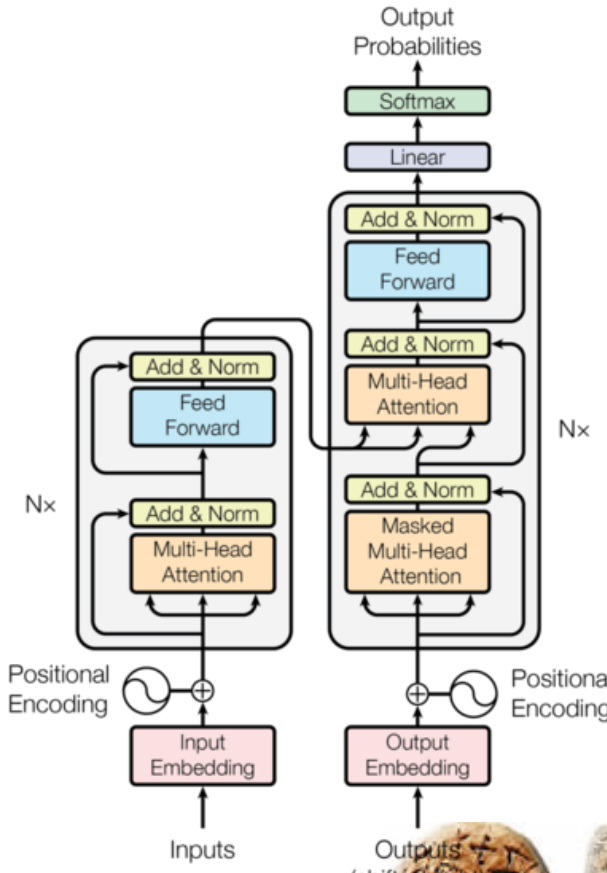
- Encapsulate a massive amount of knowledge.
- In consequence trained on very large corpora
- Sometimes marketed as the ImageNet moment (See `https://ruder.io/nlp-imagenet/`)

Transformers in practice:

- Large companies train transformers on colossal corpora, the pretrained models, requiring huge computing resources (https://arxiv.org/pdf/1906.02243.pdf)
- Mere users:
  - Reuse the models in applications
  - Fine-tune some parameters in the downstream layers

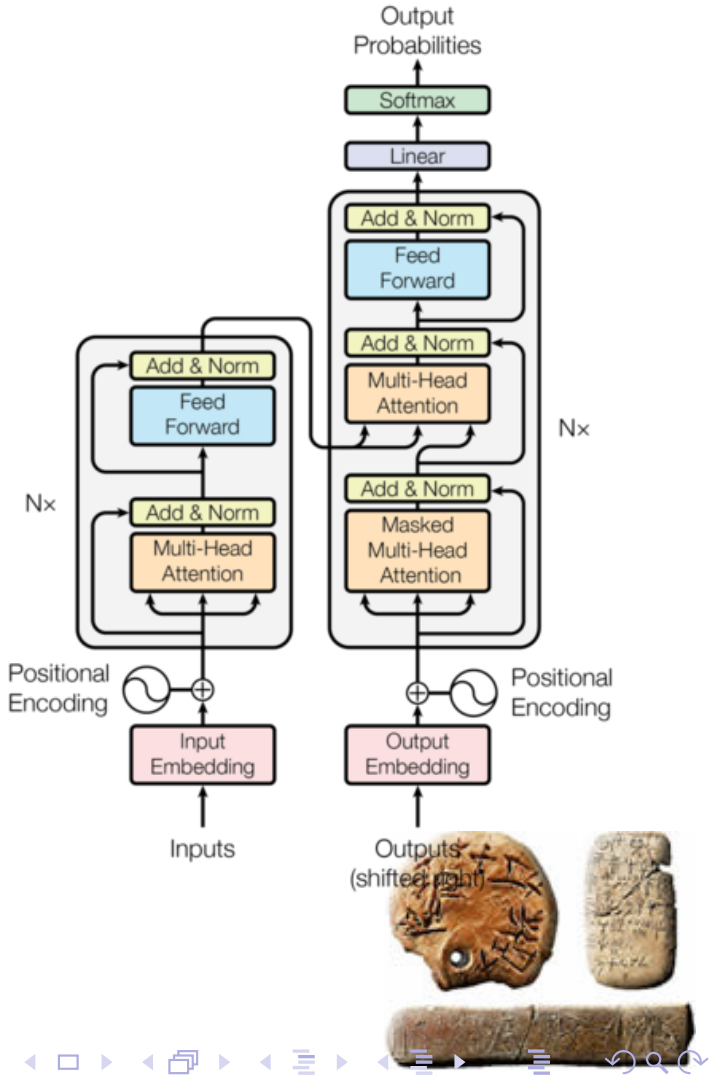
The Original Architecture

- Reference paper: *Attention Is All You Need* by Vaswani et al (2017)
  Link: `https://arxiv.org/pdf/1706.03762.pdf`
- Architecture consisting of two parts: encoder (left) and decoder (right)
- The encoder and decoder are essentially sequences of matrix multiplications with a few other functions
- Implementation in PyTorch: `https://nlp.seas.harvard.edu/2018/04/03/attention.html`



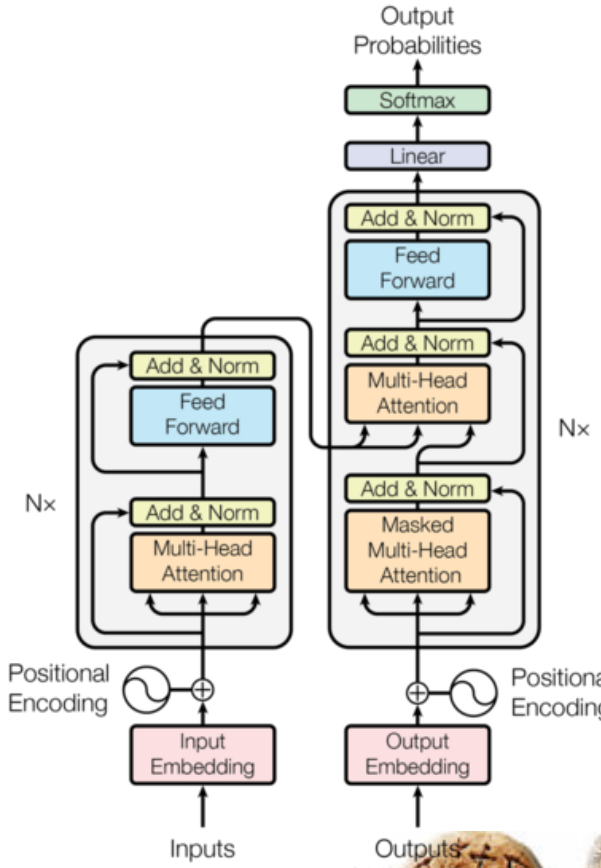
Components

1. Attention: The core instrument to learn complex lexical relations
2. The encoder-decoder. The first application was for translation.
3. The encoder produces a representation of the input sentence.
4. The decoder uses this representation to generate the target sentence with an autoregressive procedure.



