Data 188: Introduction to Deep Learning

Transformers

Speaker: Eric Kim
Lecture 16 (Week 11)
2026-03-31, Spring 2026. UC Berkeley.

Announcements

- Midterm grades out!
  - Regrade requests via Gradescope
    - Last call: Wednesday April 1st, 11:59 PM PST

- HW3 released: "Intro to Pytorch"
  - Tip: the remaining homework assignments in this course will not use Needle (ie your previous HW0 -> HW2).

Today's lecture

Sequence modeling

Transformers

Multi-head self attention (MHA): deep dive

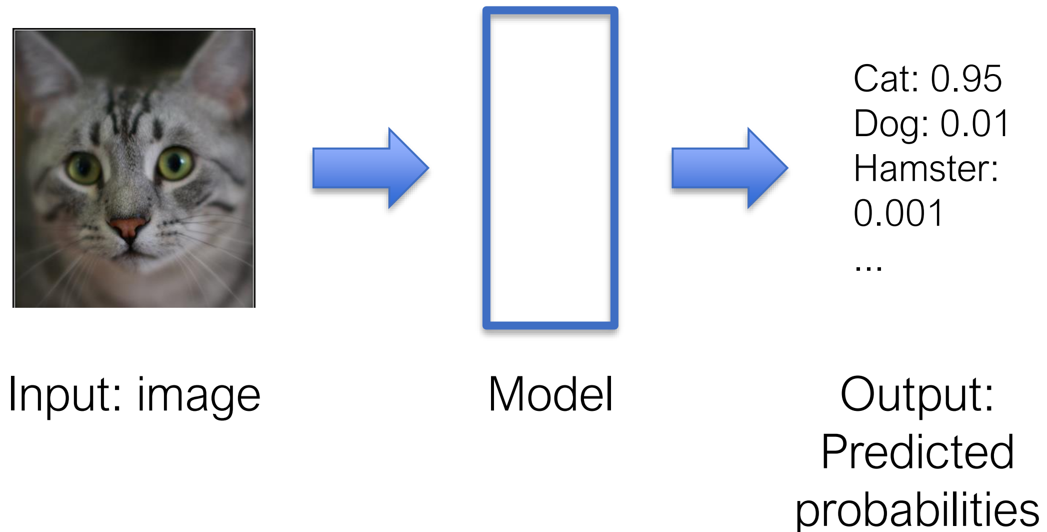
Encoders

- Text classification, image classification

Sequence modeling

So far: we've mainly focused on "point predictions"

- Ex: image classification. Given an image, what category is it?

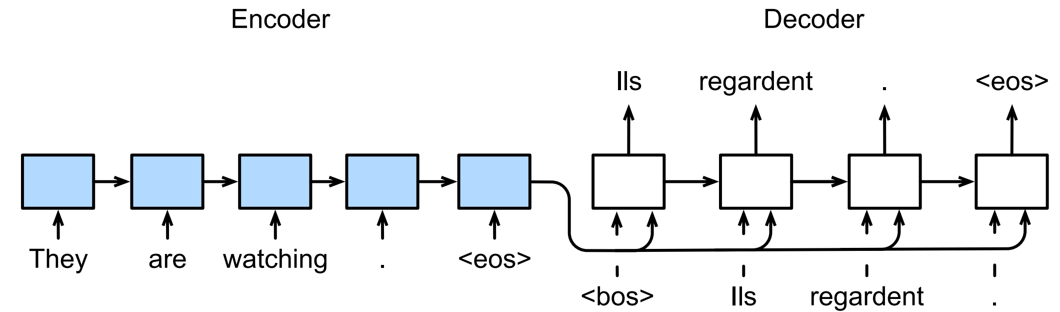


Sequence modeling

Many problems are instead naturally described via **sequences**

Example: given an input text sequence:

- **Generation**: what should I respond with? (ie "chat bots")
- **Translation**: translate from English to French.
- **Classification**: does this sentence have positive sentiment?



Machine translation: English to French.

https://d2l.ai/chapter_recurrent-modern/seq2seq.html

Tokens and tokenizers

In sequence modeling, each sequence item is called a **token**.

A **tokenizer** takes input raw text, and outputs the tokens that represent the raw text.

Ex: a simple text tokenizer that maps each word in the English language to a unique int ID.

```
"Garbage in, garbage out!"
-> ["GARBAGE", "IN", ",", "GARBAGE", "OUT", "!"]
-> [42, 205, 2, 42, 9028, 3]
```

Raw text can't be passed to models, but these int ids are easier for models to ingest.

Rather than directly pass the token int ids to models (ie as an int feature), we instead use **token embeddings** (next slide)

Tokenizers are typically static during model training, treated as data preprocessing. They are not "backpropable".

There are more sophisticated tokenizers used in practice. BytePairEncoding, for instance, is used by OpenAI in their ChatGPT-4 models:

```
# Load a pre-trained tiktoken tokenizer
# cl100k_base used by gpt-4, gpt-4o, and gpt-3.5-turbo
tiktoken_tokenizer = tiktoken.get_encoding("cl100k_base")

# Sample text for tokenization
text = "Garbage in, garbage out!"

print(f"Original Text: {text}")

# Encode the text to get token IDs
token_ids = tiktoken_tokenizer.encode(text)
print(f"Tiktoken Token IDs (integers): {token_ids}")

# Decode the token IDs back to bytes, then to string
tokens_bytes = [tiktoken_tokenizer.decode_single_token_bytes(token_id) for token_id in token_ids]
tokens_str = [b.decode('utf-8', errors='replace') for b in tokens_bytes]
print(f"Tiktoken Tokens (string): {tokens_str}")

# Decode the entire list of token IDs back to the original text
decoded_text = tiktoken_tokenizer.decode(token_ids)
print(f"Tiktoken Decoded Text: {decoded_text}")
```

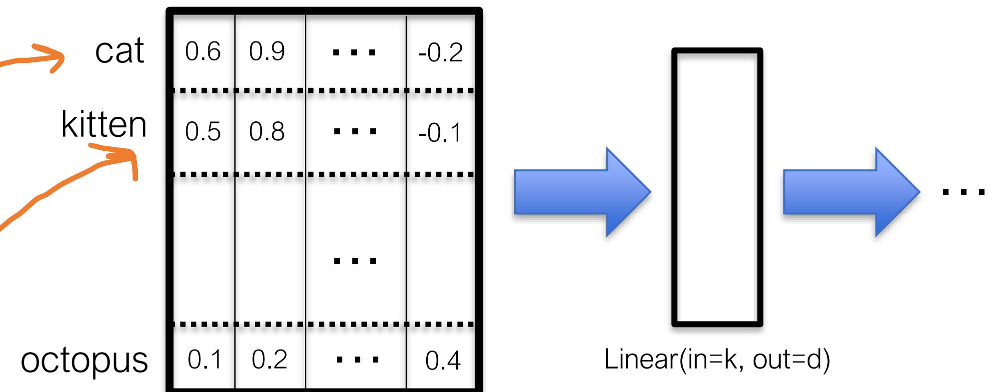
```
Original Text: Garbage in, garbage out!
Tiktoken Token IDs (integers): [45030, 21305, 304, 11, 26964, 704, 0]
Tiktoken Tokens (string): ['Gar', 'bage', ' in', ',', ' garbage', ' out', '!']
Tiktoken Decoded Text: Garbage in, garbage out!
```

Token embeddings

Each token in the token vocabulary is assigned a **token embedding** from an **embedding table**.

