



# Lecture 12: Transformers (Part 2)

Data C182 (Fall 2024). Week 07. Tuesday Oct 8th, 2024

Speaker: Eric Kim

# Announcements

- HW01 due tonight! Tues Oct 8th 11:59 PM PST
  - Reminder: Homework slip day policy [[link](#)]
  - DSP students: for those with "Assignment extension" accommodation, we have already extended your due date on Gradescope.
    - If you are expecting one, but haven't received the extension, please make a private Ed post!
- HW02 ("RNNs") out!
  - Due: Thurs Oct 24th, 11:59 PM PST
  - In Colab (pewh!)

# Announcements

- Reminder: Midterm is coming up!
  - Tuesday, October 22th 2024, 6:30 PM - 8 PM.
  - Location: ~50% in 10 Evans, ~50% in Physics 1
    - We'll send exam room assignments to students shortly
  - If you're unable to make this time, please contact us ASAP (make a private Ed post)
  - Midterm will cover everything from:
    - Lectures, discussions, HW01+HW02
  - In-person, paper + pencil exam.
  - **DSP:** if you need exam accommodations, please contact us ASAP (private post on Ed)

# Today's lecture

- Transformers (Part 2!)
- Multi-head self attention (MHA): deep dive
- Encoders
  - Text classification, image classification

# Problem setup: sequences

- So far: we've mainly focused on "point predictions"
  - Ex: given an image, what category is it? (CIFAR-10, from HW01!)
- Many problems are instead naturally described via sequences
  - Given a video (a sequence of audio and visual features), what is happening in the video? (action classification)
  - Given an input text sequence:
    - Generation: what should I respond with?
    - Translation: translate from English to French?
    - Classification: does this sentence have positive sentiment?

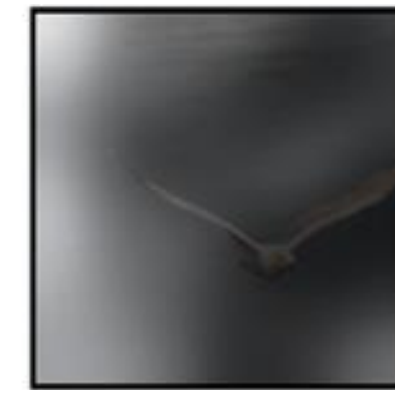
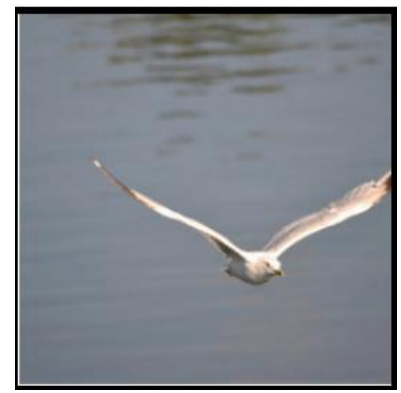
# Attention

- Motivation: given an input  $X$ , when producing a prediction/output  $Y$ , we want the model to tell us "why" it returned  $Y$ 
  - "Explainable AI"
- One popular approach: "Attention"
- Design your model such that the model considers certain parts of the input  $X$  more "important" than others
- Frequently visualized in papers as "attention masks"



# Motivation / intuition

**Observation:** model learns to correlate different regions of the image with being "relevant" to a given word.



A

bird

flying

over

a

body

of

water

.

## Problem setting:

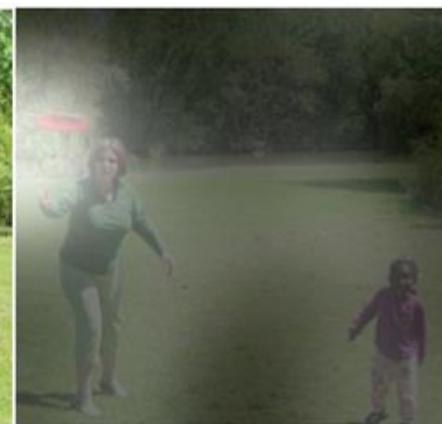
Image captioning.

**Input:** image.

**Output:** text caption describing the image.



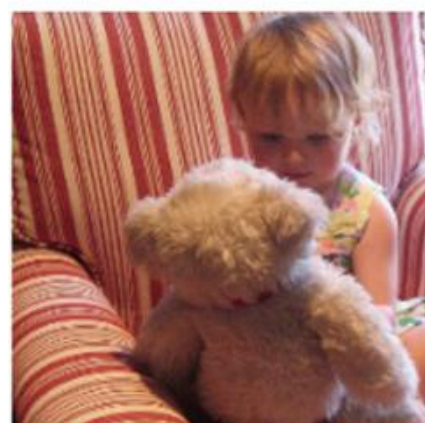
A woman is throwing a frisbee in a park.



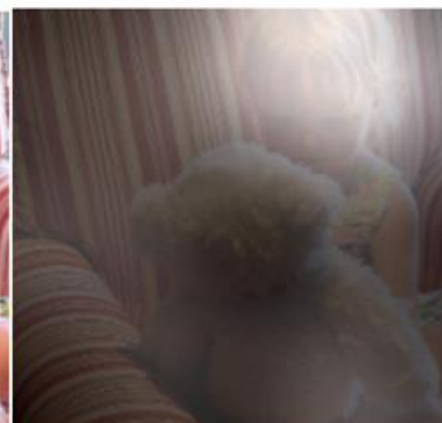
A dog is standing on a hardwood floor.



A stop sign is on a road with a mountain in the background.



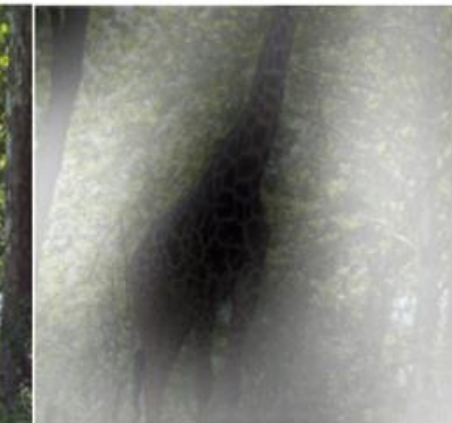
A little girl sitting on a bed with a teddy bear.



A group of people sitting on a boat in the water.



A giraffe standing in a forest with trees in the background.





# "Attention Is All You Need" (2017)

- Paper that introduced the "Transformers" model architecture [\[link\]](#)
- Highly influential and impactful
  - Ex: powers Chat-GPT!
  - Basically, transformers is now ubiquitously used in AI/ML
- Originally focused on the text domain: machine translation, and sentence parsing ("English constituency parsing")
  - Now, transformers are used in other domains like: images, videos, user actions, etc.

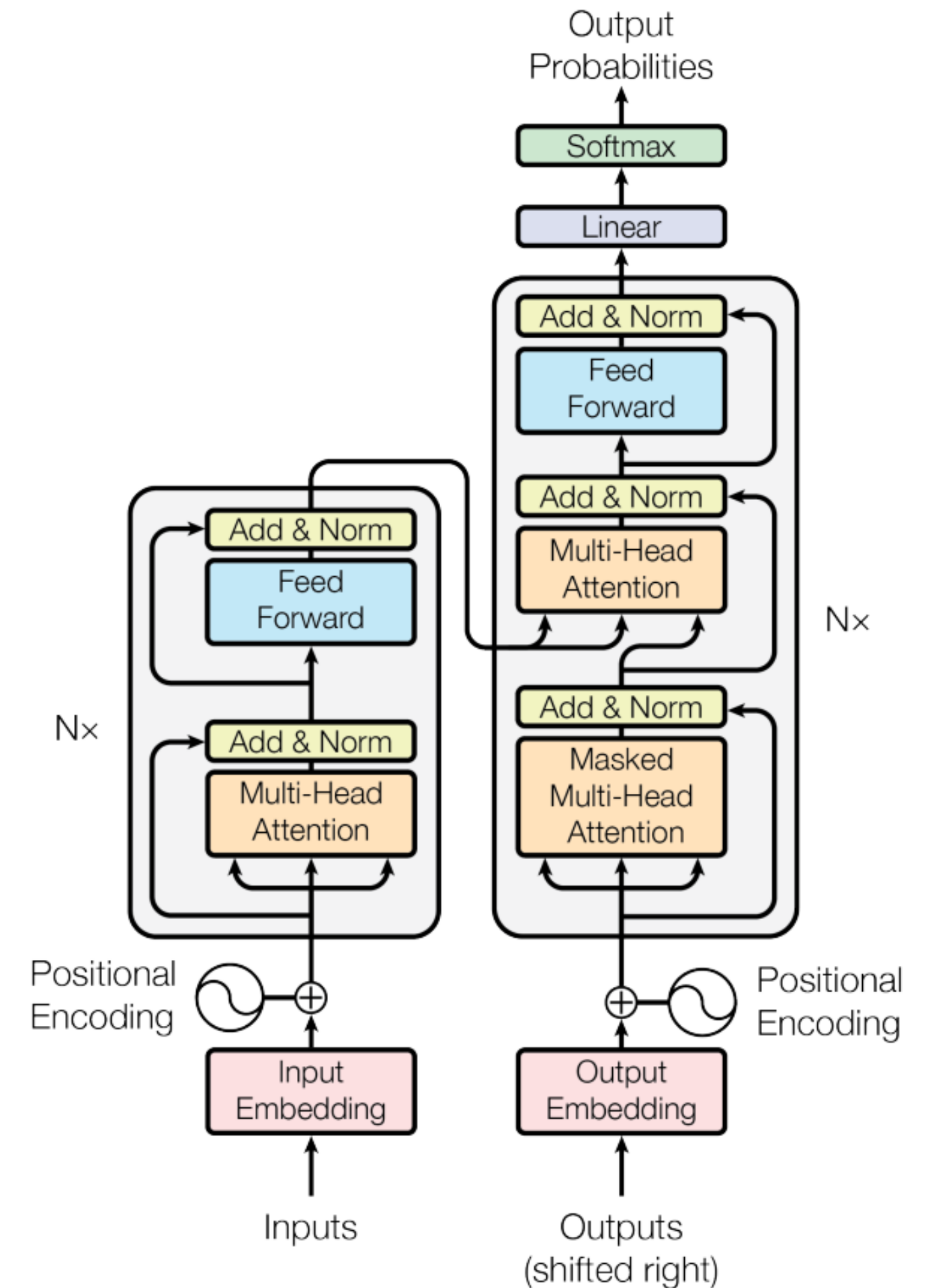


Figure 1: The Transformer - model architecture.



# Multihead self attention (MHA)

- General idea: MHA layer does two things
  - (1) **Self-attention**. Learn which parts of the input are important to produce the desired prediction.
  - (2) **Attention-aware transformation**. Transform the input features in a way where the "important" parts are highlighted (attention) to learn a stronger representation.
- Notably, MHA is good at capturing "long range" interactions, something that RNNs have historically had difficulty with

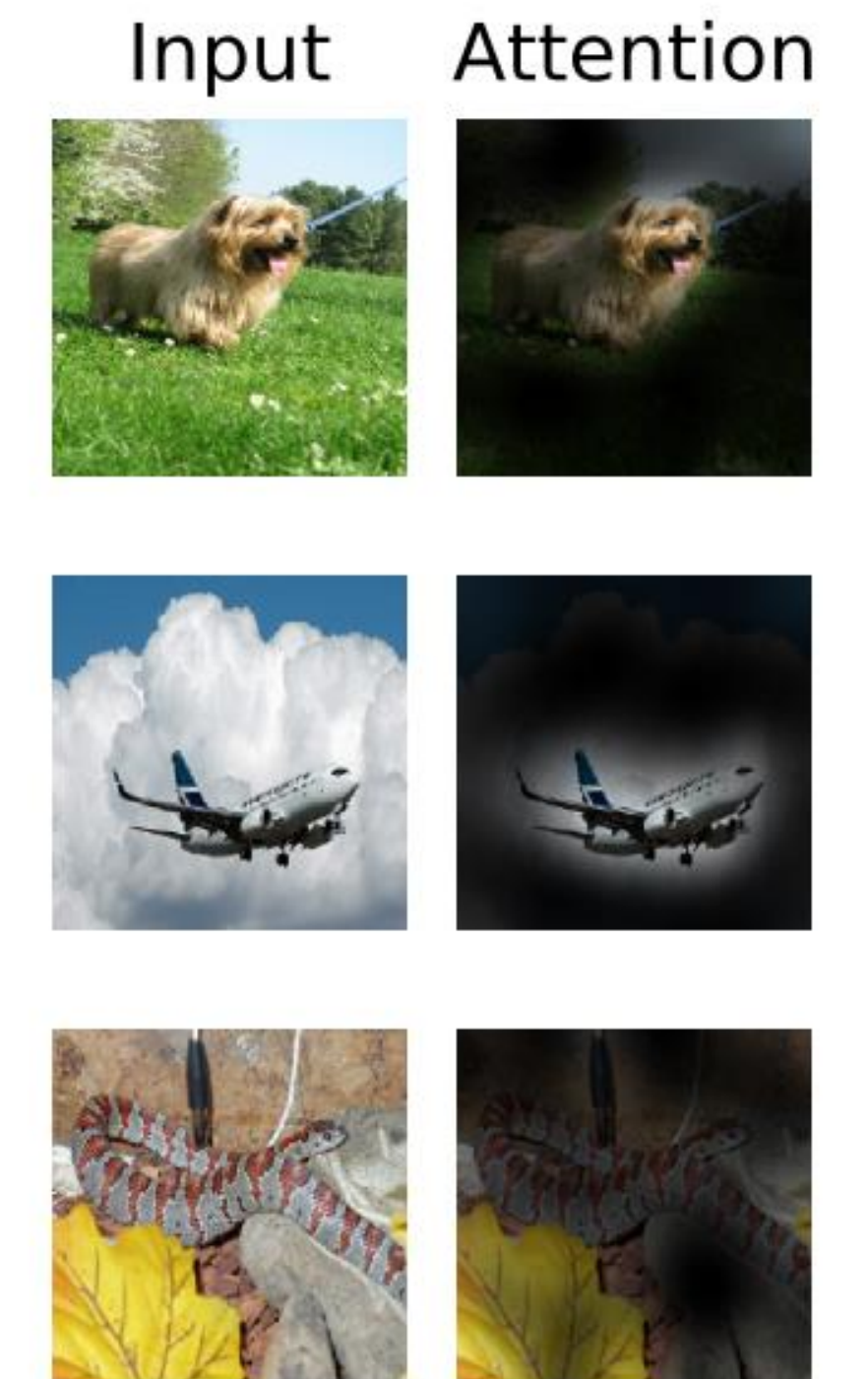


Figure 6: Representative examples of attention from the output token to the input space. See Appendix D.7 for details.