Encoder-Decoder

- The components:
  1. Encoders, the left part that can be used alone for classification tasks (BERT):
  2. Decoders, the right part that can be used to generate texts (GPT)
- We first consider attention and the encoding part



The Encoder

The encoder is a transducer: A sequence-to-sequence model with sequences of identical length.

A task exemplifying this in NLP is POS tagging.

With the phrase:

The first of their countrymen to visit Mexico...

The annotation using the UPOS tagset is:

y	Det	Adj	Adp	Pron	Noun	Part	Verb	Propn
x	The	first	of	their	countrymen	to	visit	Mexico

The **x** and **y** vectors must have the same length.

Features with a Feed-forward Network

Note the tags are from the Penn Treebank

ID	Feature vectors: X							PPOS: $\hat{y}$
	w_{i-2}	w_{i-1}	w_i	w_{i+1}	w_{i+2}	t_{i-2}	t_{i-1}	
1	BOS	BOS	**Battle**	-	tested	BOS	BOS	NN
2	BOS	Battle	**-**	tested	Japanese	BOS	NN	HYPH
3	Battle	-	**tested**	Japanese	industrial	NN	HYPH	JJ
...	...	...	...	...	...	...	...	...
19	the	first	**of**	their	countrymen	DT	JJ	IN
20	first	of	**their**	countrymen	to	JJ	IN	PRP$
21	of	their	**countrymen**	to	visit	IN	PRP$	NNS
22	their	countrymen	**to**	visit	Mexico	PRP$	NNS	TO
23	countrymen	to	**visit**	Mexico	,	NNS	TO	VB
24	to	visit	**Mexico**	,	a	TO	VB	NNP
25	visit	Mexico	**,**	a	boatload	VB	NNP	,
...	...	...	...	...	...	...	...	...
34	ashore	375	**years**	ago	.	RB	CD	NNS
35	375	years	**ago**	.	EOS	CD	NNS	RB
36	years	ago	**.**	EOS	EOS	NNS	RB	.

Two problems:

1. Embeddings have a unique association with a word (or subword)
2. Reusing the previous tags with a feed-forward architecture needs extra programming

Contextual Embeddings

Embeddings we have seen so far do not take the context into account
Attention is a way to make them aware of the context.
Consider the sentence:

> *I must go back to my ship and to my crew*
> *Odyssey, book I*

The word *ship* can be a verb or a noun with different meanings, but has only one GloVe embedding vector
Compare:

> *We process and ship your order in the most cost-efficient way possible*

from an Amazon commercial page
Self-attention will enable us to compute contextual word embeddings.

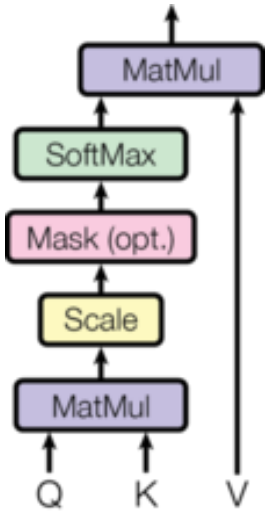
Self-Attention

In the paper *Attention is all you need*, Vaswani et al. (2017) use three kinds of vectors, queries, keys, and values. Here we will use one type corresponding to GloVe embeddings.

We compute the attention this way:

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

where d_k is the dimension of the input.



The softmax function is defined as:

$$\text{softmax}(x_1, x_2, ..., x_j, ..., x_n) = (\frac{e^{x_1}}{\sum_{i=1}^{n} e^{x_i}}, \frac{e^{x_2}}{\sum_{i=1}^{n} e^{x_i}}, ..., \frac{e^{x_j}}{\sum_{i=1}^{n} e^{x_i}}, ..., \frac{e^{x_n}}{\sum_{i=1}^{n} e^{x_i}})$$

The meaning of $QK^\intercal$

$QK^\intercal$ is the dot product of the GloVe vectors. It will tell us the similarity between the words

This is analogous to cosine similarity:

	i	must	go	back	to	my	**ship**	and	to	my	**crew**
i	1.00	0.75	0.86	0.76	0.73	0.90	0.35	0.65	0.73	0.90	0.42
must	0.75	1.00	0.85	0.68	0.87	0.69	0.42	0.69	0.87	0.69	0.45
go	0.86	0.85	1.00	0.84	0.84	0.81	0.41	0.68	0.84	0.81	0.49
back	0.76	0.68	0.84	1.00	0.83	0.76	0.49	0.77	0.83	0.76	0.51
to	0.73	0.87	0.84	0.83	1.00	0.68	0.54	0.86	1.00	0.68	0.51
my	0.90	0.69	0.81	0.76	0.68	1.00	0.38	0.63	0.68	1.00	0.44
ship	0.35	0.42	0.41	0.49	0.54	0.38	**1.00**	0.46	0.54	0.38	**0.78**
and	0.65	0.69	0.68	0.77	0.86	0.63	0.46	1.00	0.86	0.63	0.49
to	0.73	0.87	0.84	0.83	1.00	0.68	0.54	0.86	1.00	0.68	0.51
my	0.90	0.69	0.81	0.76	0.68	1.00	0.38	0.63	0.68	1.00	0.44
crew	0.42	0.45	0.49	0.51	0.51	0.44	**0.78**	0.49	0.51	0.44	**1.00**

Code Example

Jupyter Notebook:
https://github.com/pnugues/pnlp/tree/main/notebooks

Vaswani's attention score

The attention scores are scaled and normalized by the softmax function.