This embedding table is a learnable matrix! We randomly initialize the token embedding table, and the model will learn "good" token embeddings that are useful for downstream tasks.

```
"Kitten is a cat"
-> ["KITTEN", "IS", "A", "CAT"]
-> [1, 44, 58, 0]
```



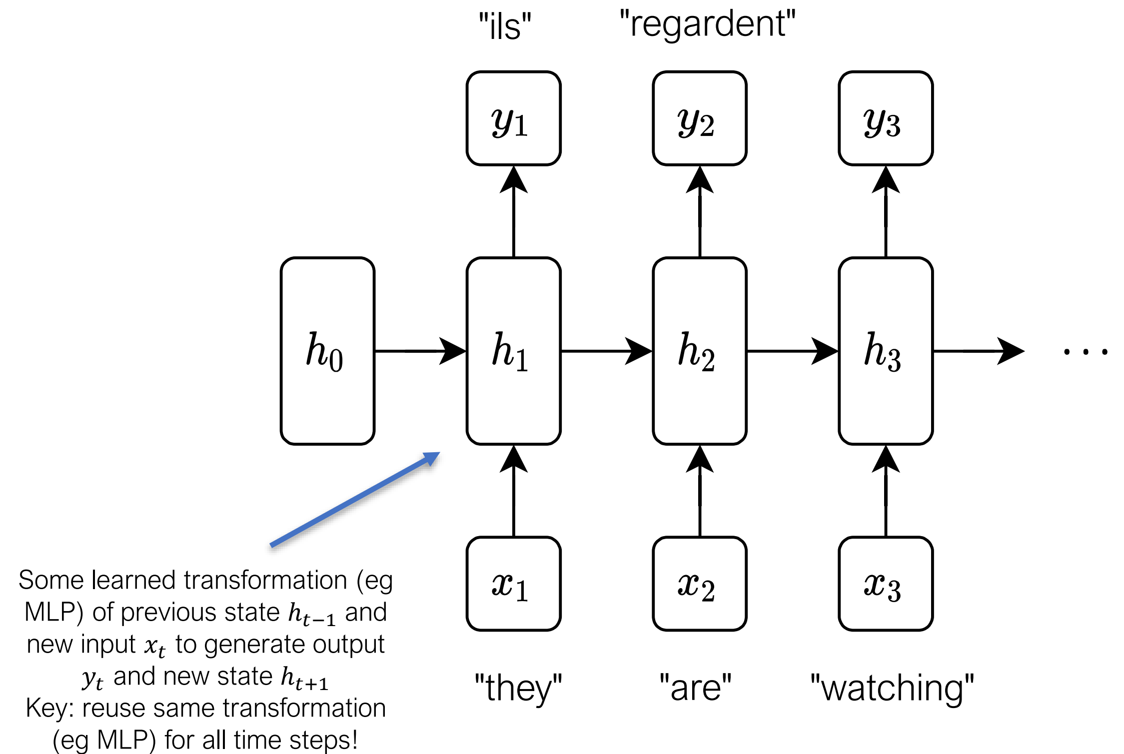
aka a **learnable** parameter matrix with shape=[n, k]
Ex: torch.nn.Embedding

The RNN "latent state" approach

Historically, a popular method for working with sequence data was to use recurrent neural networks (RNN).

Idea: maintain "latent state" h_t that summarizes *all* information up until time step t.

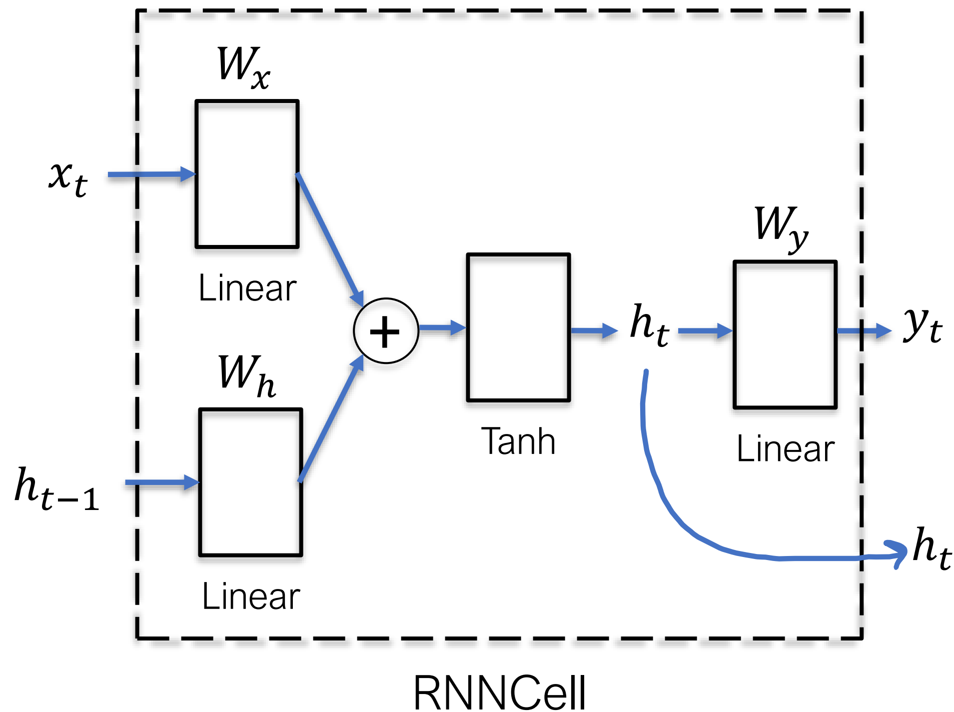
Useful for sequence data, time-series data.



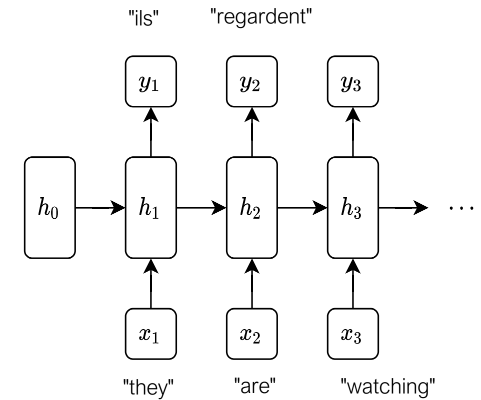
x_t: input at time step t.
h_t: hidden ("latent") state that encodes all state up to (and including) time step t.
y_t: output at time step t.

RNN: simple example

Define our RNN "cell" to be:



x_t: input at time step t.
h_t: hidden ("latent") state that encodes all state up to (and including) time step t.
y_t: output at time step t.




```
# x.shape=[B, seq_len, d_in]
x = torch.rand(size=(2, 8, 16))
# h.shape=[B, d_h]
h = torch.zeros(size=(2, 32))
# y.shape=[B, d_out]
y = torch.zeros(size=(2, 8, 10))
for t in range(x.shape[1]):
    # x_t: current input. shape=[B, d_in]
    x_t = x[:, t, :]
    # h_t_prev: previous latent state.
    # shape=[seq_len, d_h]
    h_t_prev = h
    ## Predict output y_t by incorporating both
    ## previous knowledge (h_t_prev) and new input
    ## signal (x_t)
    # y_t.shape=[B, d_out]
    y_t, h_t = rnn_cell(x_t, h_t_prev)
    y[:, t, :] = y_t
    # Update latent state
    h = h_t
```

For-loop "unrolls" the RNN

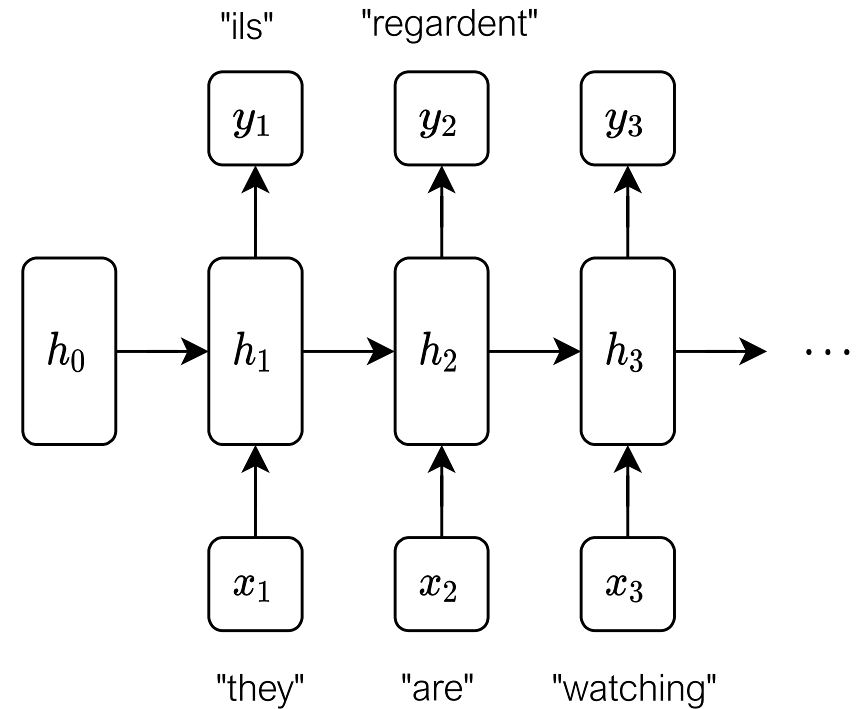
RNN: Pros and cons

Pros: Potentially “infinite” history, compact representation

Cons: Long “compute path” between history and current time ⟹ vanishing / exploding gradients, hard to learn.