# MHA (v0)

- Exercise: Let's build up, step-by-step, to the "full" MHA
- Note: to simplify things (somewhat), let's assume we're working with a single input sequence  $X$ , and `batchsize=1` (`shape=[seq_len, d]`).
  - Aka we're looking at just the Encoder (more on that soon)

# MHA (v0): attention scores

- First up: how do we compute attention scores?
- **Question**: given an input  $X$  with shape=[seq\_len, d], what should the attention scores shape be?
  - **Answer**: [seq\_len, seq\_len]! Ex:  
attn\_scores[0, 2] tells me "how important is token 2 to token 0?"
- **Question**: suppose I want to make the attention scores more interpretable (eg as probability scores), and have the rows of the attention scores sum to 1.0: `attn_scores[0, :].sum() == 1.0`. How can I achieve this?
  - **Answer**: `softmax()` across each row!

Self-attention  
Probability score matrix

	Hello	I	love	you
Hello	0.8	0.1	0.05	0.05
I	0.1	0.6	0.2	0.1
love	0.05	0.2	0.65	0.1
you	0.2	0.1	0.1	0.6

...

Softmax()  
Softmax()

# MHA (v0): attention scores

- **Question**: what is the simplest way you can think of to calculate self-attention? ( $X$  shape=[seq\_len, d])
- **Answer**: one way is to multiply all pair-wise dot products between rows of  $X$ :
  - $\text{attn\_scores} = \text{softmax}(X * X^T, \text{dim}=1)$
- Pro: simple
- Con: we're not learning anything here. What if  $X$  isn't good at calculating attention scores?
  - It's not "deep" enough...
  - Let's learn some transformations of  $X$ !

Self-attention  
Probability score matrix

	Hello	I	love	you
Hello	0.8	0.1	0.05	0.05
I	0.1	0.6	0.2	0.1
love	0.05	0.2	0.65	0.1
you	0.2	0.1	0.1	0.6

→ Softmax()

...

→ Softmax()



# MHA (v0): Query, Key, Value

Let:  $X$  be  $[\text{seq\_len}, d]$

Step 1: Calculate Query, Key, and Value:

shape= $[\text{seq\_len}, d]$

$$\begin{aligned} Q &= XW^{(q)} \\ K &= XW^{(k)} \\ V &= XW^{(v)} \end{aligned}$$

$W^q, W^k, W^v$  are  $[d, d]$ ,  
and are learned linear  
transforms

Step 2: Use  $Q, K$  to calculate attention scores:

$$\text{attn\_scores} = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)$$

softmax is done  
over rows

Divide by  $\text{sqrt}(d)$  to avoid issues with  
vanishing gradients when  $d$  is large

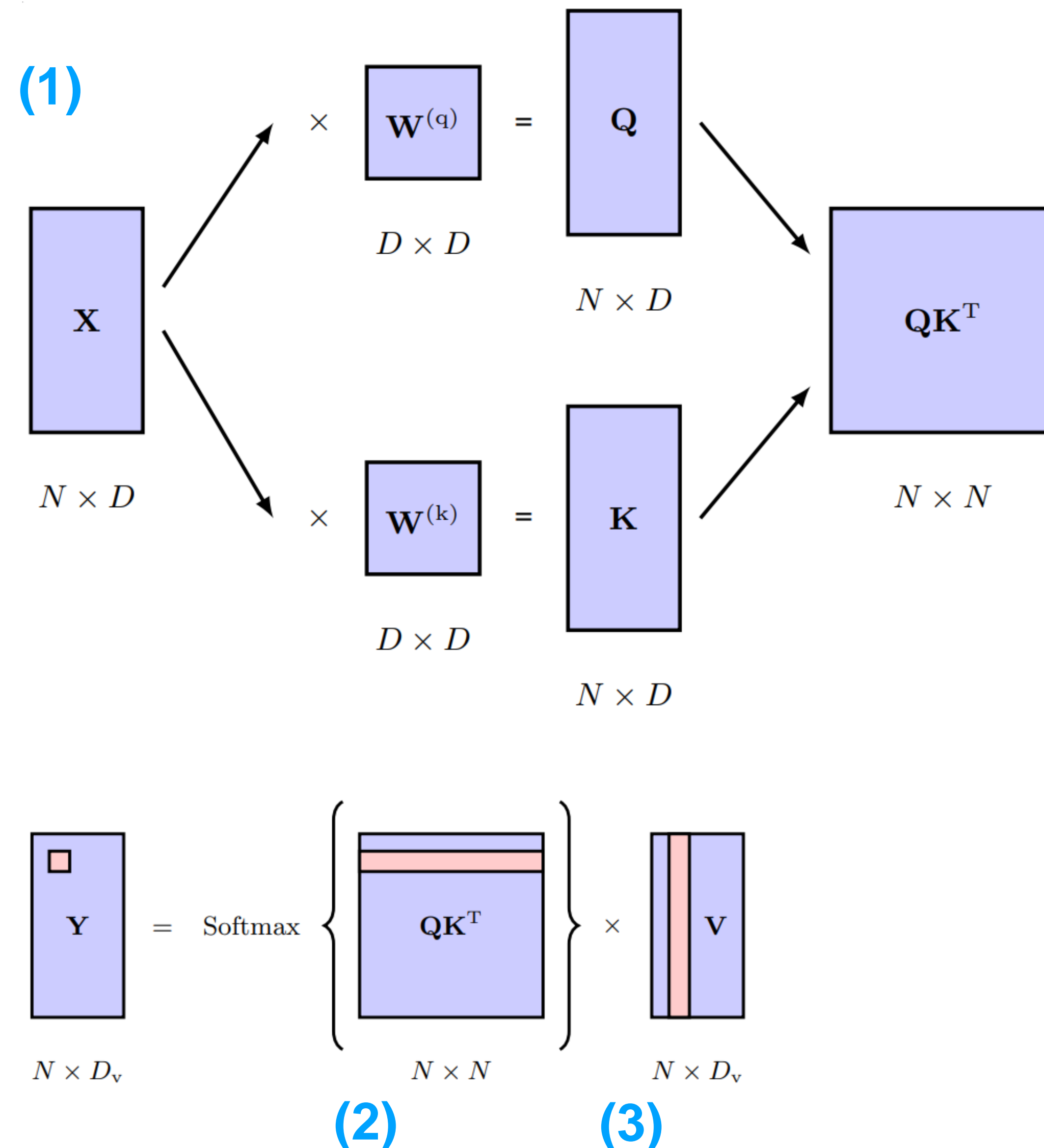
shape= $[\text{seq\_len}, \text{seq\_len}]$

Step 3: Compute final MHA output

shape= $[\text{seq\_len}, d]$

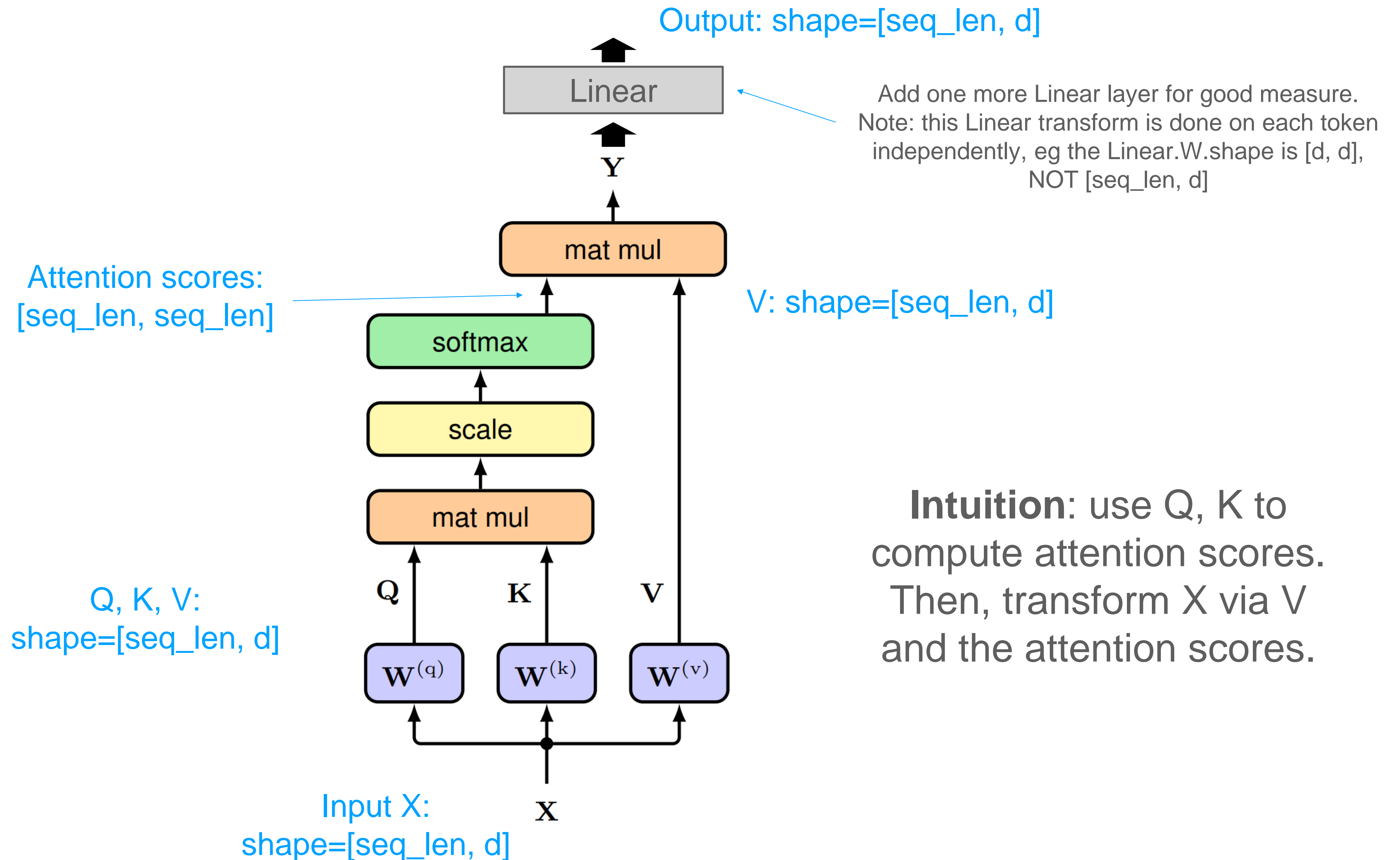
$$Y = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V$$