$$\text{softmax}(\frac{QK^\intercal}{\sqrt{d_k}}),$$

	i	must	go	back	to	my	**ship**	and	to	my	**crew**
i	0.36	0.05	0.07	0.05	0.04	0.19	0.01	0.02	0.04	0.19	0.01
must	0.14	0.20	0.10	0.06	0.11	0.10	0.03	0.05	0.11	0.10	0.02
go	0.18	0.09	0.14	0.09	0.08	0.13	0.02	0.04	0.08	0.13	0.02
back	0.14	0.05	0.09	0.19	0.08	0.12	0.03	0.06	0.08	0.12	0.03
to	0.11	0.11	0.09	0.09	0.15	0.08	0.04	0.07	0.15	0.08	0.03
my	0.19	0.03	0.05	0.04	0.03	0.29	0.01	0.02	0.03	0.29	0.01
ship	0.03	0.03	0.03	0.04	0.05	0.03	**0.55**	0.03	0.05	0.03	**0.13**
and	0.10	0.08	0.07	0.10	0.12	0.09	0.04	0.15	0.12	0.09	0.04
to	0.11	0.11	0.09	0.09	0.15	0.08	0.04	0.07	0.15	0.08	0.03
my	0.19	0.03	0.05	0.04	0.03	0.29	0.01	0.02	0.03	0.29	0.01
crew	0.06	0.05	0.05	0.06	0.05	0.06	**0.21**	0.04	0.05	0.06	**0.31**

Attention

We use these scores to compute the attention.

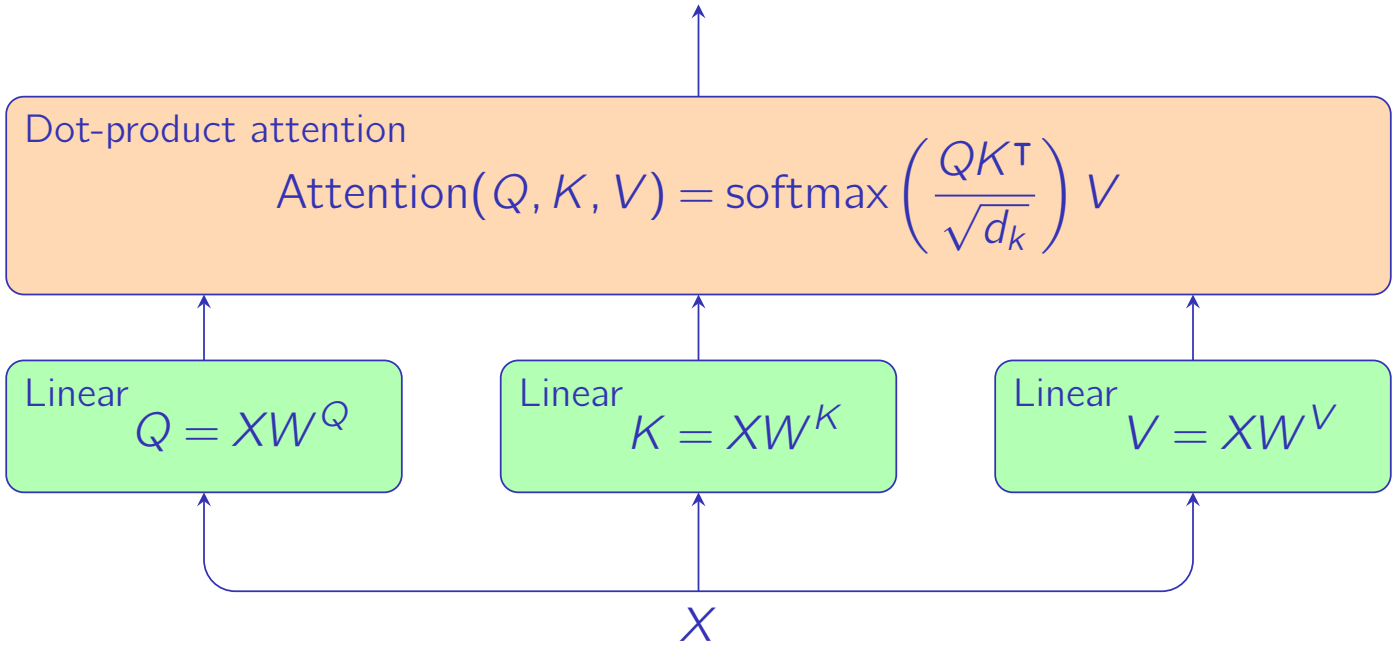
$$\text{Attention}(Q, K, Q) = \text{softmax}(\frac{QK^\intercal}{\sqrt{d_k}})V,$$

For *ship:*

```
attention_ship = (0.03 * embeddings_dict['i'] +
                  0.03 * embeddings_dict['must'] +
                  0.03 * embeddings_dict['go'] +
                  0.03 * embeddings_dict['back'] +
                  0.04 * embeddings_dict['to'] +
                  0.05 * embeddings_dict['my'] +
                  0.55 * embeddings_dict['ship'] +
                  0.03 * embeddings_dict['and'] +
                  0.05 * embeddings_dict['to'] +
                  0.03 * embeddings_dict['my'] +
                  0.13 * embeddings_dict['crew'])
```

where the *ship* vector received 13% of its value from *crew*

The Complete Input



Code Example

Jupyter Notebook:
https://github.com/pnugues/pnlp/tree/main/notebooks
(Second part of the notebook)

Multihead Attention (I)

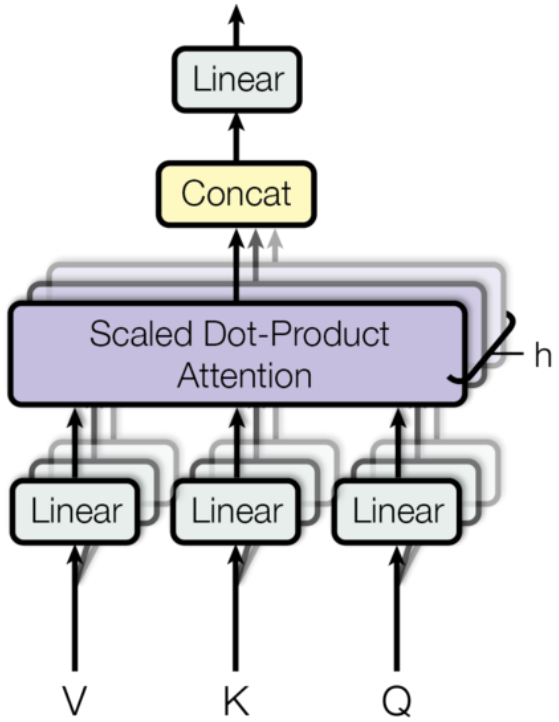
This attention is preceded by dense layers:

- If X represents complete input sequence (all the tokens), we have:

$$\begin{aligned} Q &= XW^Q, \\ K &= XW^K, \\ V &= XW^V. \end{aligned}$$

- And followed by another dense layer: W_O

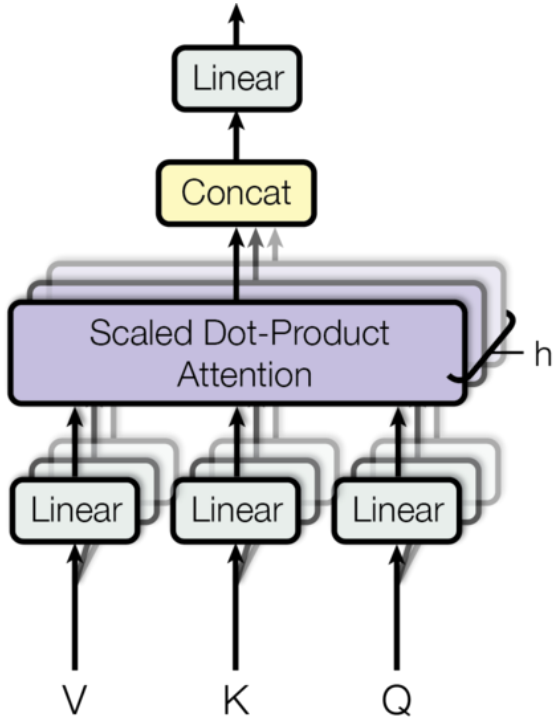
The dimension of the input (the size of the embeddings) is denoted d_{model}, for instance 100



From *Attention is all you need*, Vaswani et al. (2017)

Multihead Attention (II)

- The complete architecture has parallel attentions with h outputs (called heads)
- The h heads identify and isolate different kinds of information
- We recombine them with a concatenation
- This corresponds to the two last modules concat and linear (W^O)
- To keep the same output dimension, W^Q, W^K, and W^V have a size of $(d_{model} \times d_{model}/h)$

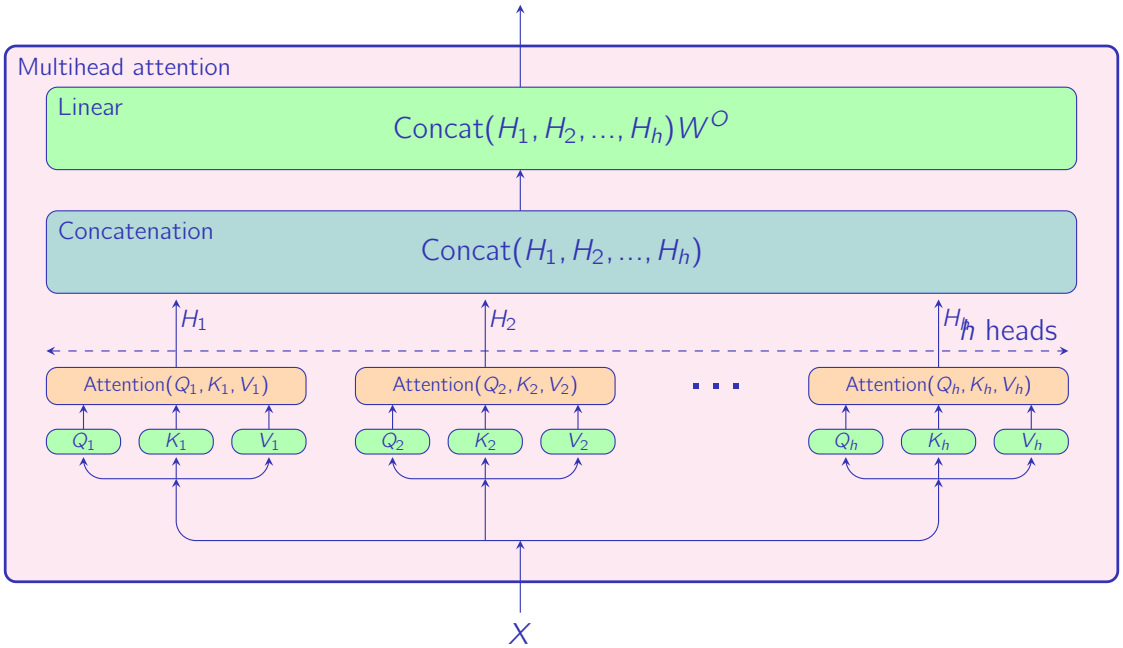


From *Attention is all you need*, Vaswani et al. (2017)

Code Example

PyTorch has an implementation of this architecture with the `https://pytorch.org/docs/stable/generated/torch.nn.MultiheadAttention.html` layer.

Experiment: Jupyter Notebook: `https://github.com/pnugues/pnlp/tree/main/notebooks` (Third part of the notebook)



Transformers: The Encoder

In transformers, the encoder is a structure, where:

1. The first part of the layer is a multihead attention;
2. We reinject the input to the attention output in the form of an addition:

$$X + \text{Attention}(Q, K, Q).$$

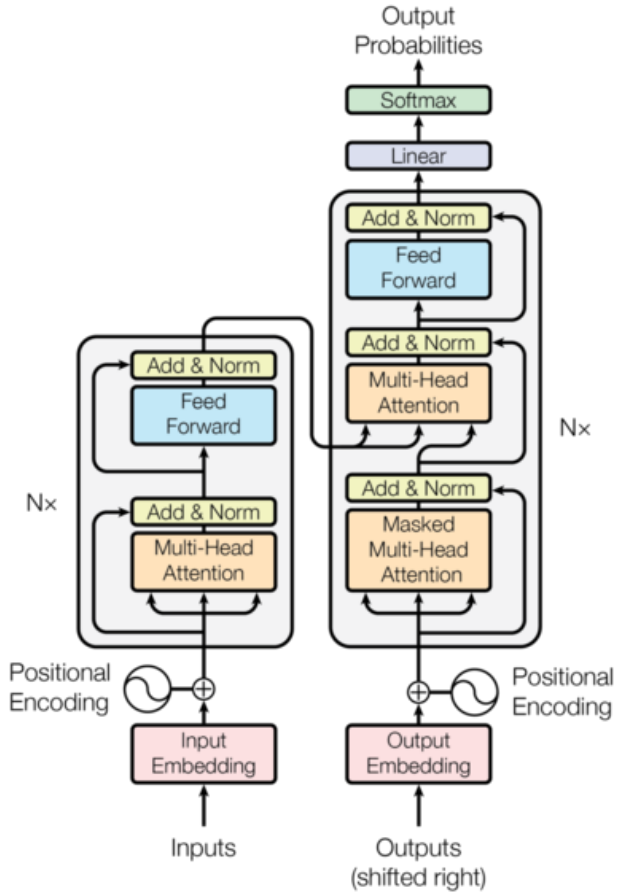
   This operation is called a skip or residual connection, which improves stability.