Computationally expensive: computation scales directly with sequence length. ("unrolling")

"Memory/forgetting": how to ensure that model hidden state "remembers" information from distant time steps?



x_t: input at time step t.
h_t: hidden ("latent") state that encodes all state up to (and including) time step t.
y_t: output at time step t.

"Attention Is All You Need" (2017)

Paper that introduced the "Transformers" model architecture [link]

Mitigates several of the issues with RNN's. Main modeling contribution is called "Multi-head self attention".

Highly influential and impactful

- Ex: powers Chat-GPT!
- As of 2026, Transformers are 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.

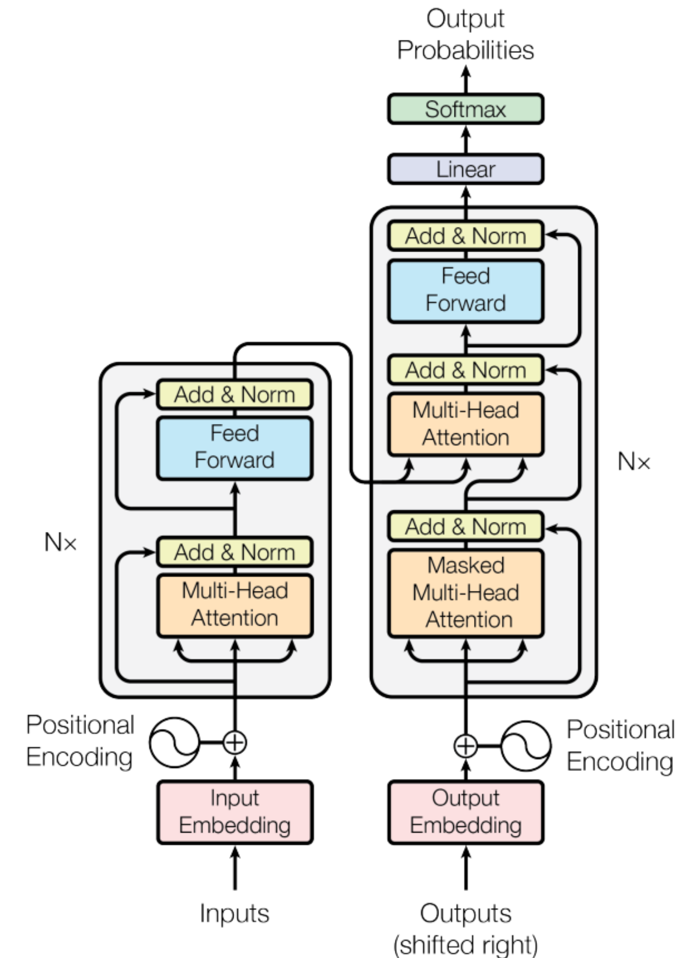


Figure 1: The Transformer - model architecture.

Transformer: high level

A transformer takes an **input sequence** and via a series of transformer blocks (utilizing multi-head self attention, "MHA"), learns a strong **latent representation** of the sequence that is useful for downstream tasks (eg text classification, text generation, etc.).

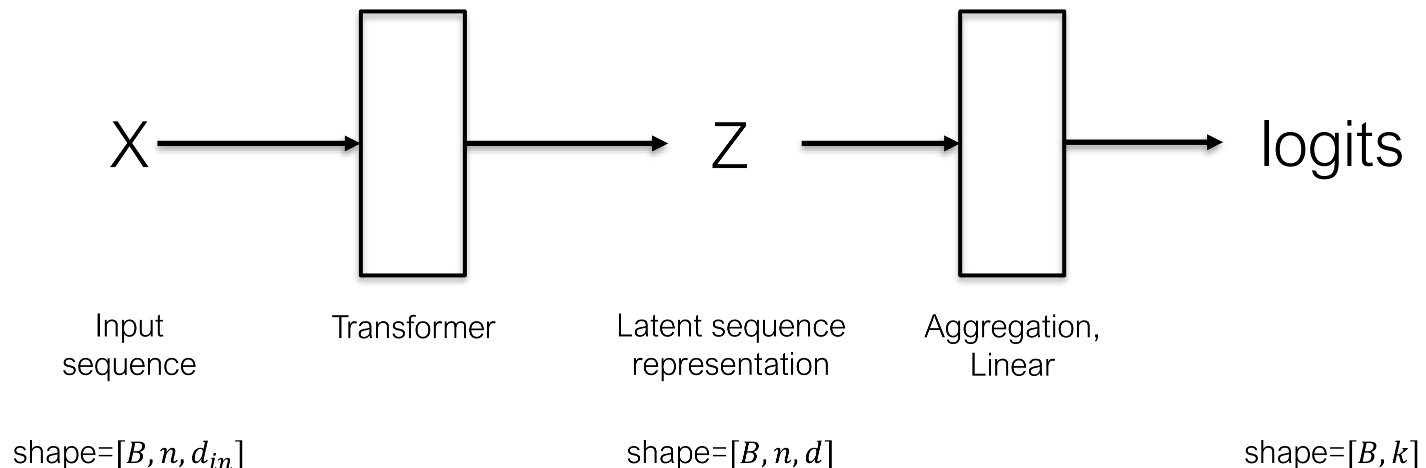
Input: Sequence. Shape=$[batchsize, n, d_{in}]$.

n is sequence length.

Latent representation: shape=$[batchsize, n, d]$

d is latent embedding size.

A transformer used for classification.
Ex: text sentiment analysis. Does this input sentence have positive/negative sentiment?



k is number of classes

Note: for implementation simplicity, we typically require that $d_{in} = d$.
If you want to increase (or decrease) your token embedding size, you can use a Linear layer before the Transformer input to your desired embedding size (aka a linear projection layer)

Multihead self attention (MHA)

MHA layer does two things

- (1) **Self-attention.** Learn which parts of the input are important for downstream tasks
- (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

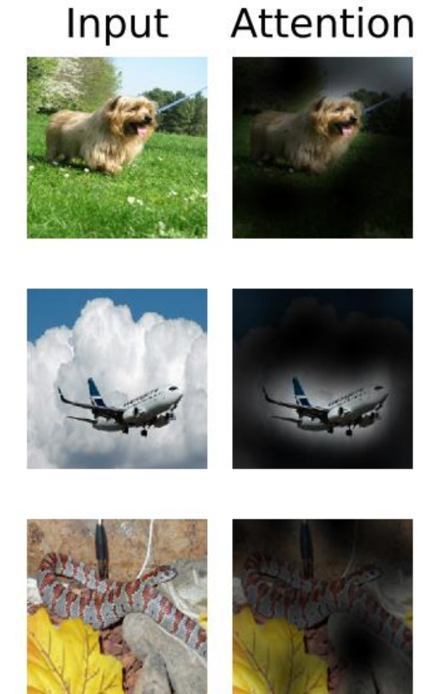


Figure 6: Representative examples of attention from the output token to the input space. See Appendix D.7 for details.

https://arxiv.org/abs/2010.11929

Attention and explainable AI

Motivation: given an input X, when producing a prediction/output Y, we want the model to tell us "why" it returned Y

- "Explainable AI"

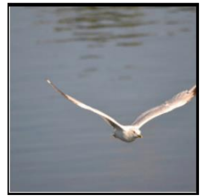
One popular approach: "Attention"

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"

Attention, explainable AI visualization

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



A

bird

flying

over

a

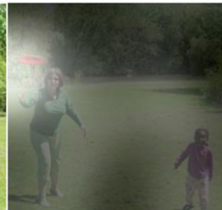
body

of

water

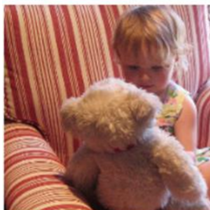
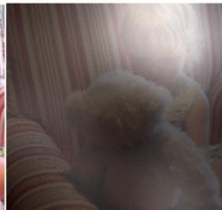
.