...and repeat! Can  
easily stack MHA  
layers



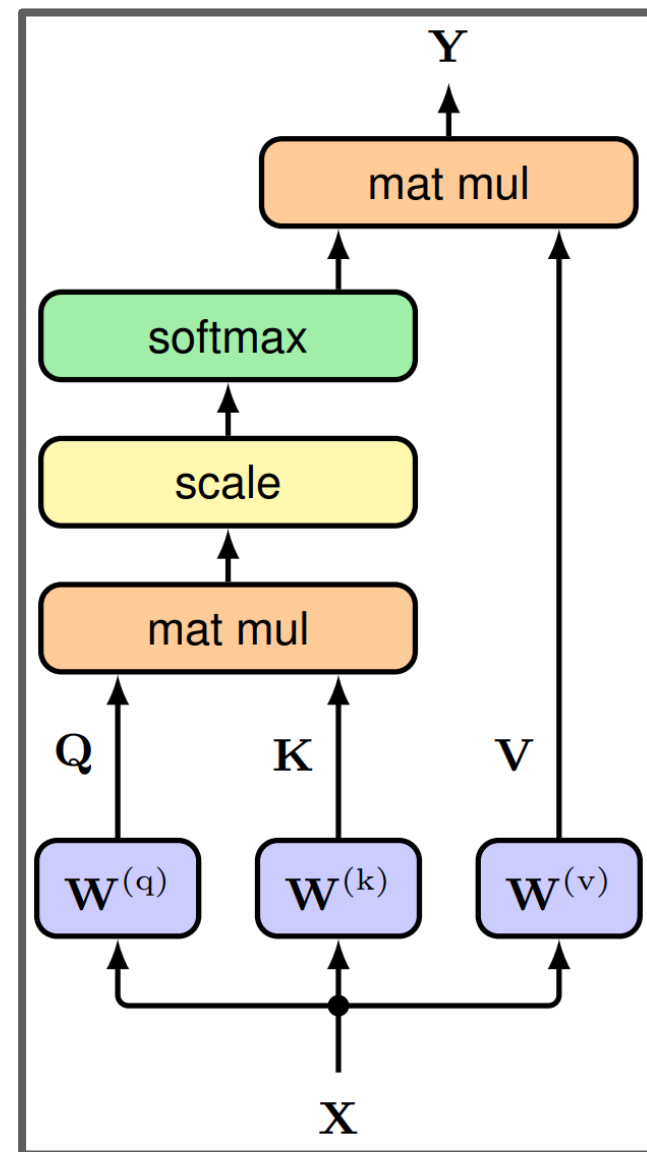


# MHA (v0): information flow

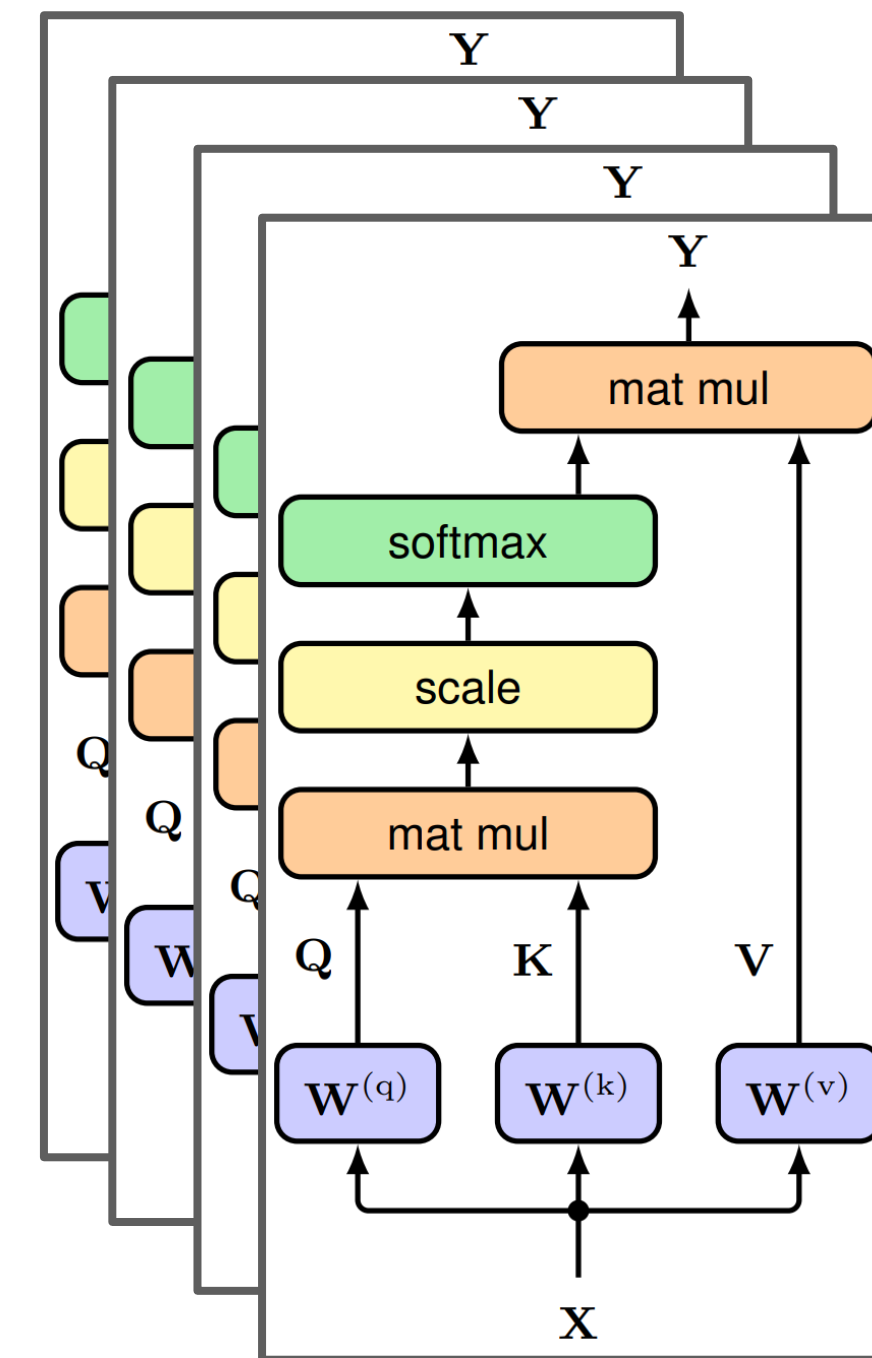


# MHA (v1): multiple heads

- Idea: let's learn multiple self-attention modules (prev slide) in parallel at a given level (eg "width" of network)



"single head"  
attention



"multi head"  
attention

# MHA (v1): multiple heads

Let `h` be the number of heads.

- (1) Run  $h$  different independent self-attention blocks, to produce  $h$  different outputs with shape=[seq\_len,  $d$ ]
- (2) **Concatenate** all  $h$  outputs, and apply a learned linear transformation to produce the MHA output (shape=[seq\_len,  $d$ ])

$$\mathbf{Q}_h = \mathbf{X} \mathbf{W}_h^{(q)}$$

$$\mathbf{K}_h = \mathbf{X} \mathbf{W}_h^{(k)}$$

$$\mathbf{V}_h = \mathbf{X} \mathbf{W}_h^{(v)}$$

$$\mathbf{H}_h = \text{Attention}(\mathbf{Q}_h, \mathbf{K}_h, \mathbf{V}_h)$$

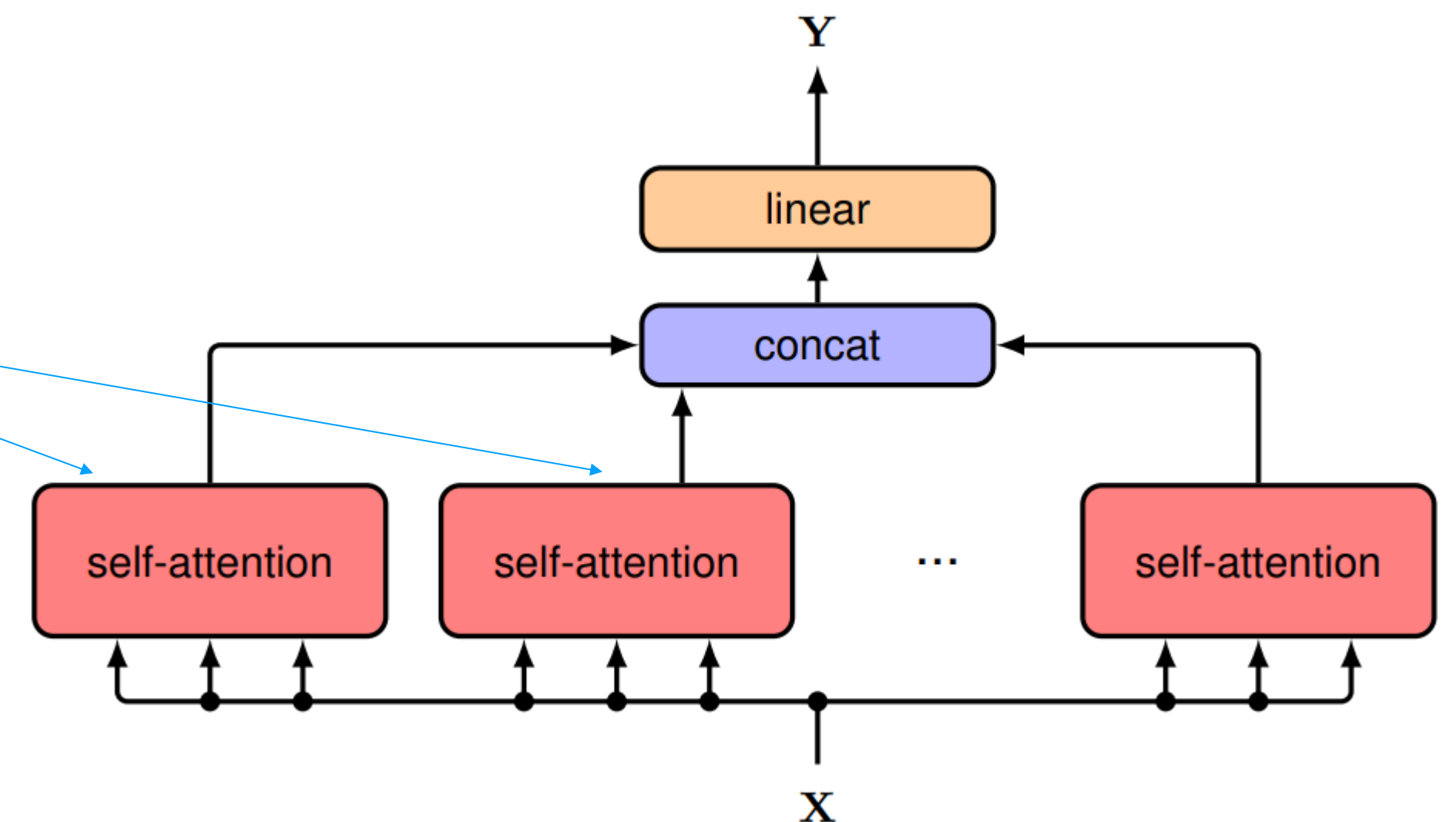
$$\mathbf{Y}(\mathbf{X}) = \text{Concat}[\mathbf{H}_1, \dots, \mathbf{H}_H] \mathbf{W}^{(o)}$$

shape=[seq\_len,  $h*d$ ]

Learned linear transform.  
Shape=[ $h*d$ ,  $d$ ]

$h$  different Q, K, V, and attention scores!

$$\begin{matrix} \begin{matrix} \mathbf{H}_1 & \mathbf{H}_2 & \dots & \mathbf{H}_H \end{matrix} & \times & \mathbf{W}^{(o)} & = & \mathbf{Y} \\ N \times HD_v & & HD_v \times D & & N \times D \end{matrix}$$



Nice property: we can easily stack MHA vertically ("deeper")!

Orig paper uses 6 layers, with  $h=8$  heads

# MHA (v1.5): multiple heads + split

In practice: to reduce computation costs, rather than have each self-attention module operate on the full embedding `d`, we **divide up the embeddings into `h` chunks**.

Example: for  $d=16$  and  $h=2$  heads,

Head0: work on first 8 embed dims:  $X[:, :8]$

Head1: work on last 8 embed dims:  $X[:, 8:]$

$$\begin{aligned} \mathbf{Q}_h &= \mathbf{X}_h \mathbf{W}_h^{(q)} && X_h \text{ shape}=[\text{seq\_len}, d_h] \\ \mathbf{K}_h &= \mathbf{X}_h \mathbf{W}_h^{(k)} && Q_h, K_h, V_h \\ &&& \text{Shape}=[\text{seq\_len}, d_h] \\ \mathbf{V}_h &= \mathbf{X}_h \mathbf{W}_h^{(v)} && W_h^q, W_h^k, V_h^v \\ &&& \text{Shape}=[d_h, d_h] \end{aligned}$$

$$d_h = \text{floor}\left(\frac{d}{h}\right)$$

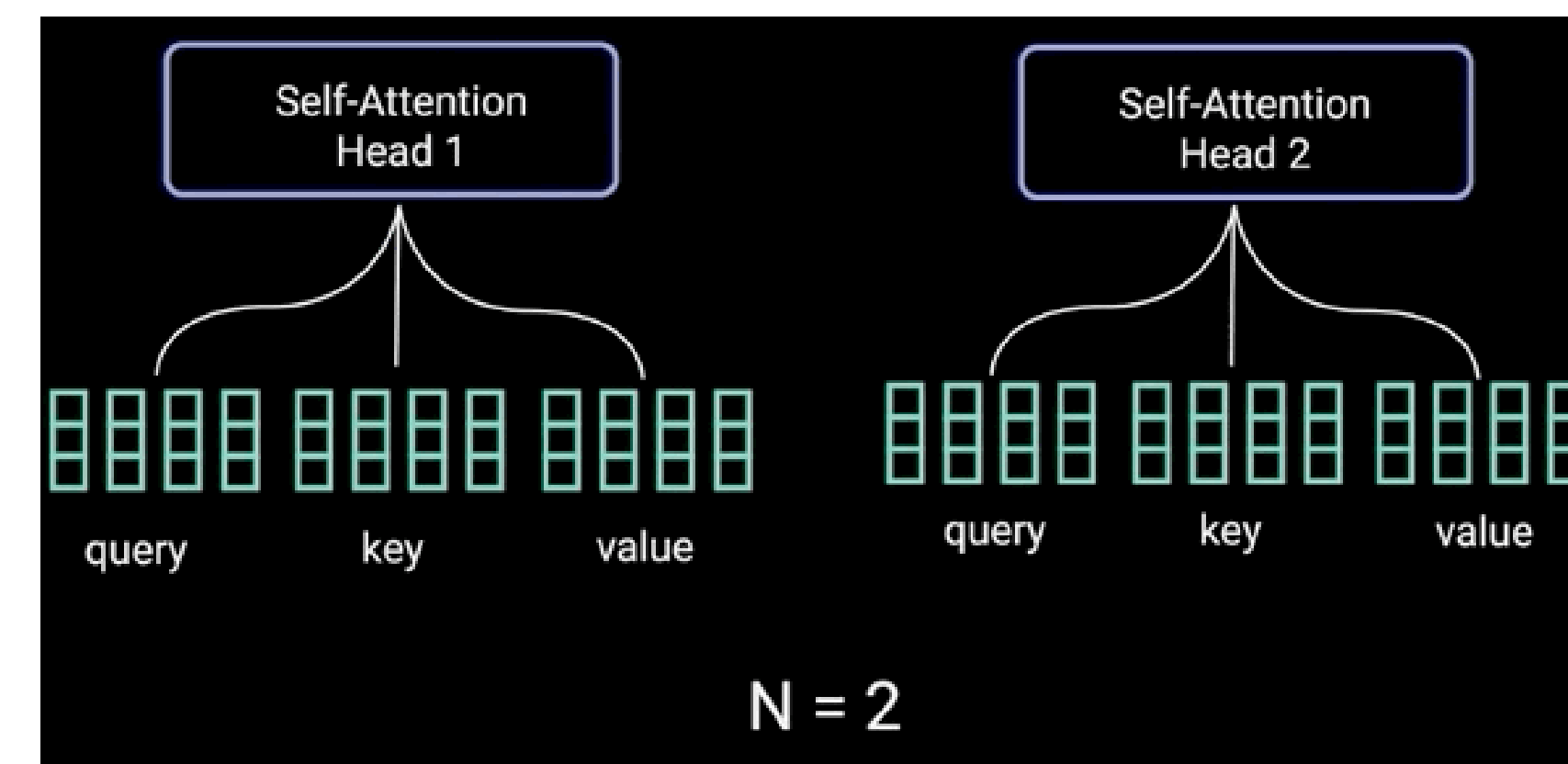
"effective" embed dimensionality for each head