3. The result in then normalized per instance, i.e. a unique sequence, defined as:

$$x_{i,j_{norm}} = \frac{x_{i,j} - \bar{x}_{i,.}}{\sigma_{x_{i,.}}}.$$

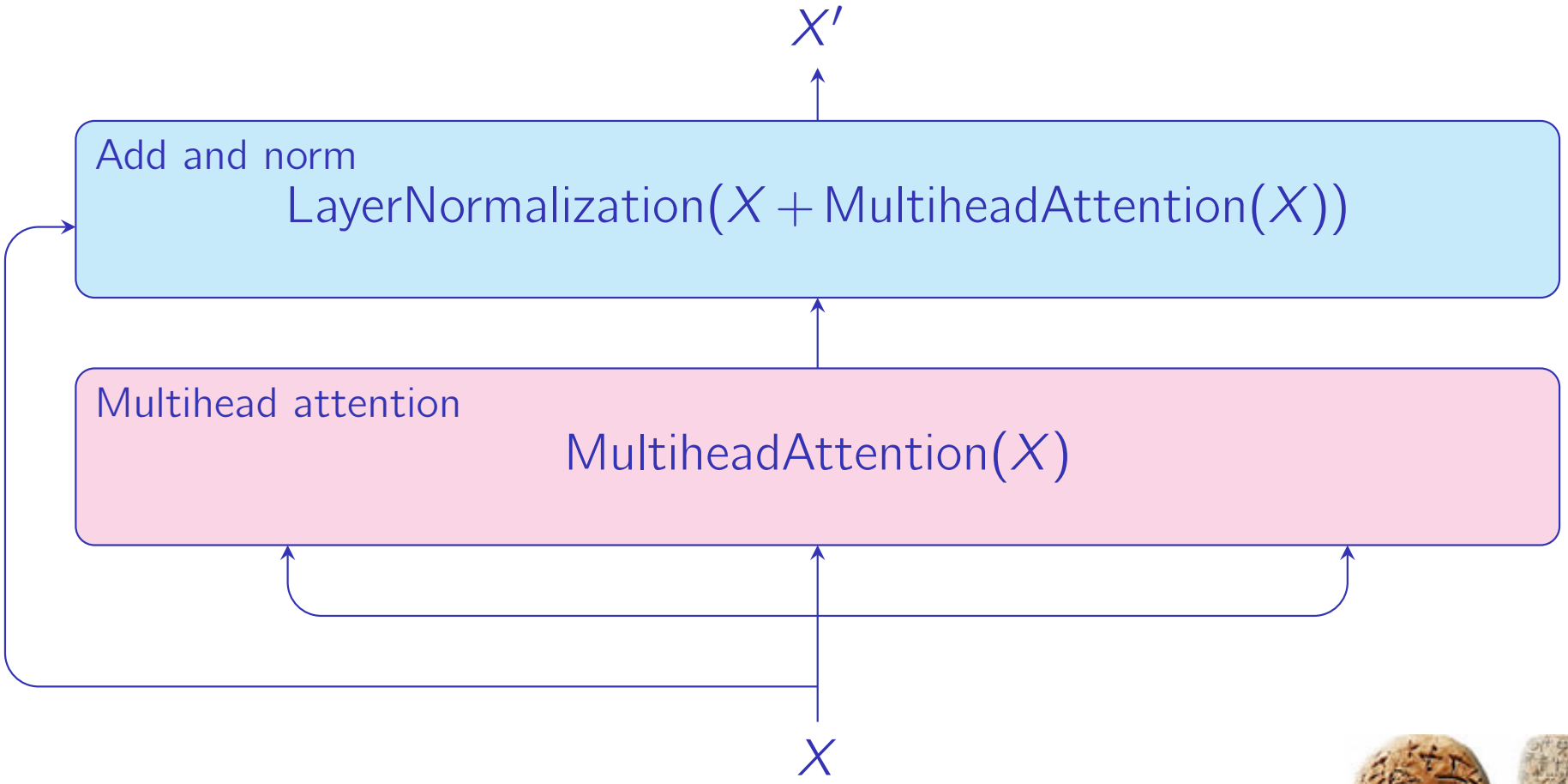
   The input distribution is more stable and improves the convergence