Problem setting: Image captioning.
Input: image.
Output: text caption describing the image.

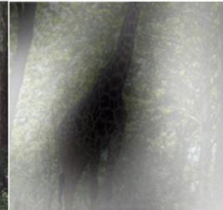

A woman is throwing a frisbee in a park.

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.

Xu et al, “Show, Attend, and Tell: Neural Image Caption Generation with Visual Attention”. ICML 2015.

MHA (v0)

Exercise: Let's build up, step-by-step, to the "full" MHA

Note: to simplify things, let's assume we're working with a single input sequence X, batchsize=1, sequence length is `n`, and each input token embedding has size `d` => X.shape=[n, 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=[n, d], what should the attention scores shape be?

- **Answer**: [n, n]! 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 (ie as probabilities), 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()

https://theaisummer.com/transformer/

MHA (v0): attention scores

Question: what is the simplest way you can think of to calculate self-attention? (X shape=[n, d])

- **Answer**: one way is to multiply all pair-wise dot products between rows of X:
  - $attn_scores = softmax(X@X^T, \dim = 1)$

Pro: simple

Con: What if X isn't good at calculating "good" attention scores?

- Solution: Let's use a learned transformations of X! Called: Query, Key, and Value.

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()

https://theaisummer.com/transformer/

MHA (v0): Query, Key, Value

Let: X be [n, d]

Step 1: Calculate Query, Key, and Value:

shape=[n, d]

$$\mathbf{Q} = \mathbf{X}\mathbf{W}^{(q)}$$
$$\mathbf{K} = \mathbf{X}\mathbf{W}^{(k)}$$
$$\mathbf{V} = \mathbf{X}\mathbf{W}^{(v)}$$

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

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

$$attn_scores = softmax(\frac{QK^T}{\sqrt{d}})$$

shape=[n, n]

softmax is done over rows

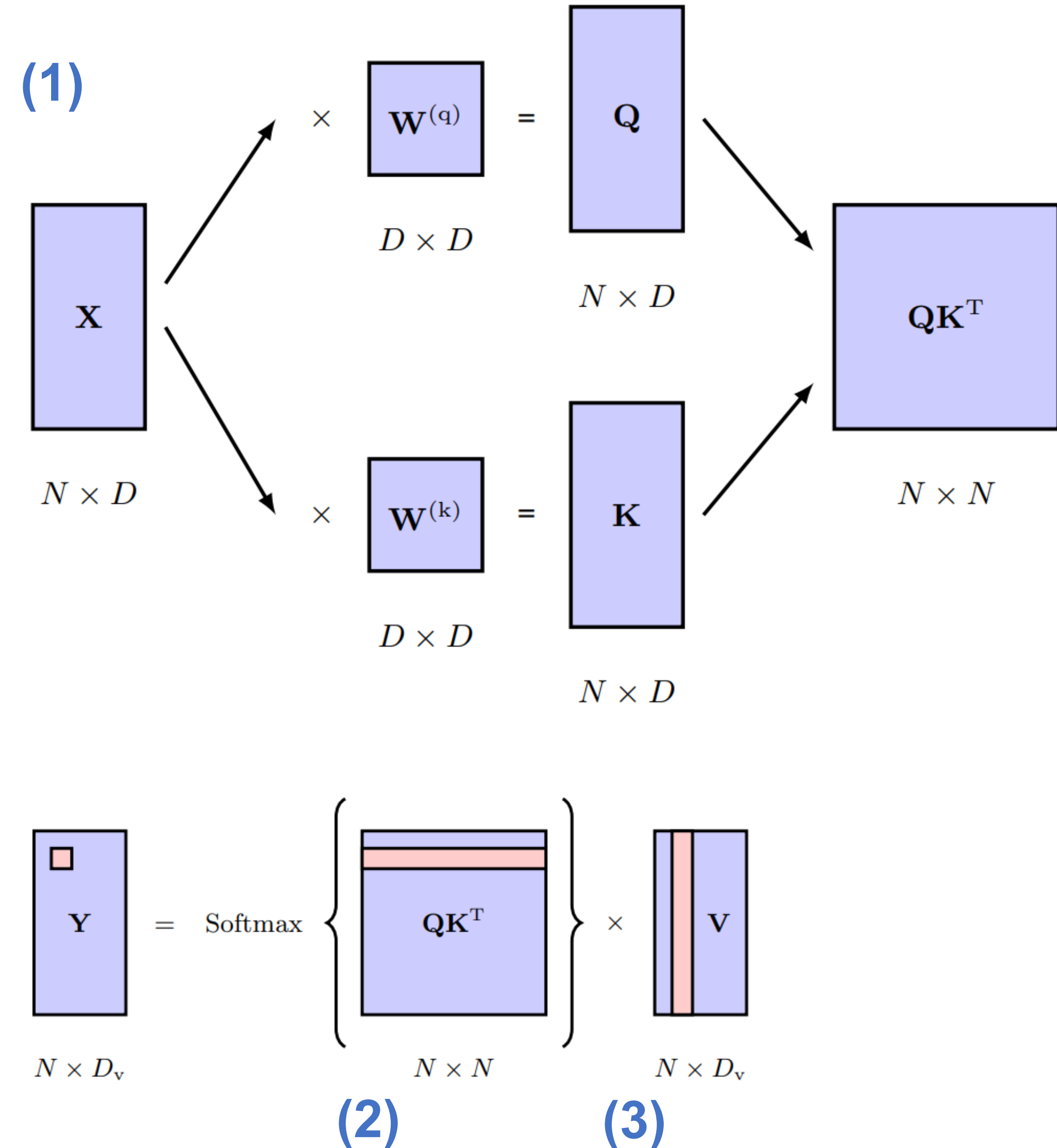
Divide by sqrt(d) to avoid issues with vanishing gradients when d is large