$$\mathbf{H}_h = \text{Attention}(\mathbf{Q}_h, \mathbf{K}_h, \mathbf{V}_h)$$

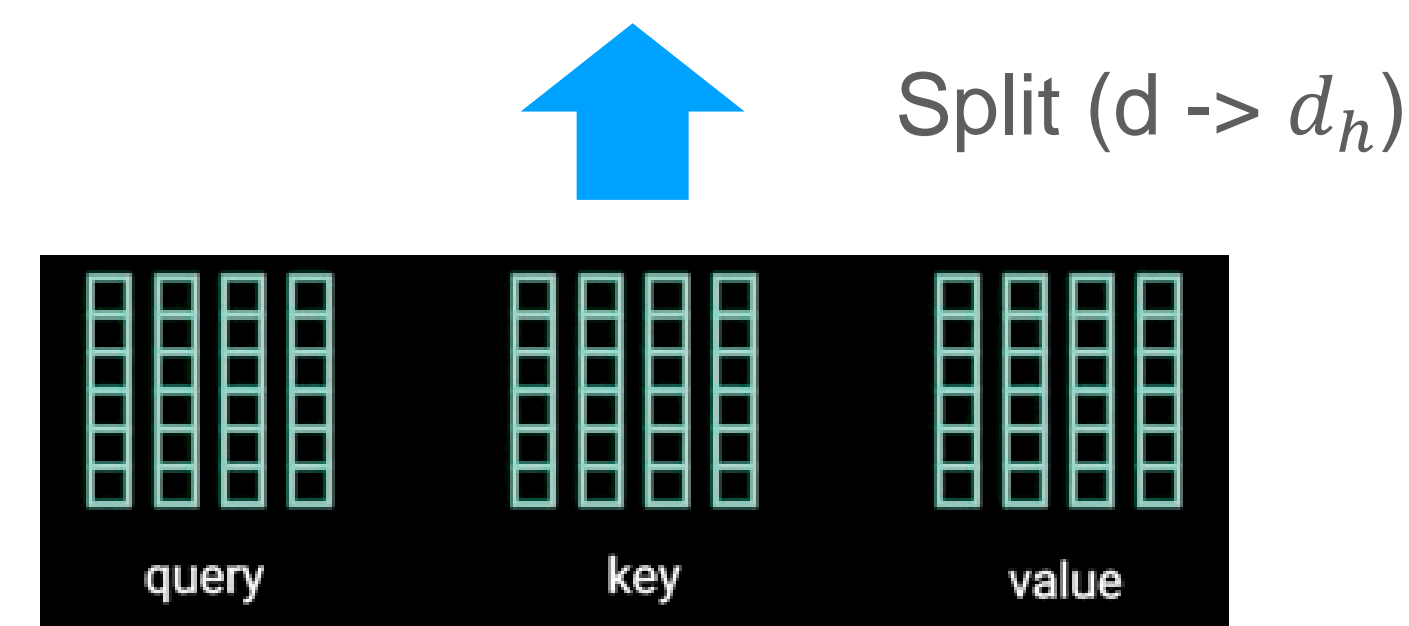
$$\mathbf{Y}(\mathbf{X}) = \text{Concat}[\mathbf{H}_1, \dots, \mathbf{H}_H] \mathbf{W}^{(o)}$$

shape=[seq\_len, h\*d\_h]  
=[seq\_len, d]

Learned linear transform.  
Shape=[d, d]



Splitting Q, K, V, N times before applying self-attention



**Implication:** with this embedding "splitting", a MHA with  $h$  heads (operating on  $d/h$  dims) is roughly the same computation cost as a MHA with 1 head but operating on the full embedding dimensionality.

# MHA: pytorch

```
import torch
input_seq_len = 8
batchsize = 2
embed_dim = 16
num_heads = 4
print(f"Scenario: input_seq_len={input_seq_len} batchsize={batchsize}
embed_dim={embed_dim} num_heads={num_heads}")

# https://pytorch.org/docs/stable/generated/torch.nn.MultiheadAttention.html
mha = torch.nn.MultiheadAttention(
    embed_dim=embed_dim,
    num_heads=num_heads,
    batch_first=True,
)

input_seq = torch.rand(size=[batchsize, input_seq_len, embed_dim],
dtype=torch.float32)
print(f"input_seq.shape: {input_seq.shape}")

out_mha, attn_weights = mha(query=input_seq, key=input_seq, value=input_seq,
need_weights=True)

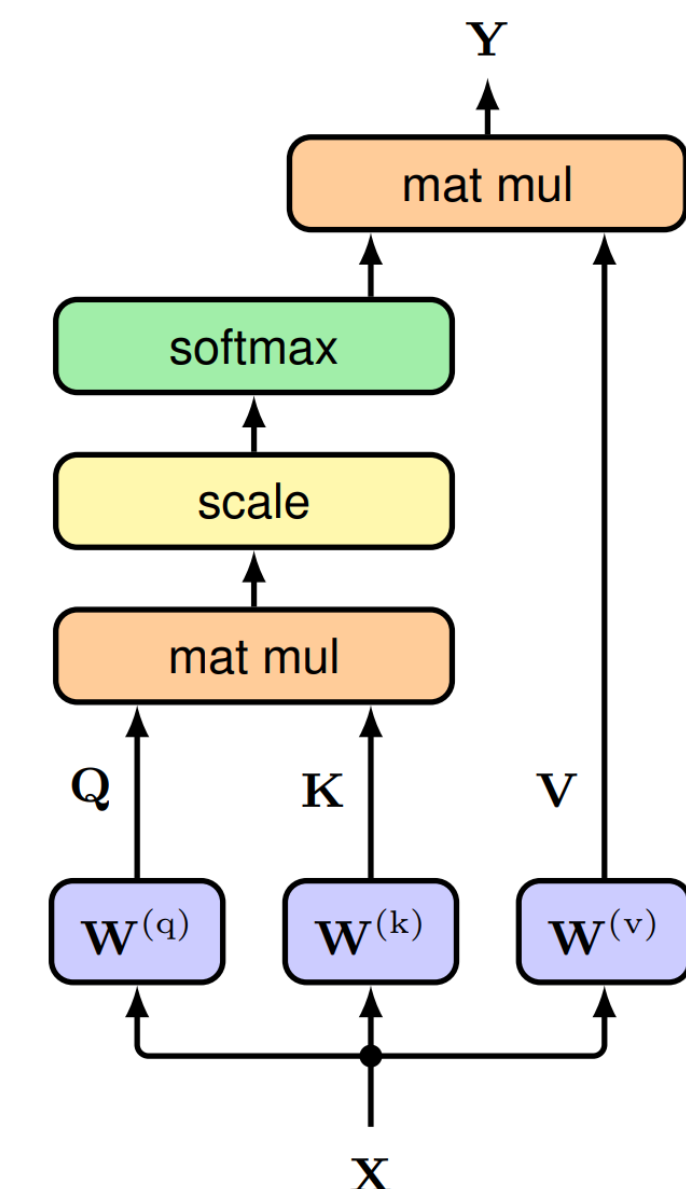
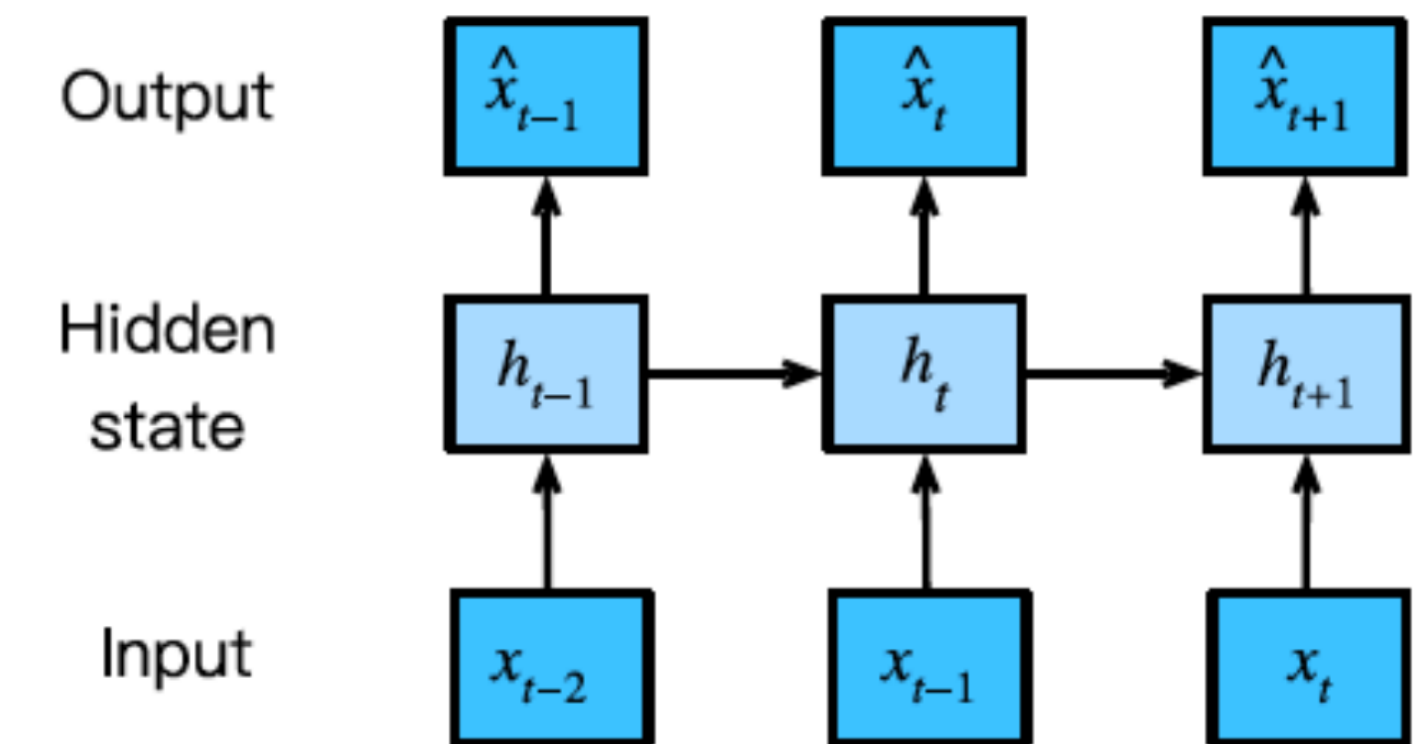
print(f"out_mha.shape: {out_mha.shape}, attn_weights.shape:
{attn_weights.shape}")
out_proj_layer = mha.out_proj
print(f"out_proj_layer.weight.shape: {out_proj_layer.weight.shape}")
```

```
# Output
Scenario: input_seq_len=8 batchsize=2
embed_dim=16 num_heads=4
input_seq.shape: torch.Size([2, 8, 16])
out_mha.shape: torch.Size([2, 8, 16]),
attn_weights.shape: torch.Size([2, 8, 8])
out_proj_layer.weight.shape: torch.Size([16, 16])
```



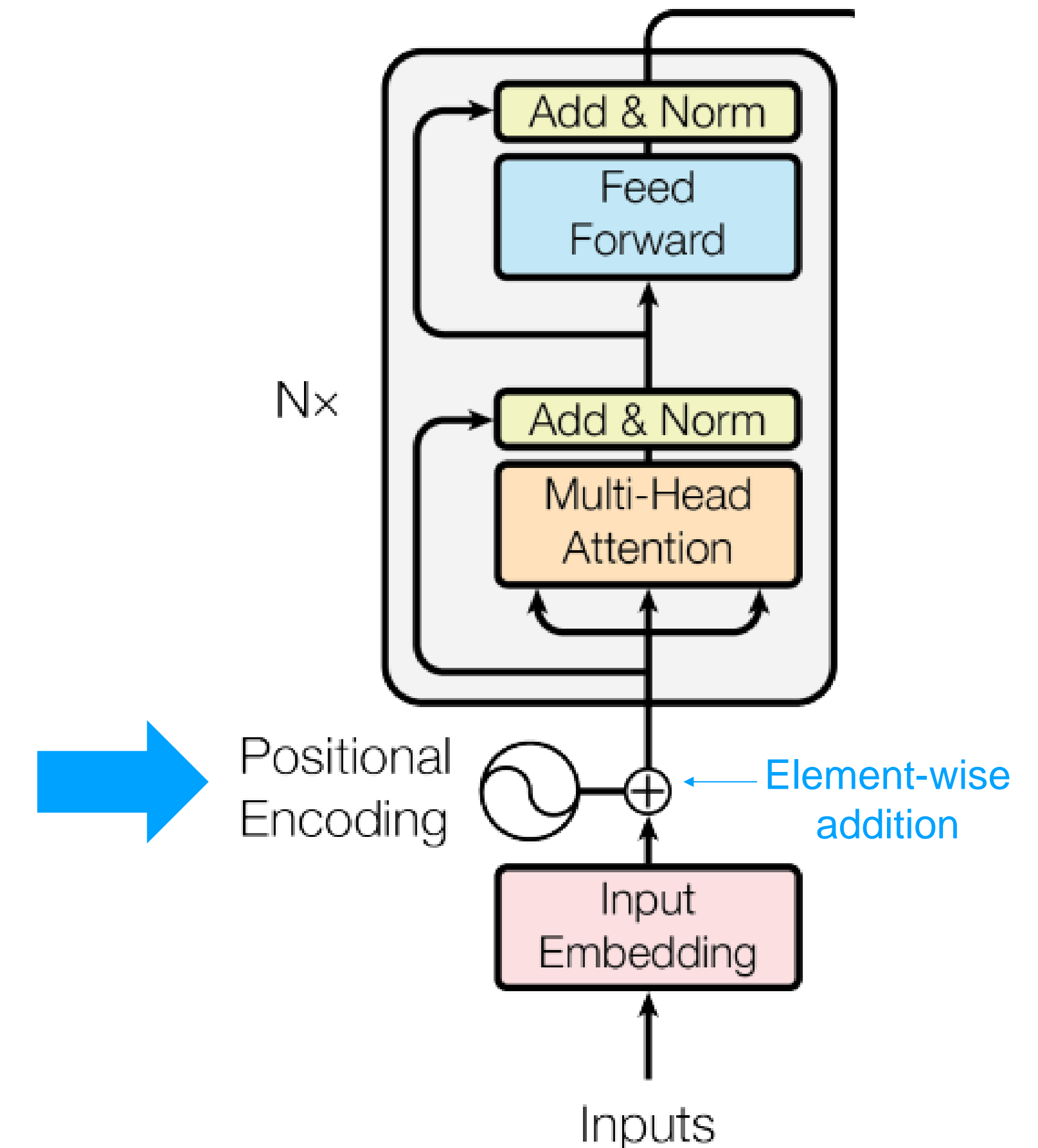
# Transformers: ordered sequences

- Recall: in RNNs, we iteratively feed each input token to the model one at a time
  - Thus, token order information can be preserved, eg through the hidden state
- However: so far (as we've covered it), the MHA block **ignores** token ordering!
  - Ex: {"I", "am", "happy"} looks the same as {"am", "I", "happy"}.
    - Aka a "set" embedding, rather than a "sequence" embedding. Which is valuable in some scenarios, but not here.
  - When token order matters, this is concerning from a modeling standpoint...