4. It is followed by dense layers.



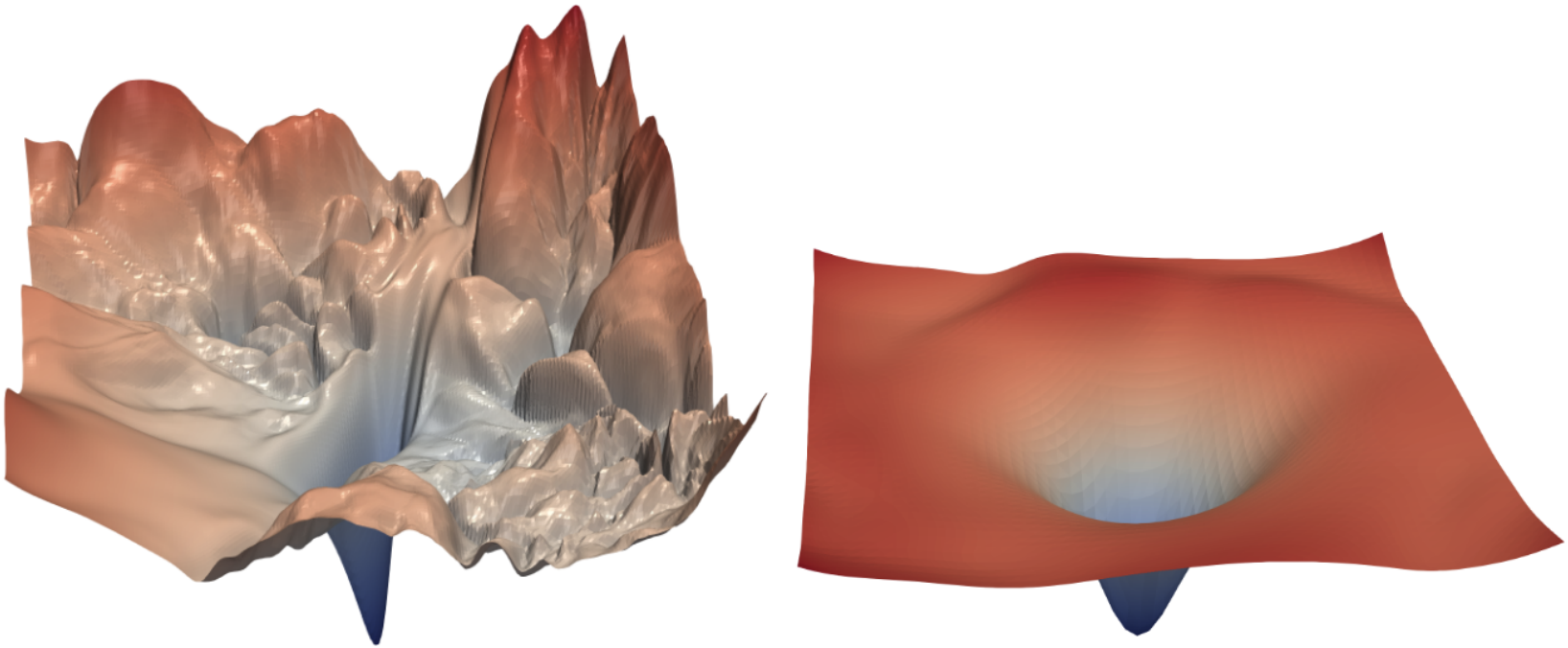
Left part, from *Attention is all you need*, Vaswani et al. (2017)

Residual Networks



Residual Network Loss

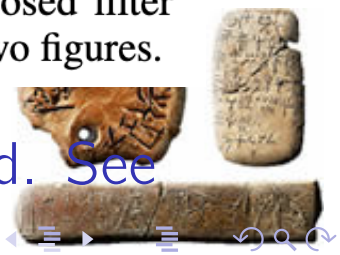
Experimental results show a better loss profile. Source: https://arxiv.org/abs/1712.09913



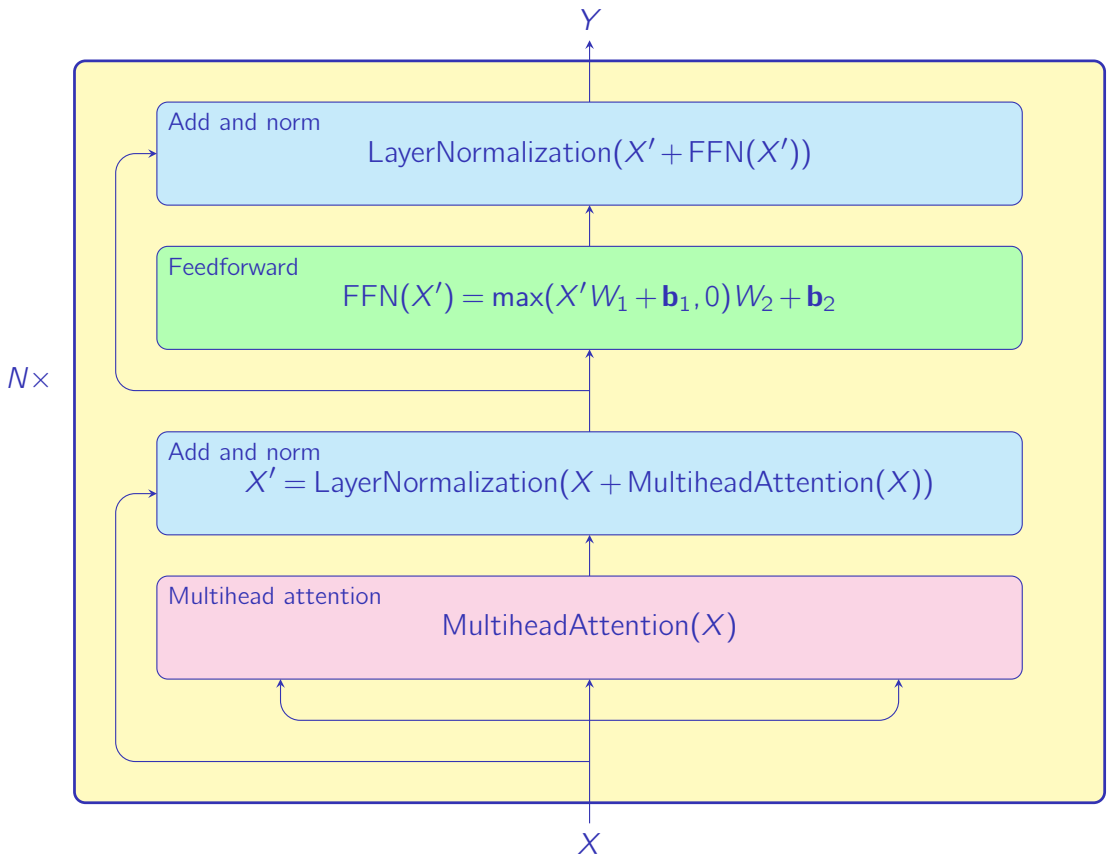
(a) without skip connections (b) with skip connections

Figure 1: The loss surfaces of ResNet-56 with/without skip connections. The proposed filter normalization scheme is used to enable comparisons of sharpness/flatness between the two figures.