Step 3: Compute final MHA output

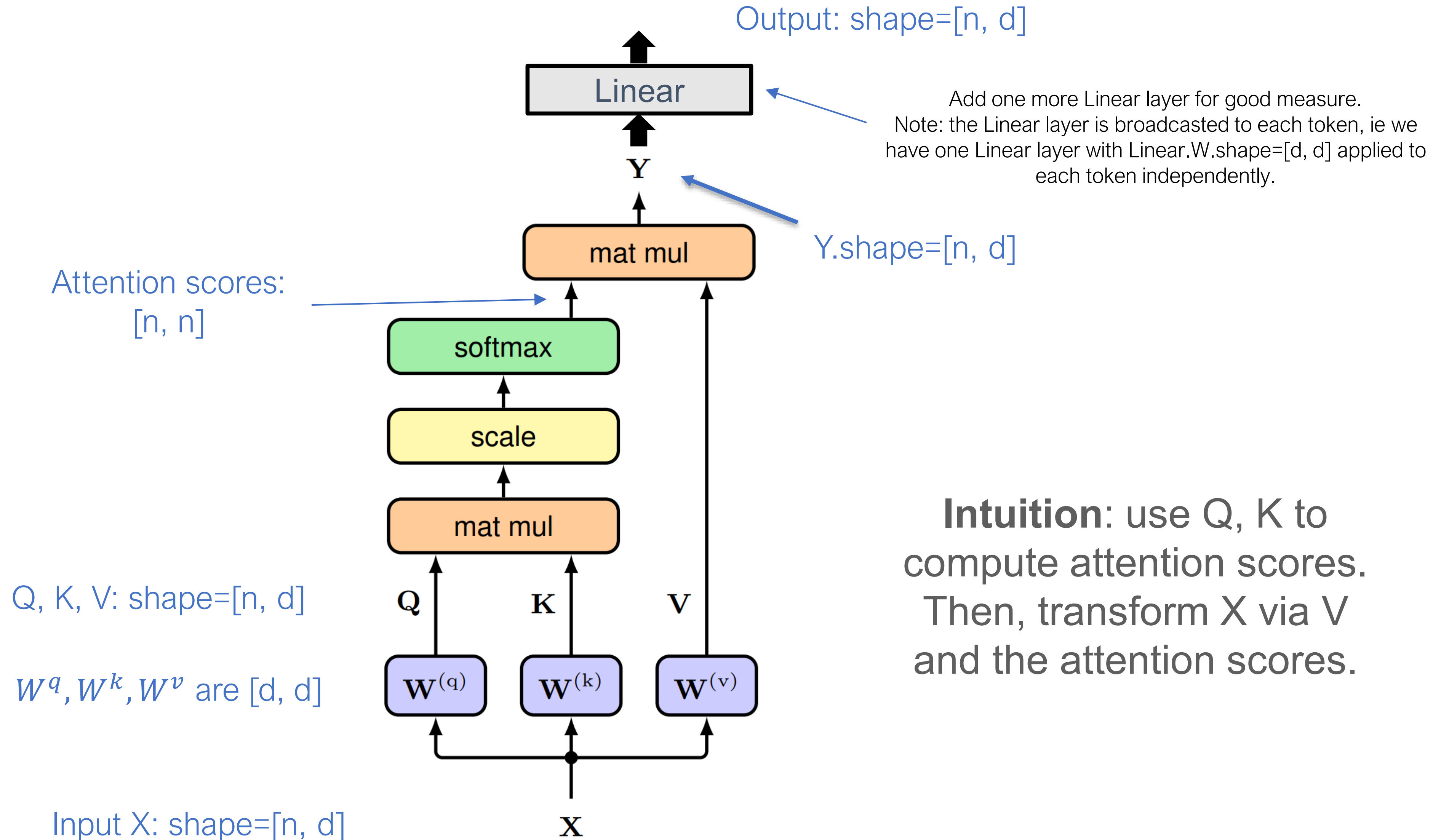
$$Y = softmax\left(\frac{QK^T}{\sqrt{d}}\right)V$$

shape=[n, d]

...and repeat! Can easily stack MHA layers



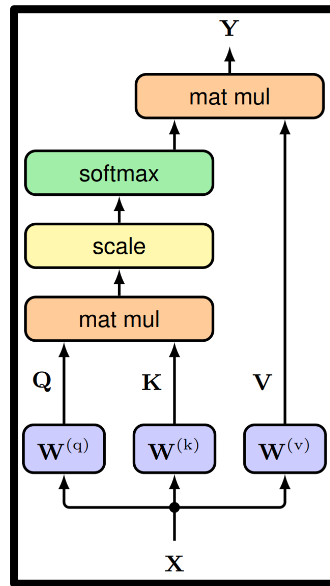
MHA (v0): information flow



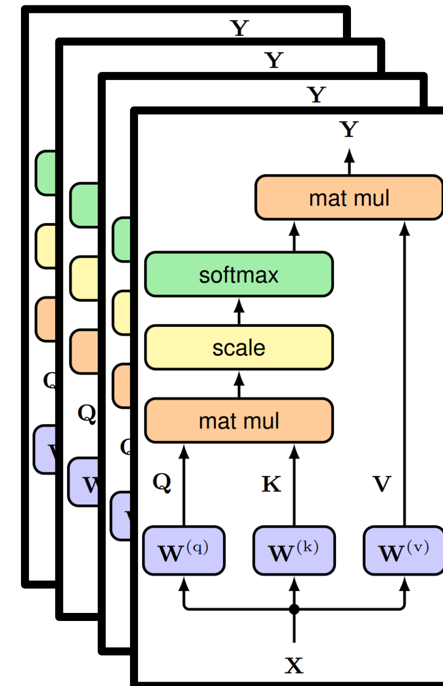
Intuition: use Q, K to compute attention scores. Then, transform X via V and the attention scores.

MHA (v1): multiple heads

Idea: let's learn multiple self-attention modules in parallel at a given level. Increases "width" of network.



"single head" attention

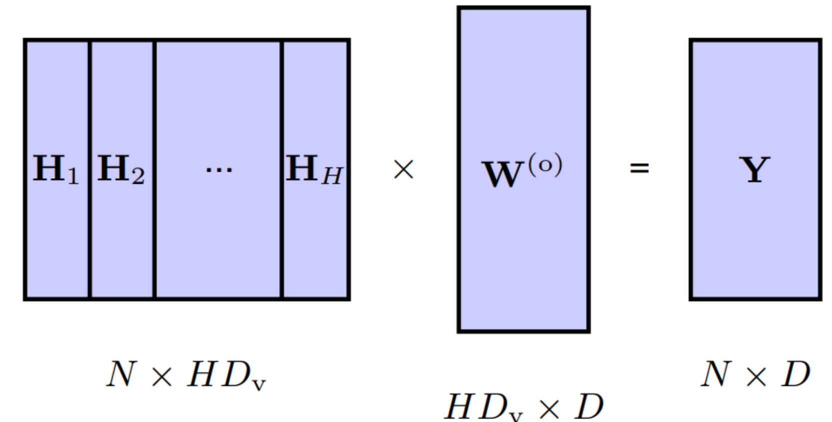


"multi head" attention

Here, num_heads=4

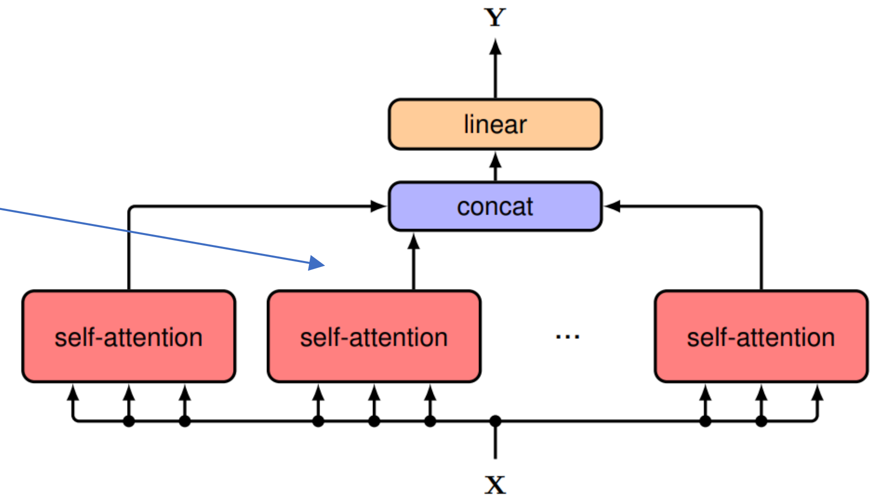
MHA (v1): multiple heads

Let `h` be the number of heads.
(1) Run h different independent self-attention blocks, to produce h different outputs with shape=[n, d]
(2) **Concatenate** all h outputs, and apply a learned linear transformation to produce the MHA output (shape=[n, d])



$$\mathbf{Q}_h = \mathbf{X}\mathbf{W}_h^{(q)}$$
$$\mathbf{K}_h = \mathbf{X}\mathbf{W}_h^{(k)}$$
$$\mathbf{V}_h = \mathbf{X}\mathbf{W}_h^{(v)}$$
$$\mathbf{H}_h = \text{Attention}(\mathbf{Q}_h, \mathbf{K}_h, \mathbf{V}_h)$$

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



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

shape=[n, h*d]

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

Note: h*d is pretty large! See next slide for what's done in practice ("split").