# Positional encodings

- The transformers solution: **positional encodings**
- **Idea:** at the beginning of the transformer model, element-wise add a positional embedding (with the same dimensionality as the input  $X$ ) to each token embedding before the first MHA block
- **Requirement:** Positional embedding must encode the "token position"



# Positional encodings

- Positional embedding can either be learned (eg a `torch.nn.Embedding` bank), or hardcoded in a "special way"
- Orig. paper used a sin/cos formula as the positional embedding

$$PE_{(pos, 2i)} = \sin(pos/10000^{2i/d_{\text{model}}})$$

$$PE_{(pos, 2i+1)} = \cos(pos/10000^{2i/d_{\text{model}}})$$

Where:  
pos: sequence index  
i: embedding dim index

```
# Let X be input seq (shape=[seq_len, d])
X_with_pos = torch.zeros(X.shape)
X_with_pos[0, 0] = X[0,0] + sin(0 / (10000 ** (2*0 / d)))
X_with_pos[0, 1] = X[0,1] + cos(0 / (10000 ** (2*1 / d)))
X_with_pos[0, 2] = X[0,2] + sin(0 / (10000 ** (2*2 / d)))
...
X_with_pos[1, 0] = X[1,0] + sin(1 / (10000 ** (2*0 / d)))
X_with_pos[1, 1] = X[1,1] + cos(1 / (10000 ** (2*1 / d)))
X_with_pos[1, 2] = X[1,2] + sin(1 / (10000 ** (2*2 / d)))
...
```

Alternate sin/cos along embed dim

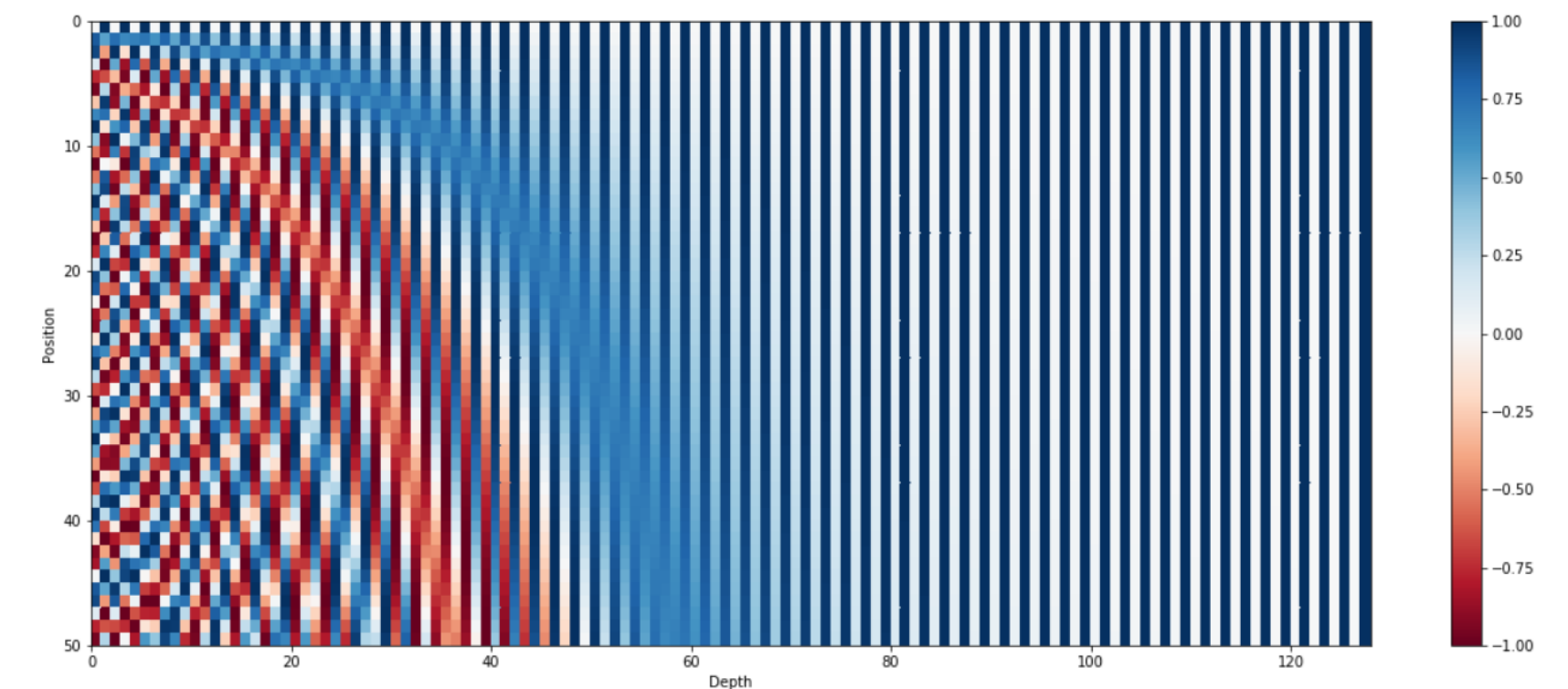
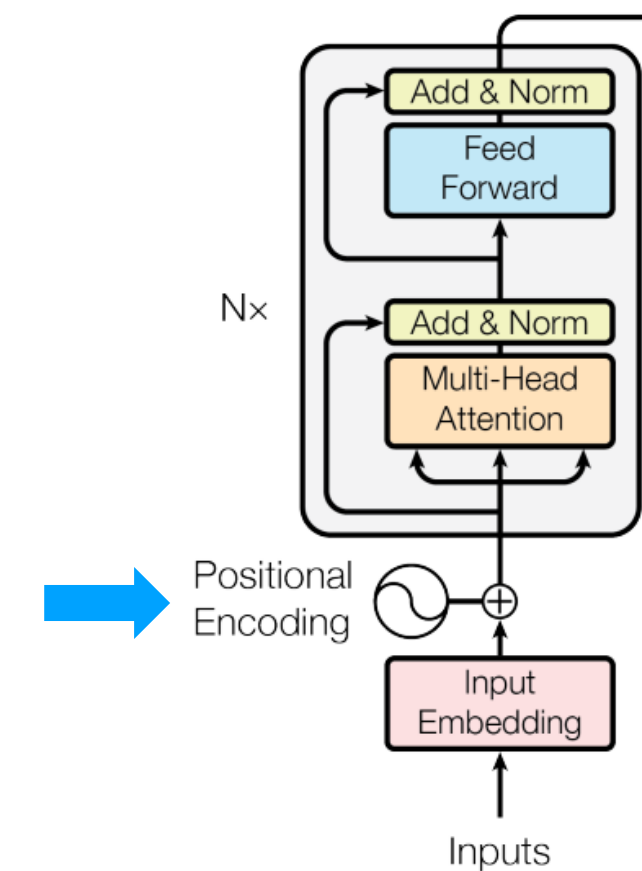


Figure 2 - The 128-dimensional positional encoding for a sentence with the maximum length of 50. Each row represents the embedding vector  $\vec{p}_t$

For more details/intuition on sin/cos, see: [\[link\]](#)

# Transformer: encoder and decoder

- The transformers paper originally discussed two components: an **Encoder**, and a **Decoder**
- **Encoder**: given an input sequence  $X$ , generate "good" token embeddings
- **Decoder**: given an input sequence  $X$ , generate output tokens  $Y$
- First, let's focus on Encoders (it's conceptually simpler)

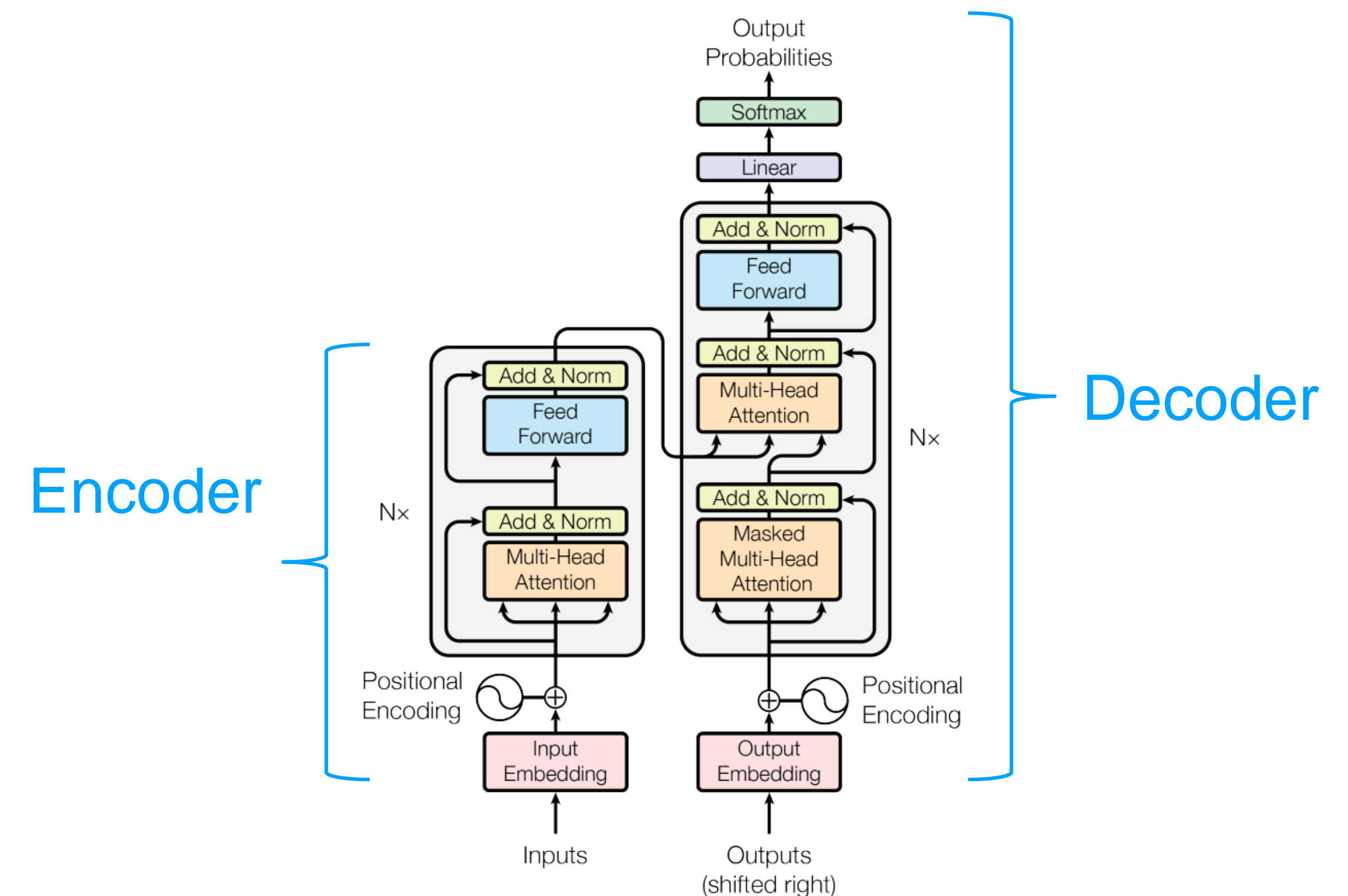
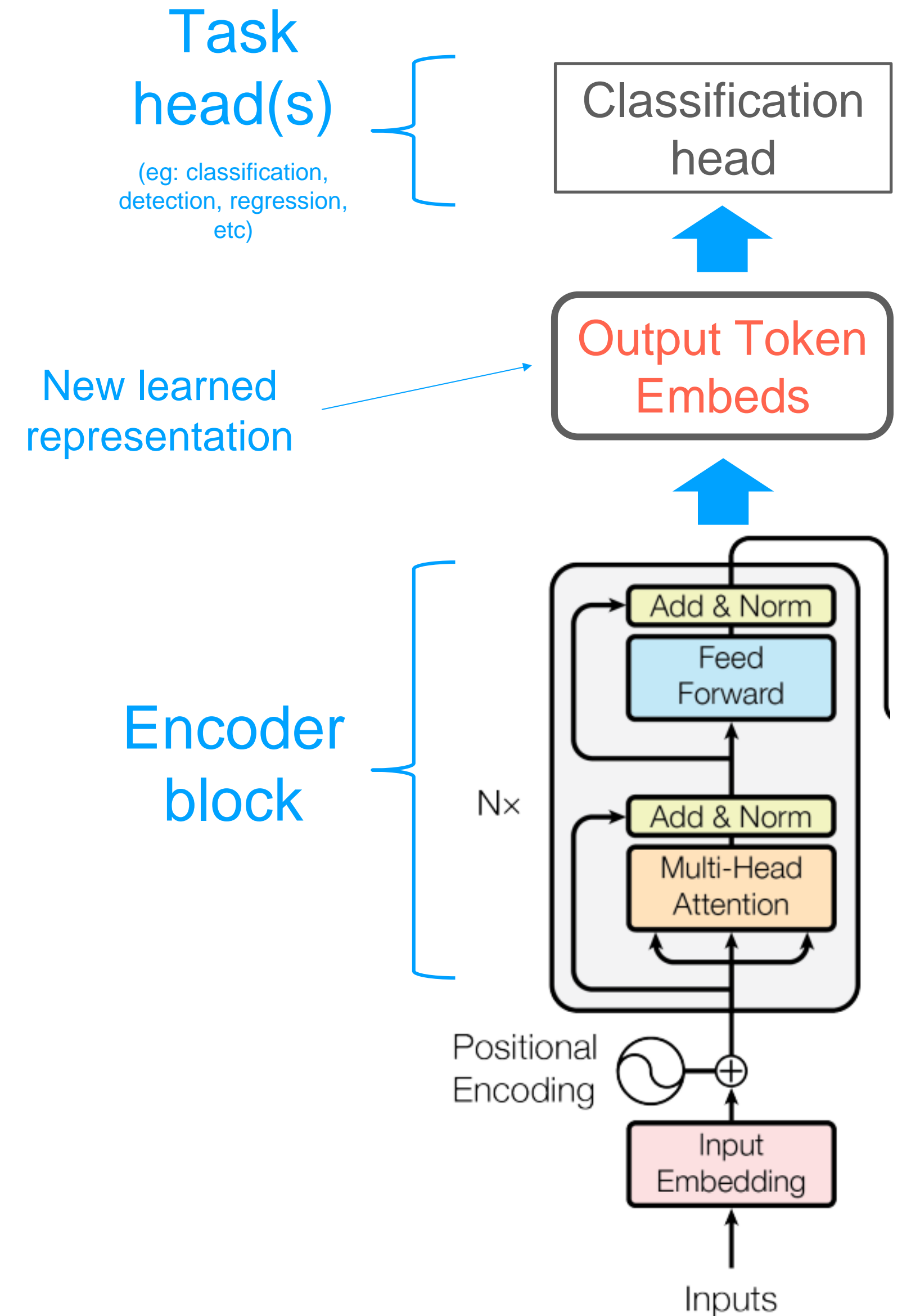


Figure 1: The Transformer - model architecture.