Note that the residual networks are not completely understood. See https://arxiv.org/abs/1911.07013



The Encoder


Y
Add and norm
LayerNormalization(X′ + FFN(X′))
Feedforward
FFN(X′) = max(X′W1 + b1, 0)W2 + b2
N×
Add and norm
X′ = LayerNormalization(X + MultiheadAttention(X))
Multihead attention
MultiheadAttention(X)
X

Positional Encoding

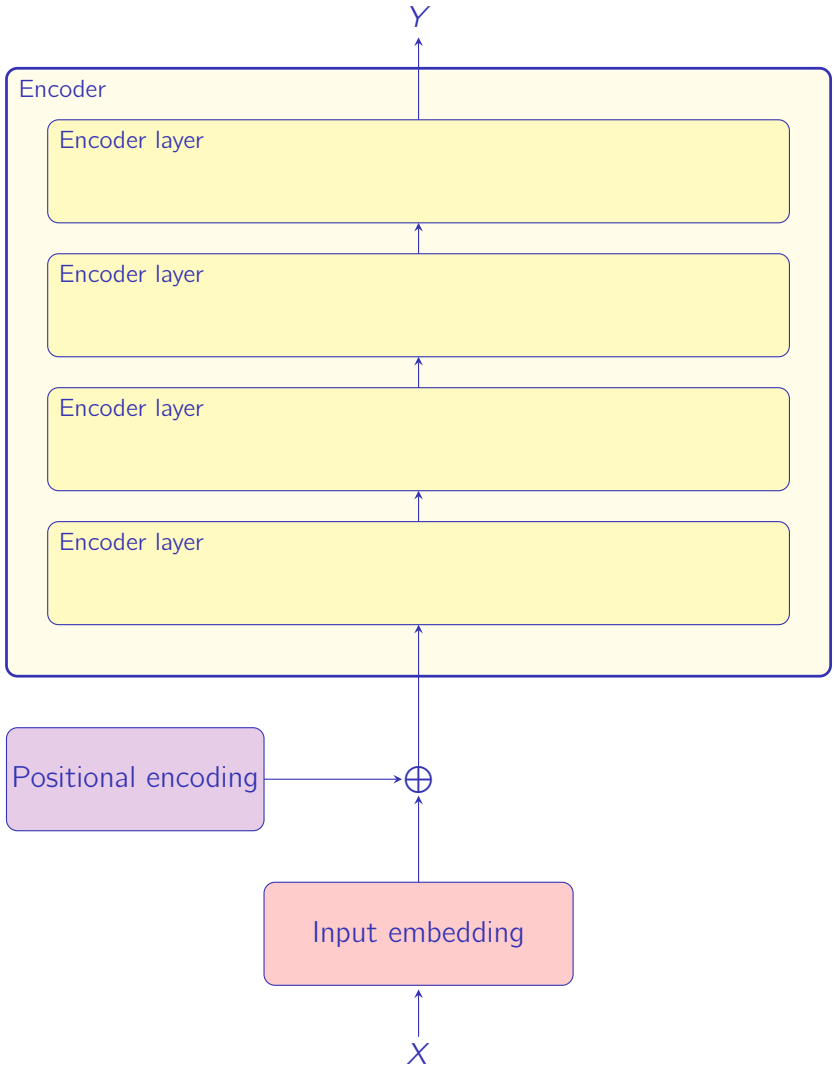
Vaswani et al. proposed two techniques to add information on the word positions. Both consist of vectors of dimension d_{model} that are summed with the input embeddings:

- The first one consists of trainable position embeddings, i.e. index i is associated with a vector of dimension d_{model} that is summed with the embedding of the input word at index i;
- The other consists of fixed vectors encoding the word positions. For a word at index i, they are defined by two functions:

$$PE(i,2j) = \sin\left(\frac{i}{10000^{\frac{2j}{d_{model}}}}\right),$$

$$PE(i,2j+1) = \cos\left(\frac{i}{10000^{\frac{2j}{d_{model}}}}\right).$$

Positional Encoding



PyTorch Encoders

PyTorch has a class to create an encoder layer (https://pytorch.org/docs/stable/generated/torch.nn.TransformerEncoderLayer.html):

```
encoder_layer = nn.TransformerEncoderLayer(d_model, nheads)
enc_output = encoder_layer(input)
```

and another to create a stack of N layers (https://pytorch.org/docs/stable/generated/torch.nn.TransformerEncoder.html#torch.nn.TransformerEncoder):

```
encoder = nn.TransformerEncoder(encoder_layer, num_layers)
```

```
encoder = nn.TransformerEncoder(encoder_layer, 6)
enc_output = encoder(input)
```

Input Format for Batches

The X default input format is different from what we have seen before. We need to build two lists: one for the input and the other for the output

y	DET	NOUN	VERB	DET	NOUN

x	The	waiter	brought	the	meal

All the vectors in a same batch must have the same length. We pad them:

y	PAD	PAD	PAD	DET	NOUN	VERB	DET	NOUN

x	PAD	PAD	PAD	The	waiter	brought	the	meal

We could apply the padding after

Building the Sequences

```
def build_sequences(corpus_dict, key_x='form', key_y='pos',
                    tolower=True):
    X, Y = [], []
    for sentence in corpus_dict:
        x, y = [], []
        for word in sentence:
            x += [word[key_x]]
            y += [word[key_y]]
        if tolower:
            x = list(map(str.lower, x))
        X += [x]
        Y += [y]
    return X, Y
```

At this point, we have **x** and **y** vectors of symbols

Building Index Sequences

0 is for the padding symbol and 1 for the unknown words

```
idx_word = dict(enumerate(vocabulary_words, start=2))
idx_pos = dict(enumerate(pos, start=2))
word_idx = {v: k for k, v in idx_word.items()}
pos_idx = {v: k for k, v in idx_pos.items()}
```

At this point, we have **x** and **y** vectors of numbers

Batch First

Batch-first ordering with these segments:

Sing, O goddess, //the anger // of Achilles son of Peleus, // that brought countless ills // upon the Achaeans.

$$X = \begin{bmatrix} \text{sing} & \text{o} & \text{goddess} & \text{PAD} & \text{PAD} \\ \text{the} & \text{anger} & \text{PAD} & \text{PAD} & \text{PAD} \\ \text{of} & \text{achilles} & \text{son} & \text{of} & \text{peleus} \\ \text{that} & \text{brought} & \text{countless} & \text{ills} & \text{PAD} \\ \text{upon} & \text{the} & \text{achaeans} & \text{PAD} & \text{PAD} \end{bmatrix}$$

PyTorch uses an optimized tensor ordering:

$$X = \begin{bmatrix} \text{sing} & \text{the} & \text{of} & \text{that} & \text{upon} \\ \text{o} & \text{anger} & \text{achilles} & \text{brought} & \text{the} \\ \text{goddess} & \text{PAD} & \text{son} & \text{countless} & \text{achaeans} \\ \text{PAD} & \text{PAD} & \text{of} & \text{ills} & \text{PAD} \\ \text{PAD} & \text{PAD} & \text{peleus} & \text{PAD} & \text{PAD} \end{bmatrix}$$

To use the batch-first convention, you have to set `batch_first=True`.