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

Orig paper uses 6 layers, with h=8 heads

Neat animation from: https://towardsdatascience.com/illustrated-guide-to-transformers-step-by-step-explanation-f74876522bc0

MHA (v1.5): multiple heads + split

In practice: to reduce computation costs, rather than have each self-attention head operate on the full embedding `d`, instead have each head operate on a reduced dimensionality d_h by splitting each Q_h, K_h, V_h matrix into `h` chunks.
Example: for d=16 and h=2 heads, $d_h = 8$

$$\mathbf{Q}_h = \mathbf{X}\mathbf{W}_h^{(q)}$$
$$\mathbf{K}_h = \mathbf{X}\mathbf{W}_h^{(k)}$$
$$\mathbf{V}_h = \mathbf{X}\mathbf{W}_h^{(v)}$$

X_h shape=[n, d]

Q_h, K_h, V_h Shape=[n, d_h]

W_h^q, W_h^k, V_h^v Shape=[d, d_h]

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

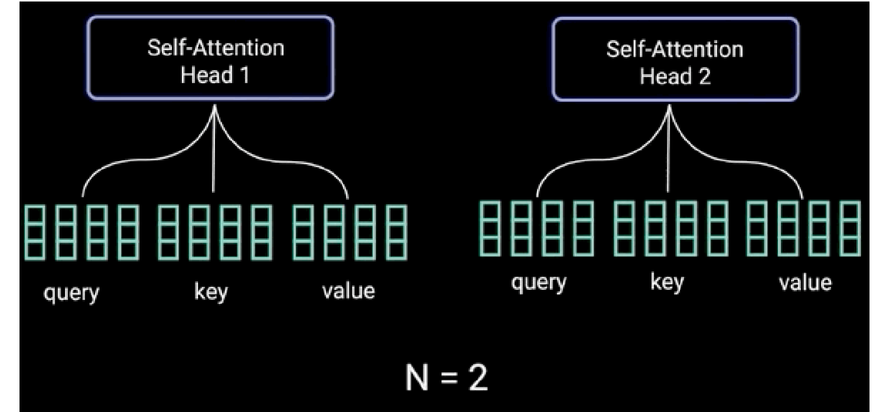
"effective" embed dimensionality for each head

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

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

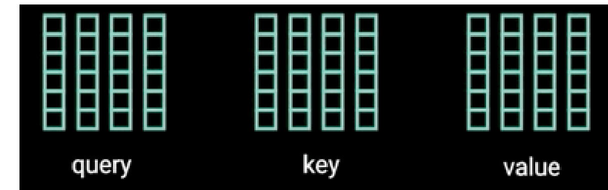
shape=[n, h*d_h] =[n, d]

Learned linear transform. Shape=[d, d]



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

Split (d -> d_h)



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

Interpretation: each head can focus on different input semantics. Similar to how Conv2d layers have multiple filters.

MHA: summary

Learns a transformation of input tokens, weighted by self-attention.

Input: tokens X, shape=$[n, d]$

Output: Y(X) shape=$[n, d]$

Also known as: "Scaled dot product attention"

$$\mathbf{Q}_h = \mathbf{X}\mathbf{W}_h^{(\mathrm{q})}$$
$$\mathbf{K}_h = \mathbf{X}\mathbf{W}_h^{(\mathrm{k})}$$
$$\mathbf{V}_h = \mathbf{X}\mathbf{W}_h^{(\mathrm{v})}$$

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

$$\mathrm{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) \equiv \mathrm{Softmax}\left[\frac{\mathbf{Q}\mathbf{K}^{\mathrm{T}}}{\sqrt{D_{\mathrm{k}}}}\right]\mathbf{V}.$$

$$\mathbf{Y}(\mathbf{X}) = \mathrm{Concat}\left[\mathbf{H}_1, \ldots, \mathbf{H}_H\right]\mathbf{W}^{(\mathrm{o})}$$

MHA: softmax scaling

When calculating attention scores, we divide by $\sqrt{D_k}$ (where D_k is the dimensionality of the key embedding) to avoid vanishing gradient problem of Softmax().

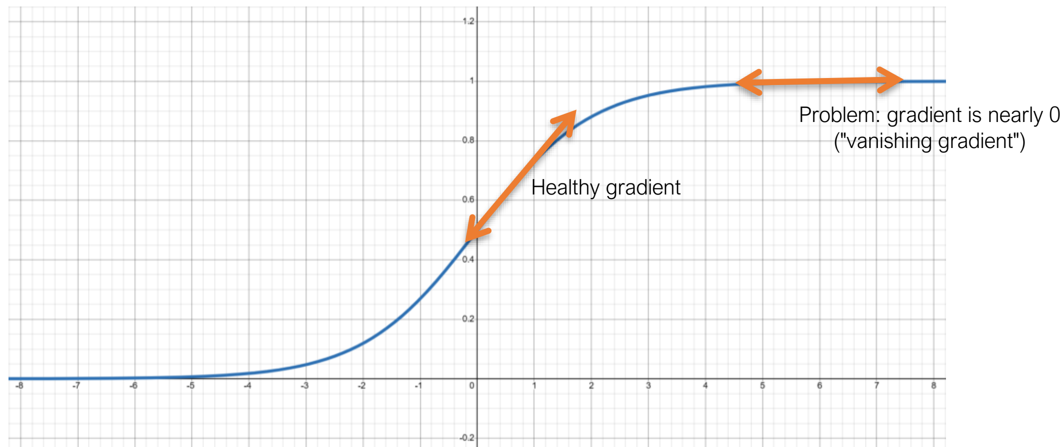
$$\mathbf{Q}_h = \mathbf{X}\mathbf{W}_h^{(\mathrm{q})}$$

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

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

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

$$\mathrm{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) \equiv \mathrm{Softmax}\left[\frac{\mathbf{Q}\mathbf{K}^{\mathrm{T}}}{\sqrt{D_{\mathrm{k}}}}\right]\mathbf{V}.$$



Note: with multiple heads, D_k should be the dimensionality of each head's key embedding, which typically is floor($\frac{d}{num_heads}$)

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 and MHA

Multi-head self attention (MHA) is a crucial building block for the transformer model.

Let's dig into how MHA is used to build a Transformer model

MLP, skip connections, LayerNorm1d

While $Y(X)$ is a nonlinear transformation of the inputs X (due to softmax nonlinearity in computing attention scores H), the nonlinearity is limited.

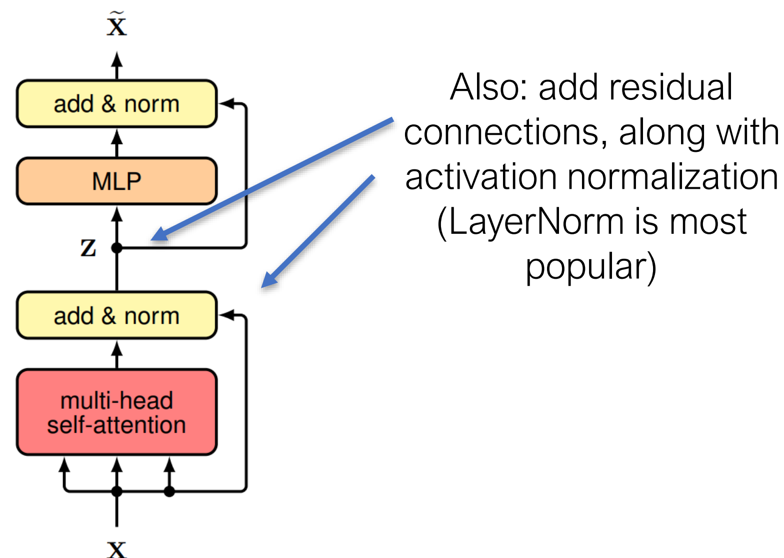
Solution: add an additional MLP block to learn a richer nonlinear transformation of Y(X).