# Encoder

- Input:  $X$  with shape=[bs, seq\_len, dim]
- Output:  $X'$  with shape=[bs, seq\_len, dim]
- **Idea:** learn a representation of the input sequence that is good for downstream tasks (eg classification, text generation, etc)

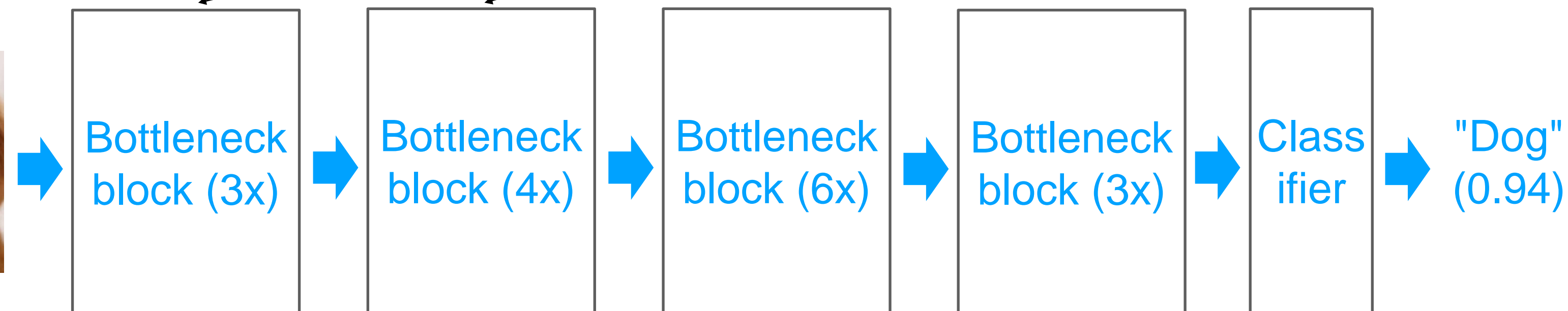
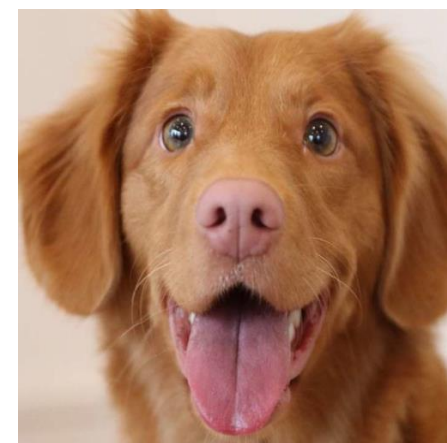




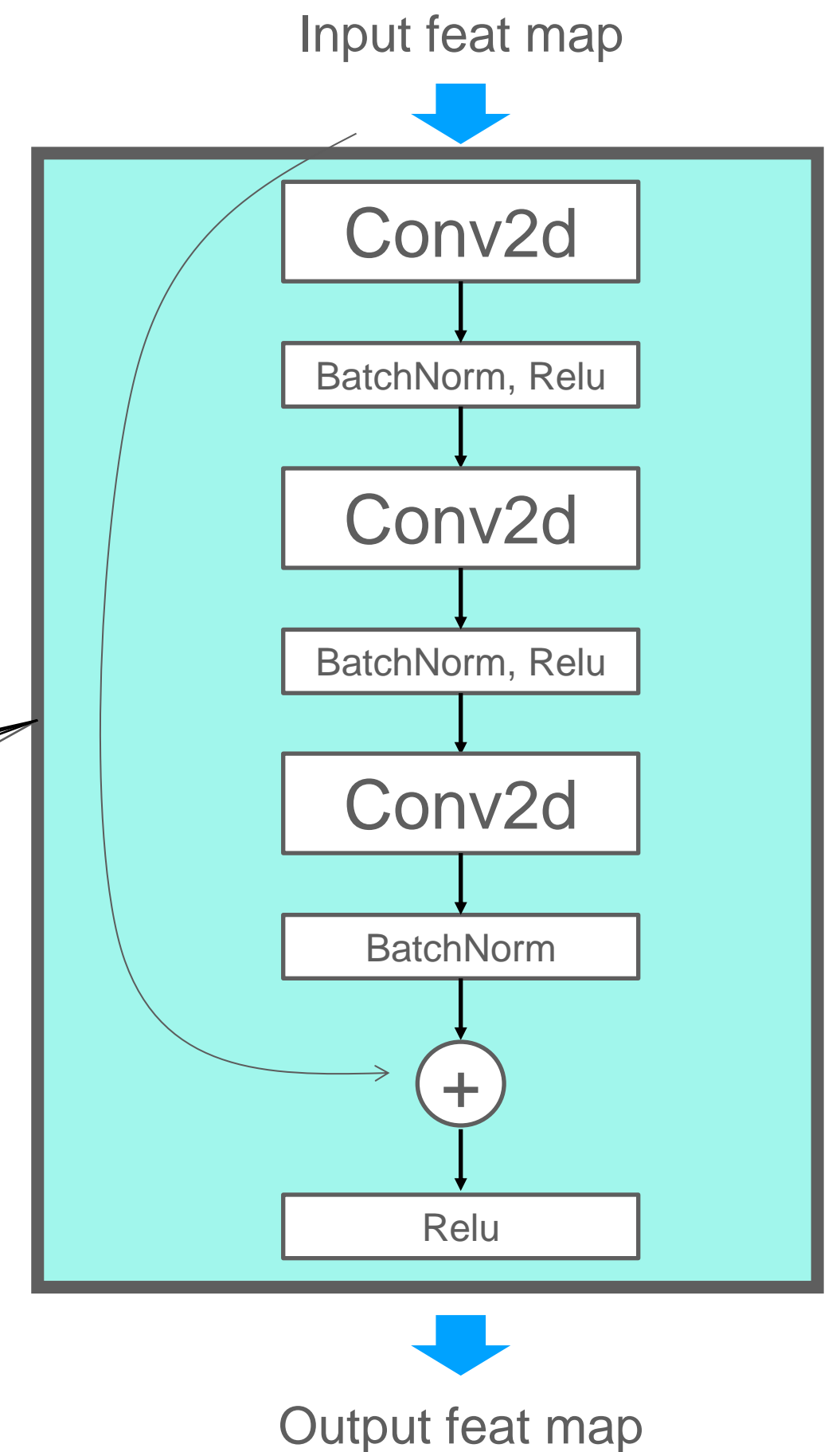
# Aside: NN "blocks"

- It's common to define a DNN in terms of "blocks", rather than individual layers
- Ex: a "ResNet Bottleneck Block" [\[link\]](#) consists of:

**ResNet50** (Image classification arch [\[link\]](#)) is then built by repeating the "Bottleneck block" a bunch of times



"Bottleneck" block

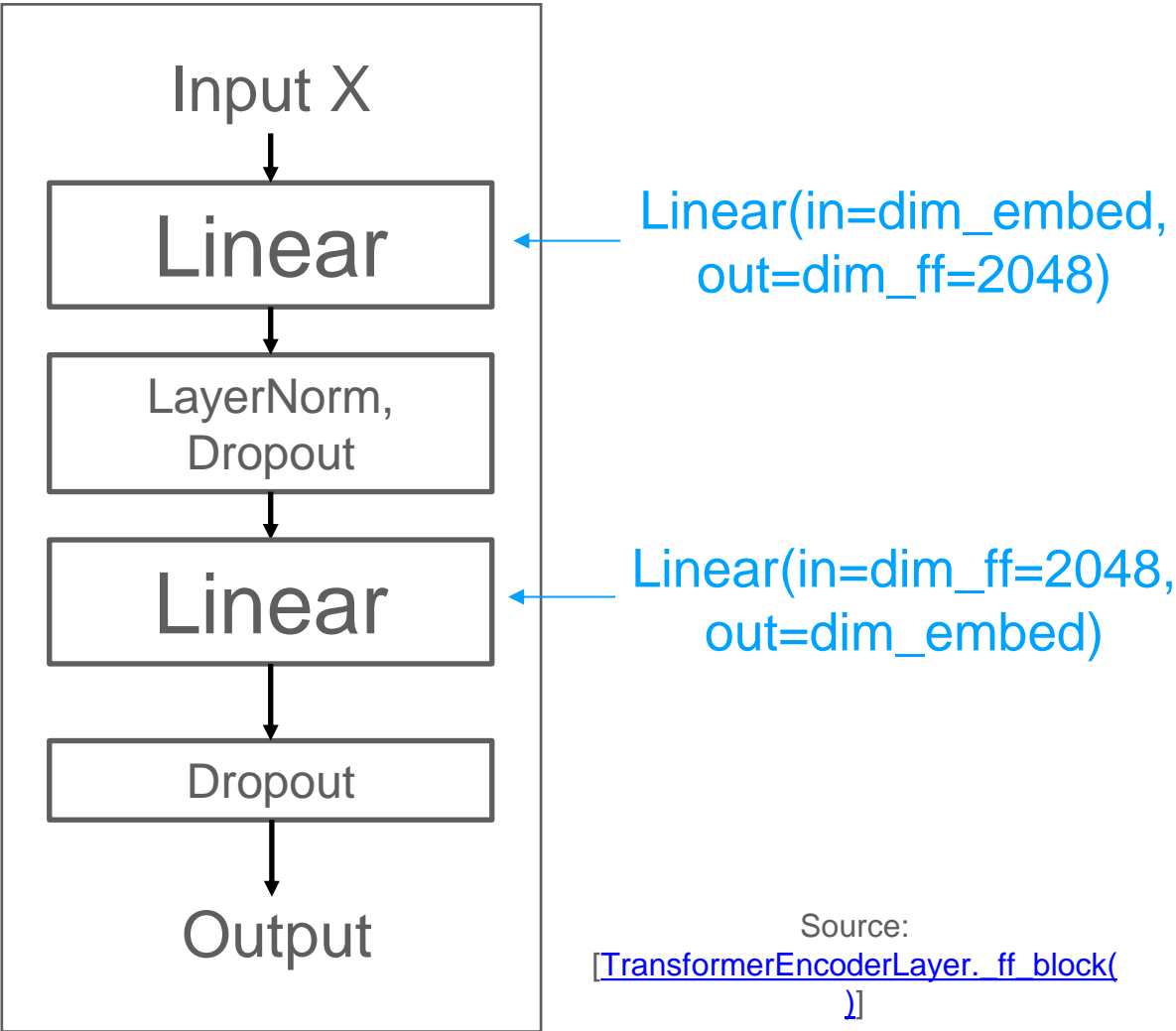
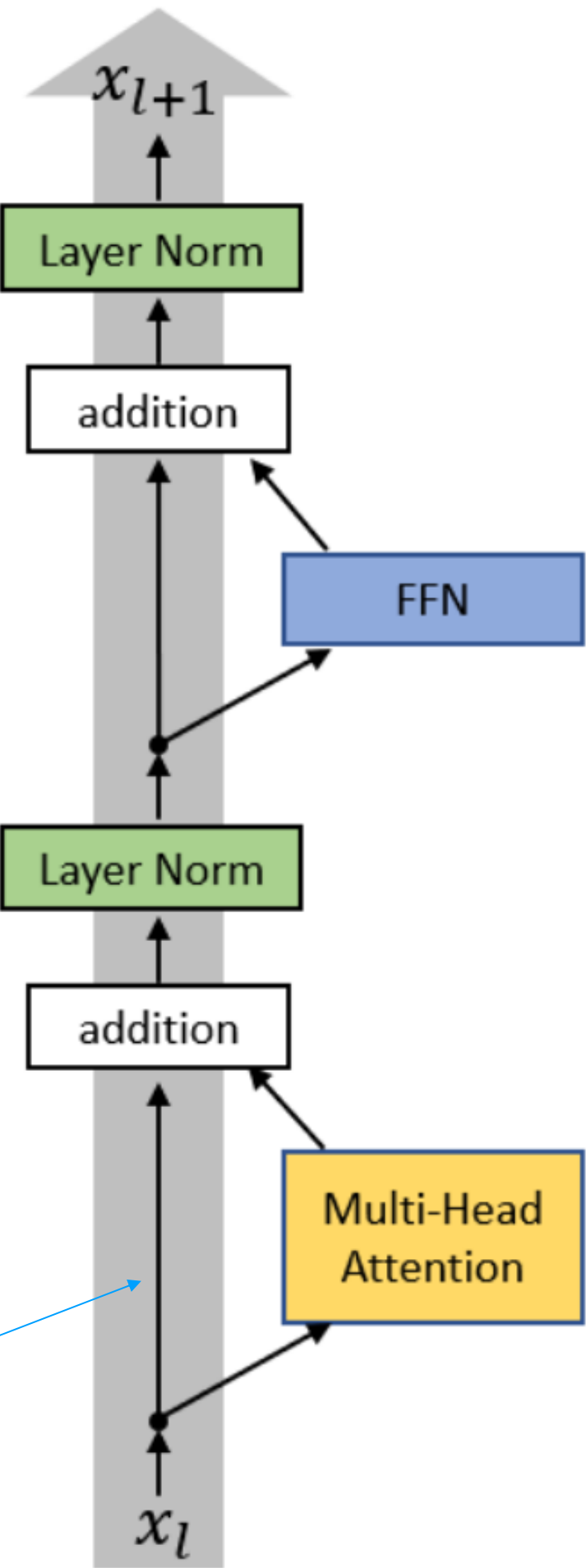