Padding the Index Sequences

We build the complete X_idx and Y_idx matrices for the whole corpus
And we pad the matrices:

```
X_train_padded = pad_sequence(X_train_idx, batch_first=True)
Y_train_padded = pad_sequence(Y_train_idx, batch_first=True)

X_val_padded = pad_sequence(X_val_idx, batch_first=True)
Y_val_padded = pad_sequence(Y_val_idx, batch_first=True)
```

See: `https://pytorch.org/docs/stable/generated/torch.nn.utils.rnn.pad_sequence.html`
`pad_sequences` can have an argument that specifies the padding value `padding_value`
The padded sentences must have the same length in a batch. This is automatically computed by PyTorch

Padding Mask

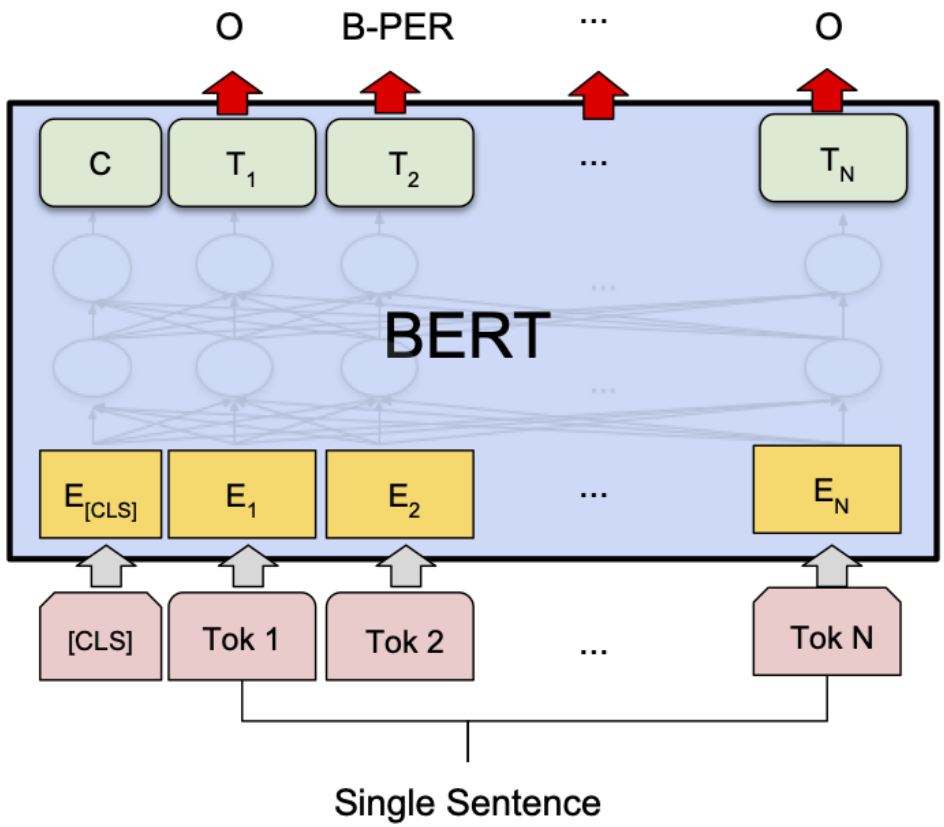
We had to remove the padding symbols from the attention
We use a padding mask, assuming index 0 is the PAD index:

```
>>> padding_mask = (X_idx == 0)
>>> padding_mask
tensor([[False, False, False, False, False, False, False,
         False, False, False, False],
        [False, False, False, False, False, False,  True,
         True,  True,  True, True]])
```

We pass this mask to the encoder:

```
X = self.encoder(
            X, src_key_padding_mask=padding_mask)
```

Application: Sequence Tagging



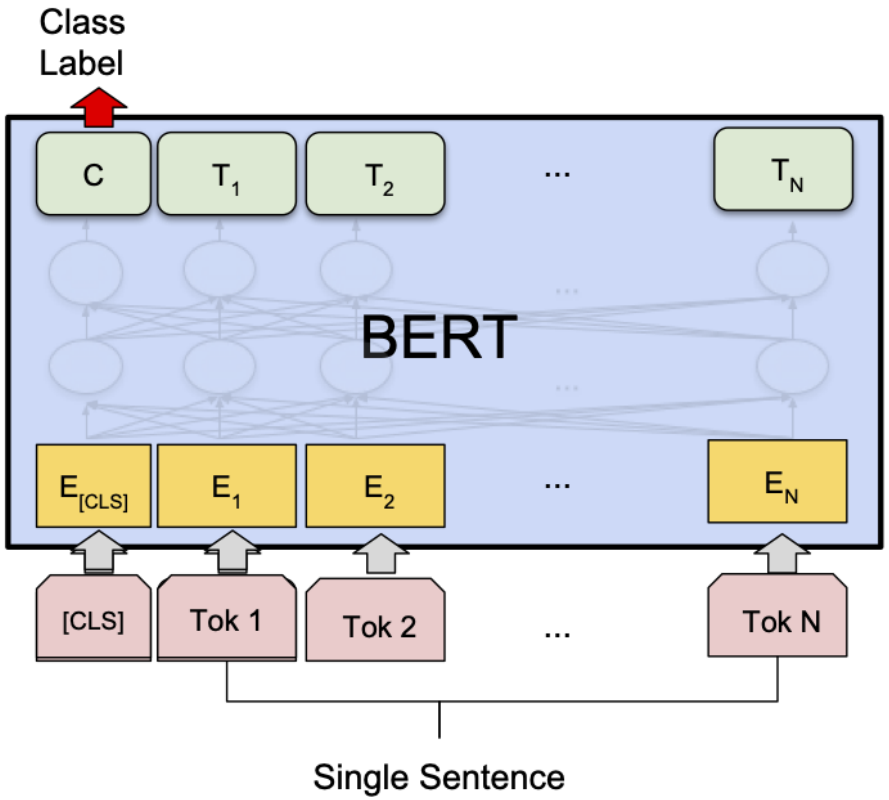
(d) Single Sentence Tagging Tasks: CoNLL-2003 NER

from Devlin et al., *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*, 2019

Code Example

Experiment: Jupyter Notebook:
https://github.com/pnugues/pnlp/tree/main/notebooks

Application: Sentence Classification



(b) Single Sentence Classification Tasks: SST-2, CoLA

from Devlin et al., *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*, 2019