$$\mathbf{Q}_h = \mathbf{X}\mathbf{W}_h^{(\mathrm{q})}$$
$$\mathbf{K}_h = \mathbf{X}\mathbf{W}_h^{(\mathrm{k})}$$
$$\mathbf{V}_h = \mathbf{X}\mathbf{W}_h^{(\mathrm{v})}$$

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

$$\mathrm{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) \equiv \mathrm{Softmax}\left[\frac{\mathbf{Q}\mathbf{K}^{\mathrm{T}}}{\sqrt{D_{\mathrm{k}}}}\right]\mathbf{V}.$$

$$\mathbf{Y}(\mathbf{X}) = \mathrm{Concat}\left[\mathbf{H}_1, \ldots, \mathbf{H}_H\right]\mathbf{W}^{(\mathrm{o})}$$



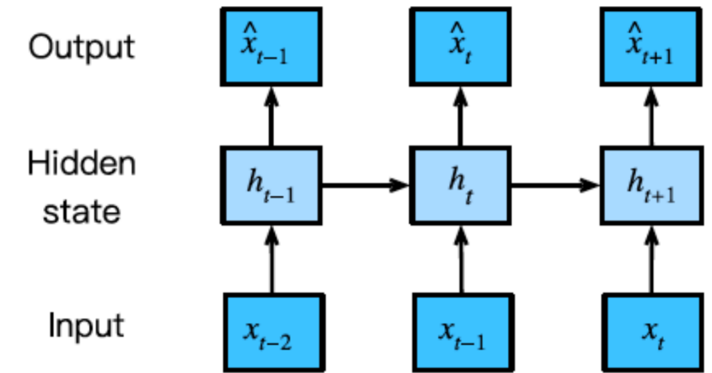
Transformers: ordered sequences

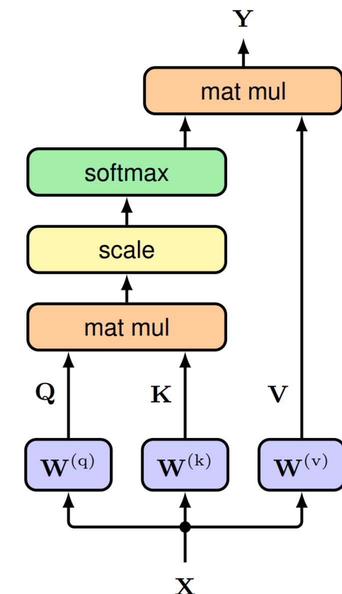
Recall: in RNNs, we iteratively feed each input token to the model one at a time

- 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...





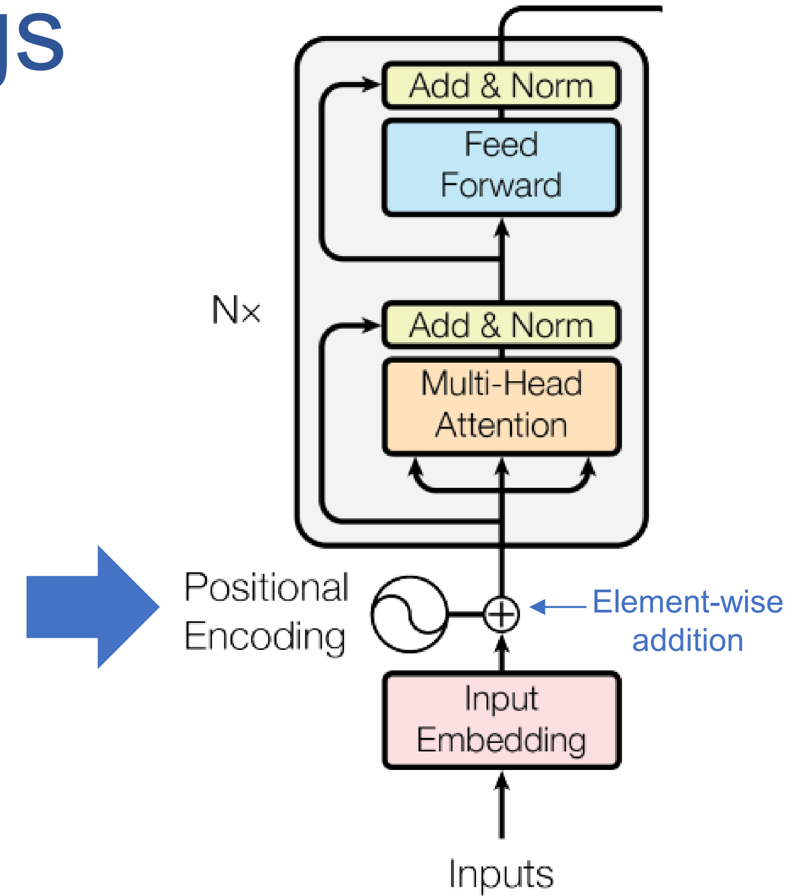
Positional encodings

The transformers solution: **positional encodings**

Goal: make MHA "position aware", either via absolute or relative position techniques.

Absolute position encoding: 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

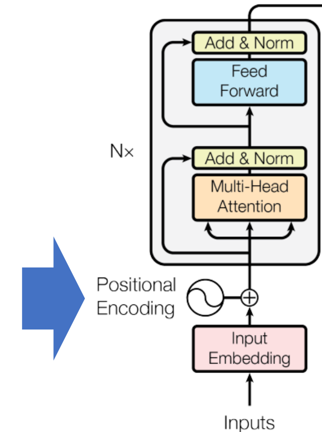
Requirement: Positional embedding must encode the "token position"



Absolute 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

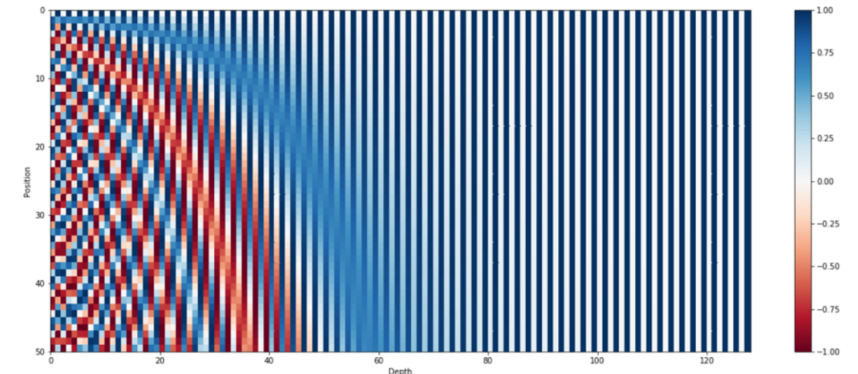


Figure 2 - The 128-dimensional positonal encoding for a sentence with the maximum lenght of 50. Each row represents the embedding vector $\vec{p_t}$