\* Some things missing from this resnet50 picture: initial "stem", downsampling (spatial resolution) layers, other minor details

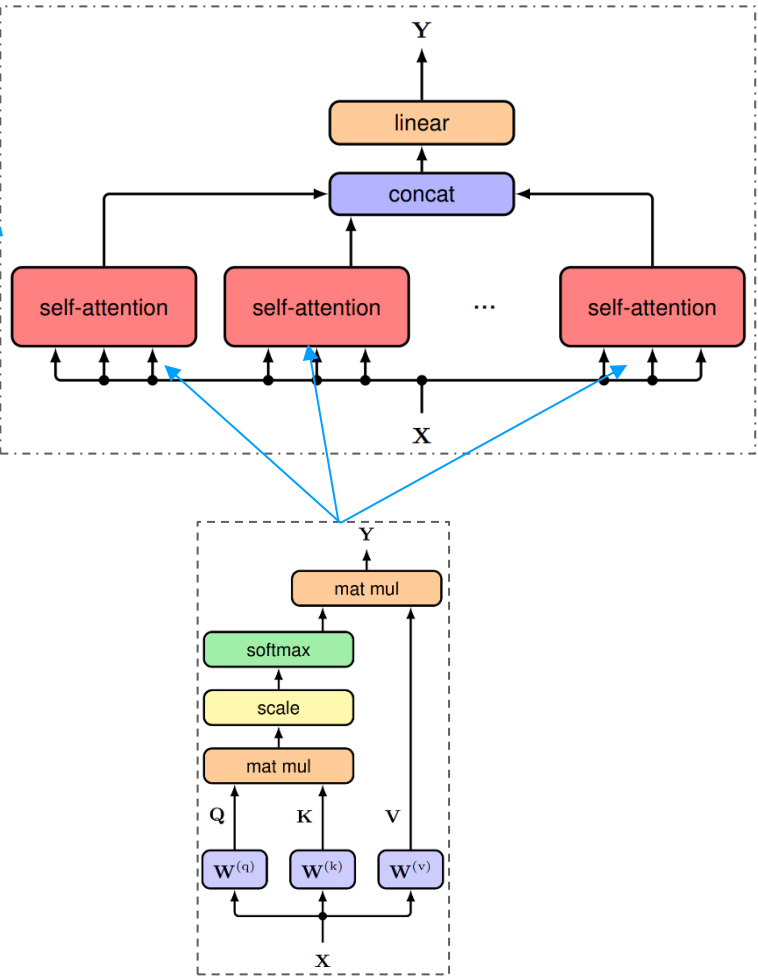
If you're curious here is pytorch's (torchvision) implementation of ResNet50 (in all of its gory detail): [\[link\]](#)  
Tip: you should be able to read this code and understand 90% of what's going on! The implementation is complicated because it's super generic+flexible, but with some study you can see what's going on :)

# Encoder block

Source:  
[\[TransformerEncoderLayer.forward\(\)\]](#)



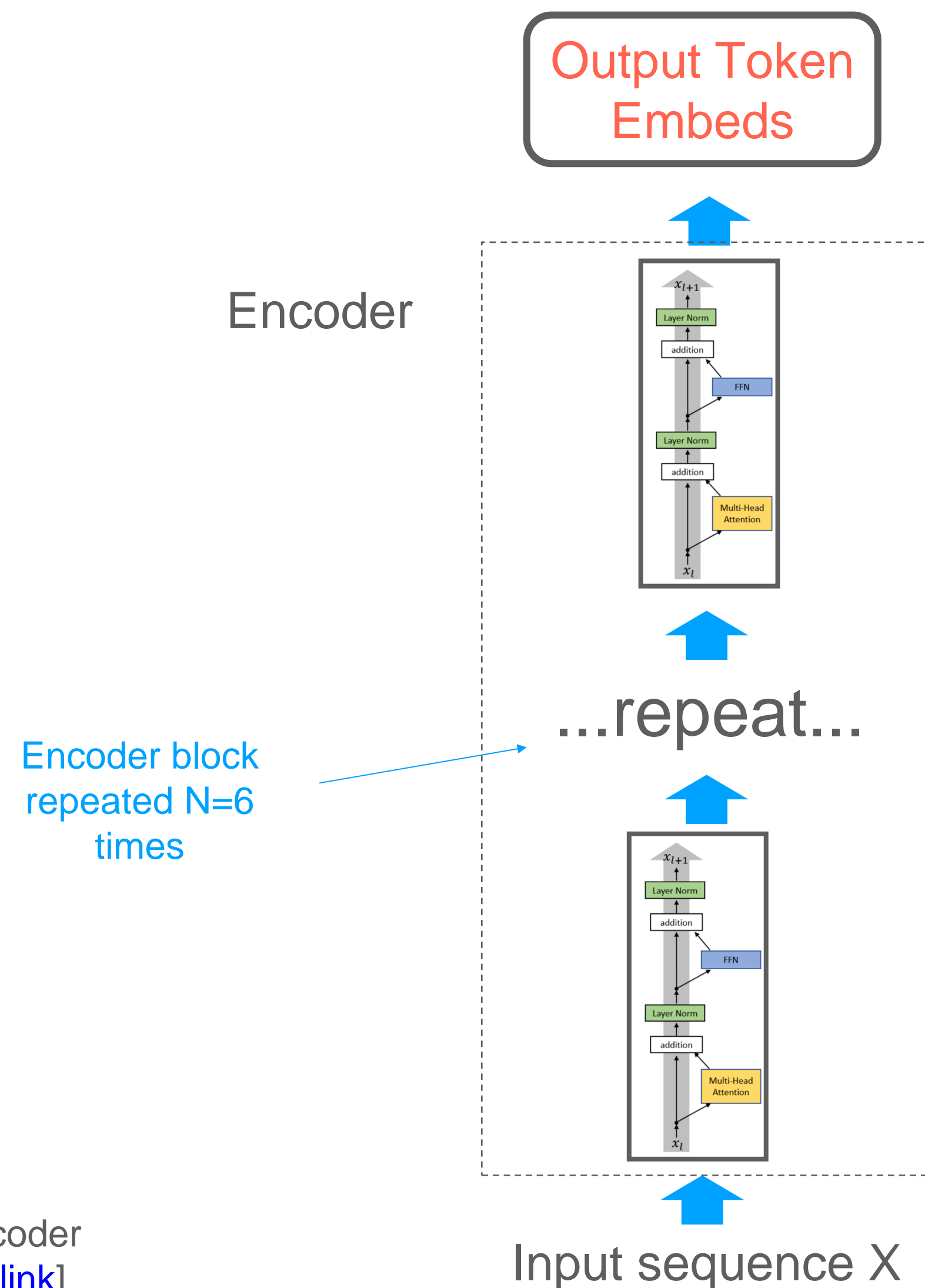
Source:  
[\[TransformerEncoderLayer.ff\\_block\(\)\]](#)



Residual/skip  
connection

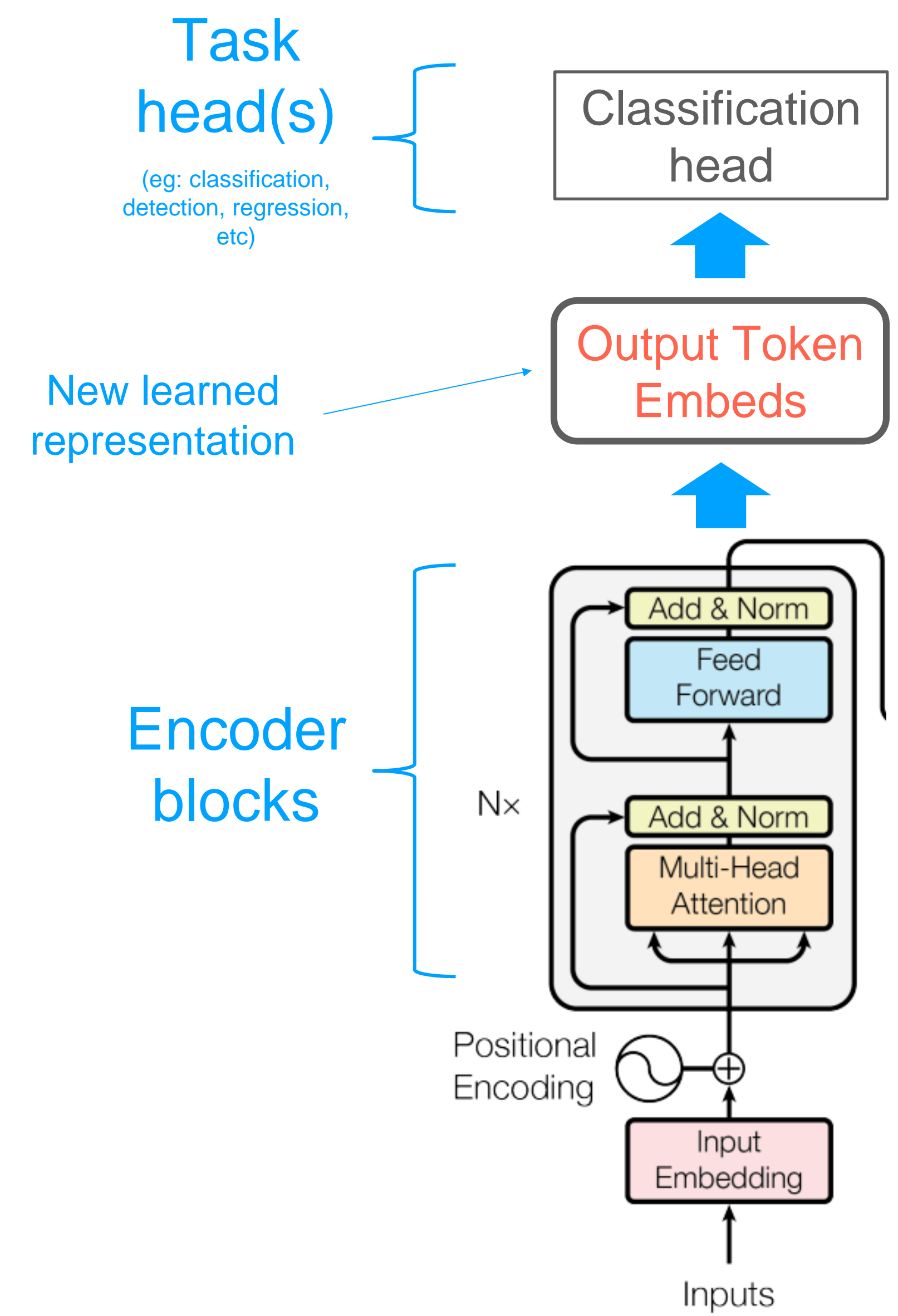
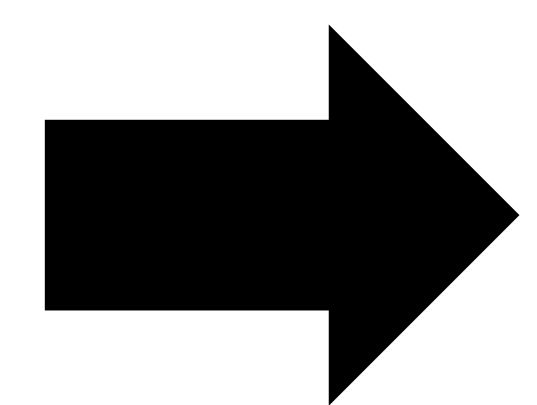
(a)

# Encoder Arch



TransformerEncoder  
pytorch code: [\[link\]](#)

(drawn another way,  
eg from the original  
Transformers paper)



# Detail: Pre-norm vs Post-norm

- Original transformers paper did "Post-LayerNorm" (Post-LN)
- But, smart people found out that "Pre-LayerNorm" (Pre-LN) works better
  - Better training stability, can use larger learning rate, etc
- For more details, see: [[link](#)]

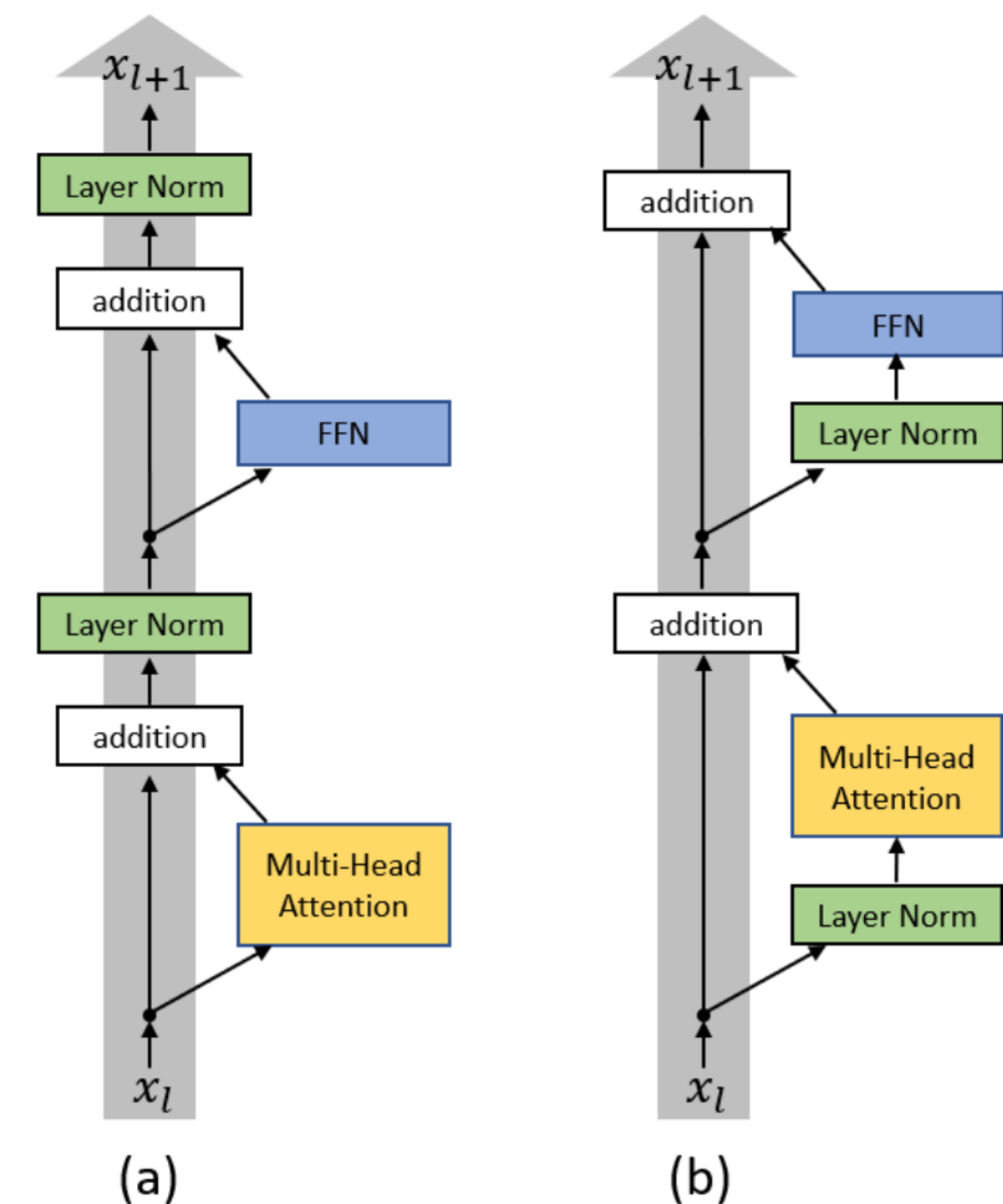
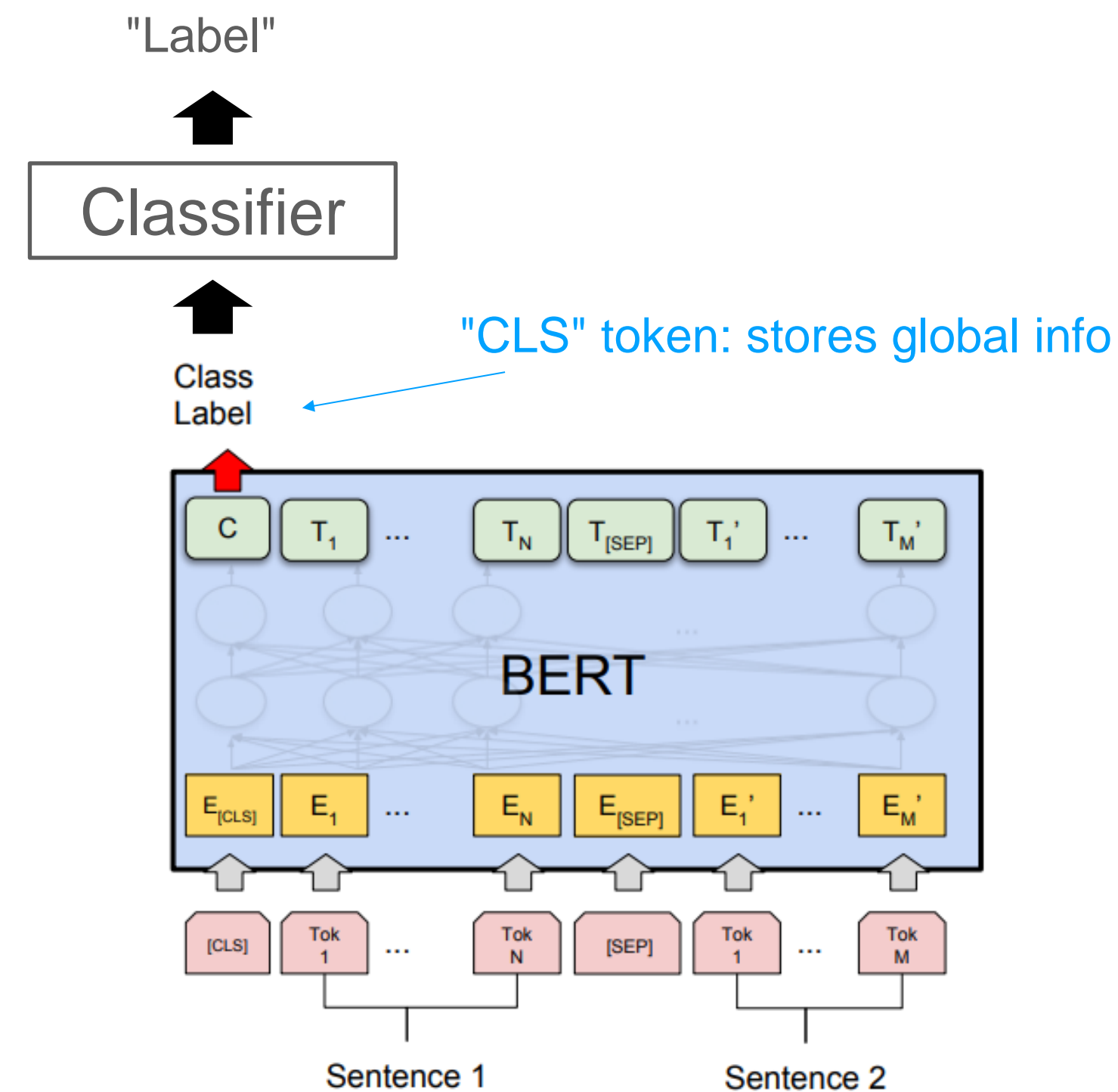


Figure 1. (a) Post-LN Transformer layer; (b) Pre-LN Transformer layer.

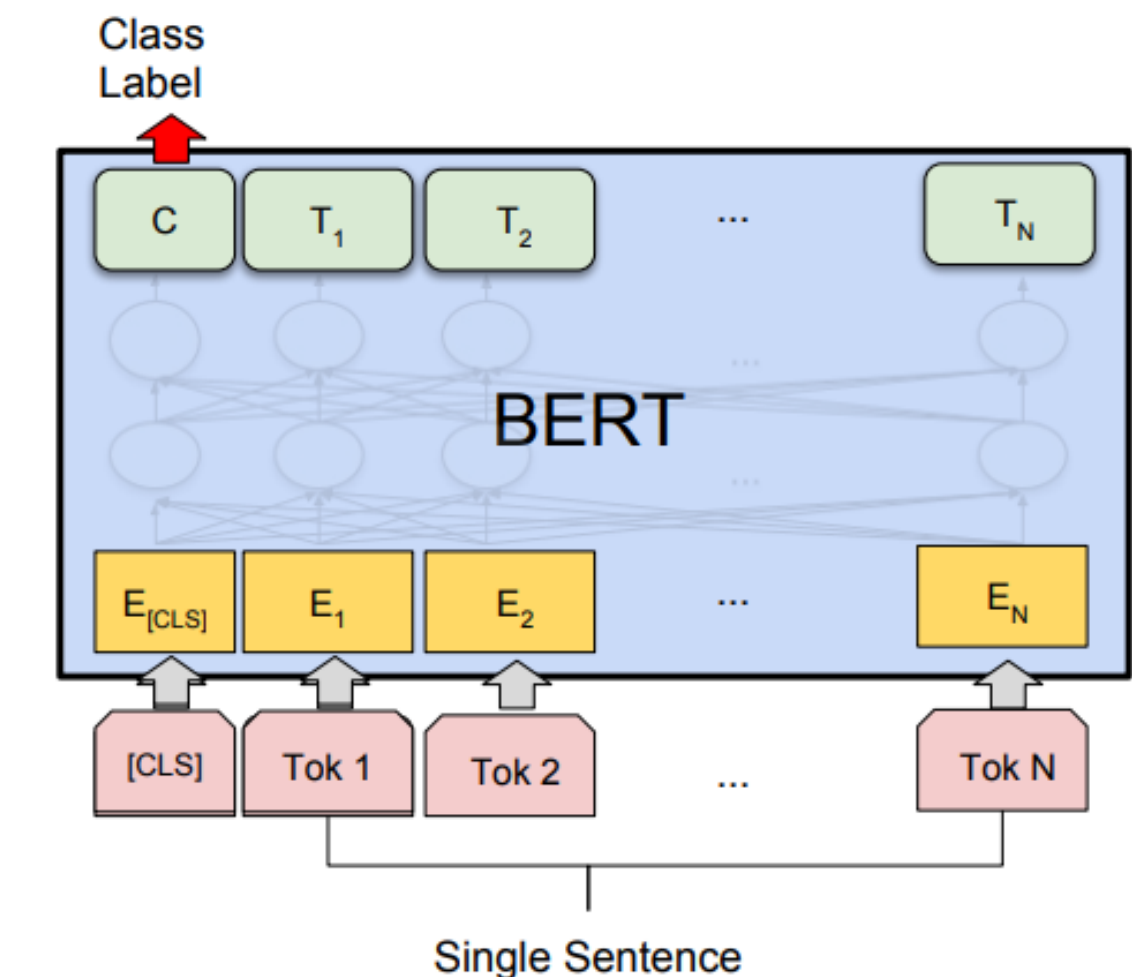
(Left) Original version. (Right)  
"Improved" version

# Application: text classification (Bert)

- BERT ("Bidirectional Encoder Representations from Transformers"): [\[link\]](#)
- Perform text sentence classification using a Transformer Encoder + pretraining/finetuning
- Key idea: prepend a "CLS" token to the start of every sequence. Then, train a classifier on top of this CLS token embedding
  - Intuition: CLS token stores the "global" info about the sentence



(a) Sentence Pair Classification Tasks:  
MNLI, QQP, QNLI, STS-B, MRPC,  
RTE, SWAG

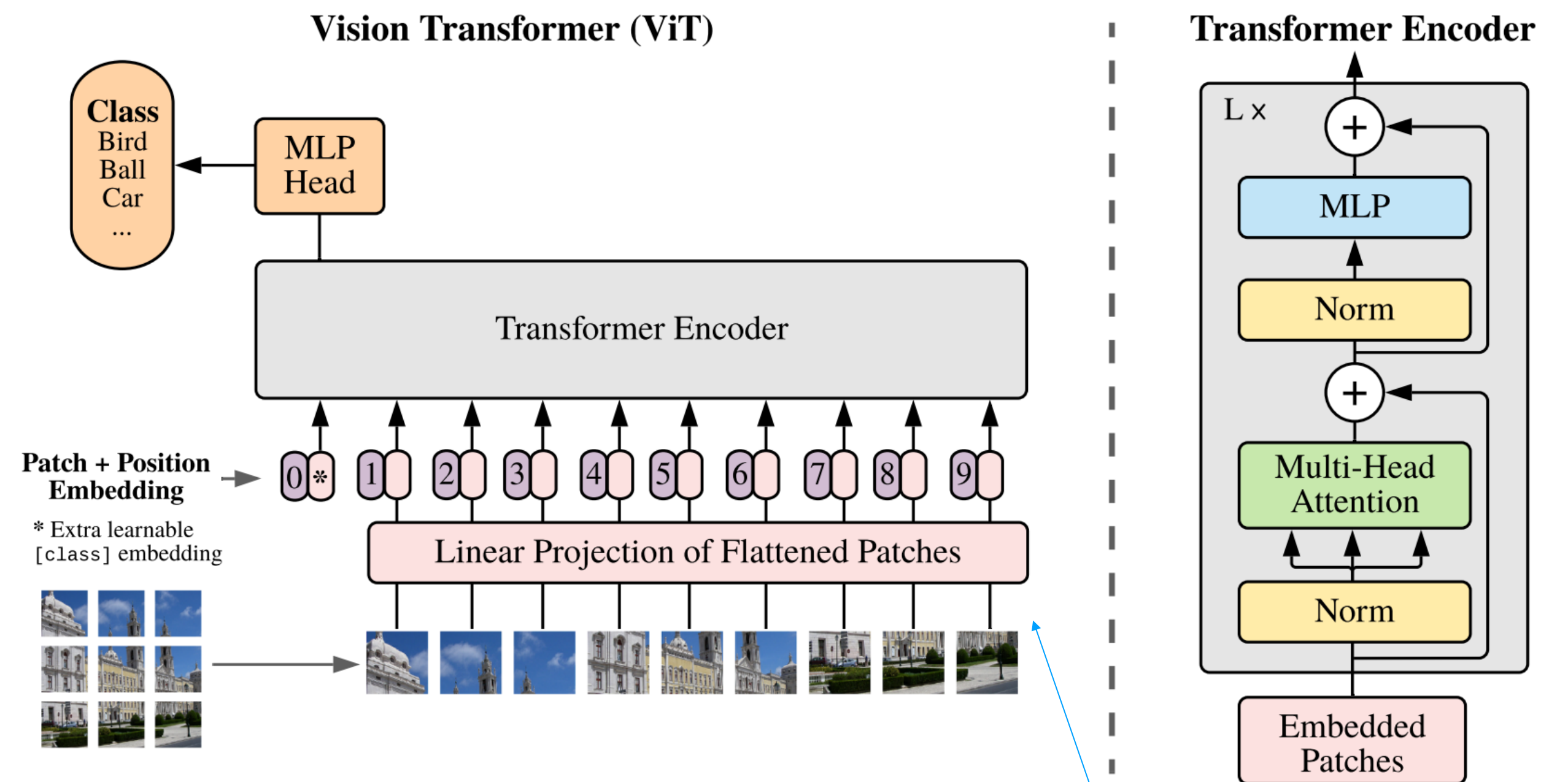


(b) Single Sentence Classification Tasks:  
SST-2, CoLA



# Application: image classification (ViT)

- "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale" [[link](#)]
- Image classification with Transformer Encoder
- Idea: represent an image as a sequence of image patches!



Fun fact: this "linear projection of flattened patches" is basically a Conv2d, which people laughed about for awhile...