For more details/intuition on sin/cos, see: [link]

https://kazemnejad.com/blog/transformer_architecture_positional_encoding/

Positional encodings approaches

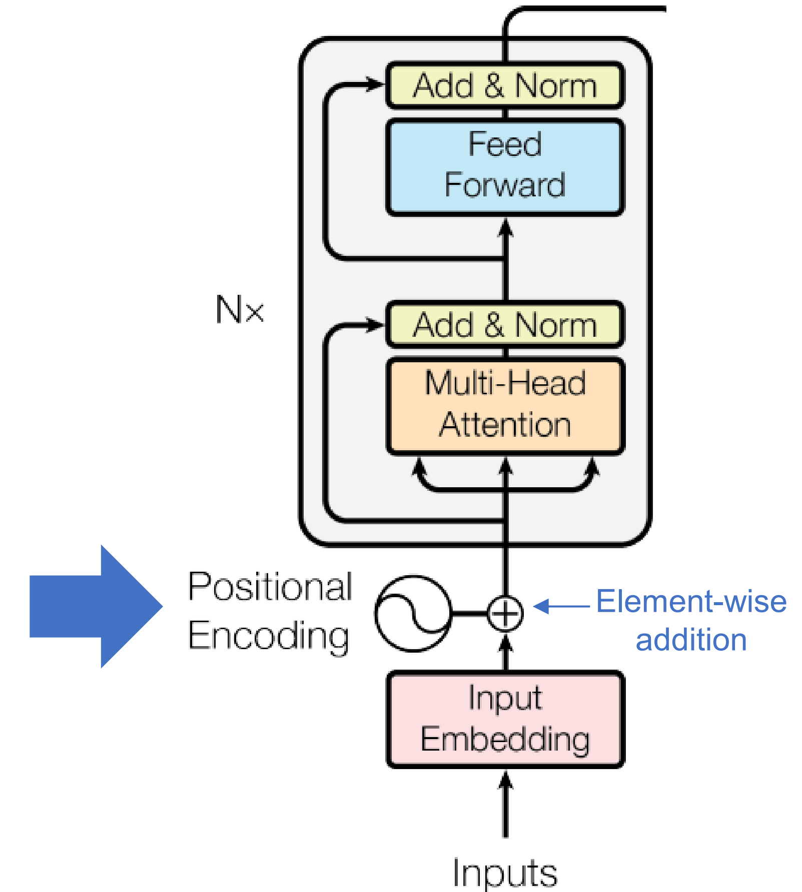
As of 2026, there have been several advancements in positional encodings.

Examples:

- RoPE (2021): "Rotary Position Embedding".
- ALiBi (2022): "Attention with Linear Biases"
- ...

Each approach tackles the questions:

- How to effectively encode position into MHA?
- Does this allow us to extrapolate to longer sequence lengths?
    - Ex: Train on seq_len=512, test on seq_len=1024? Especially for LLM chatbots, being able to extend context length is valuable!



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)

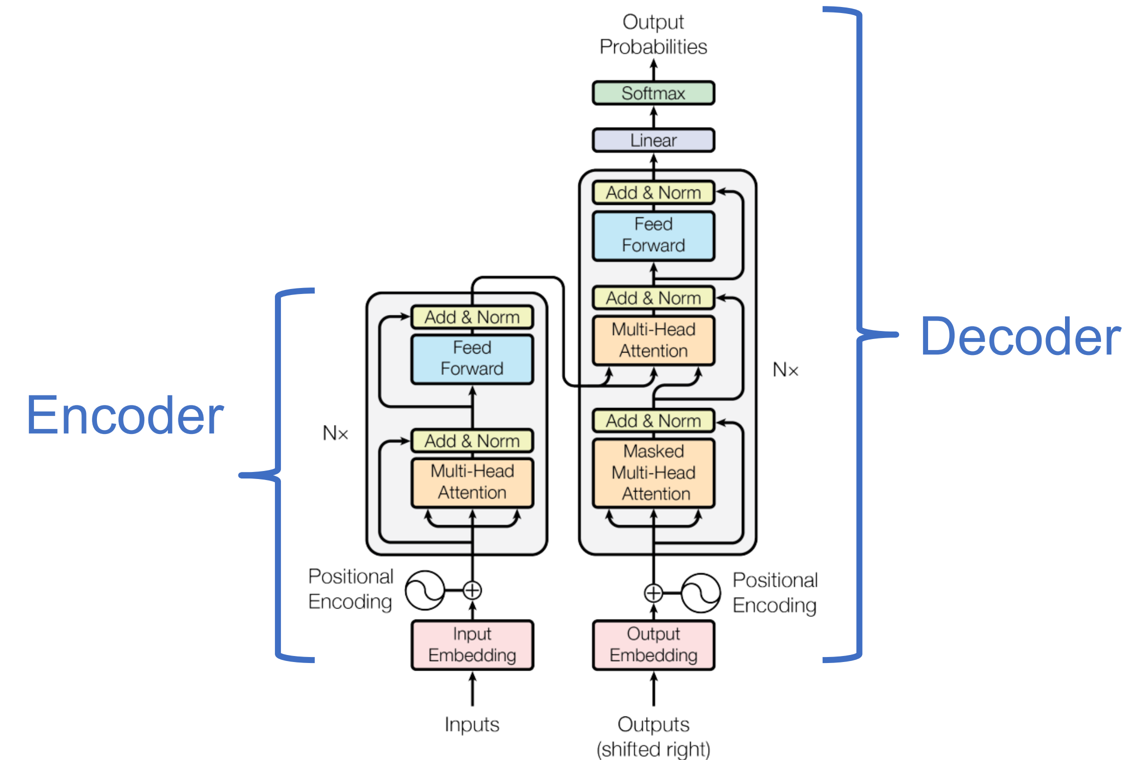


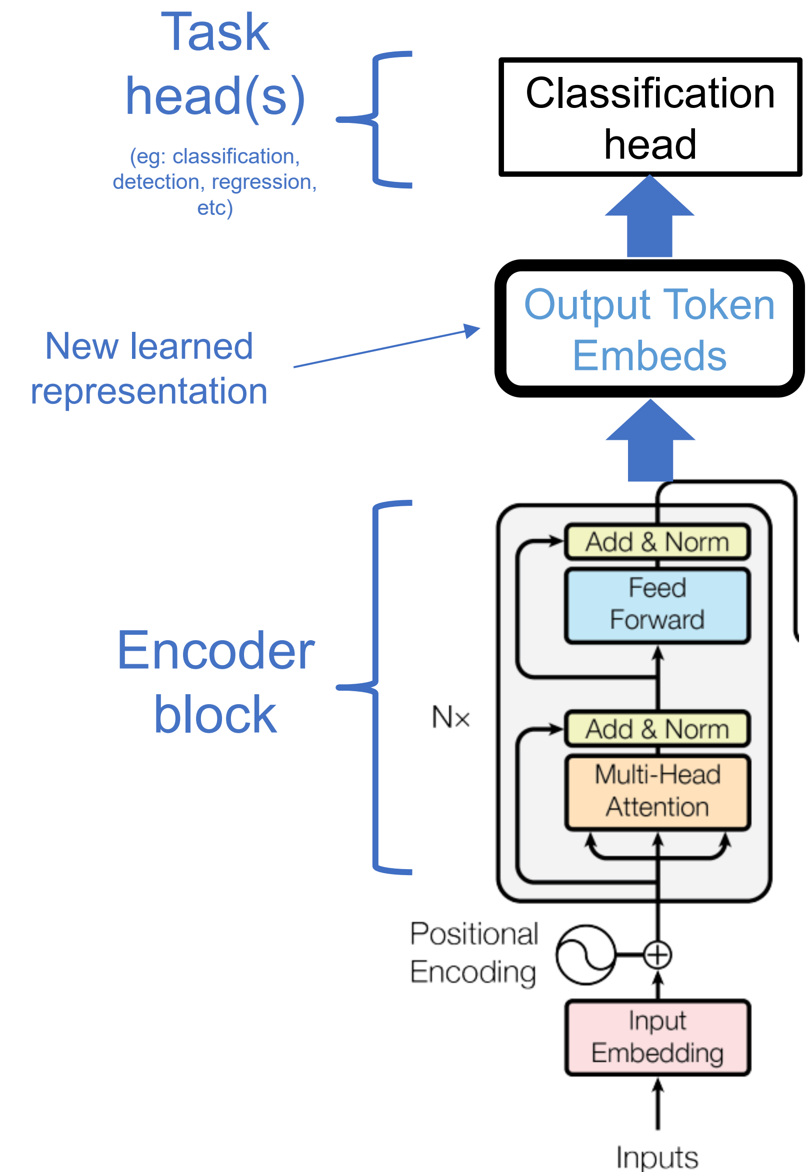
Figure 1: The Transformer - model architecture.

Encoder

Input: X with shape=$[B, n, d]$

Output: Z with shape=$[B, n, d]$

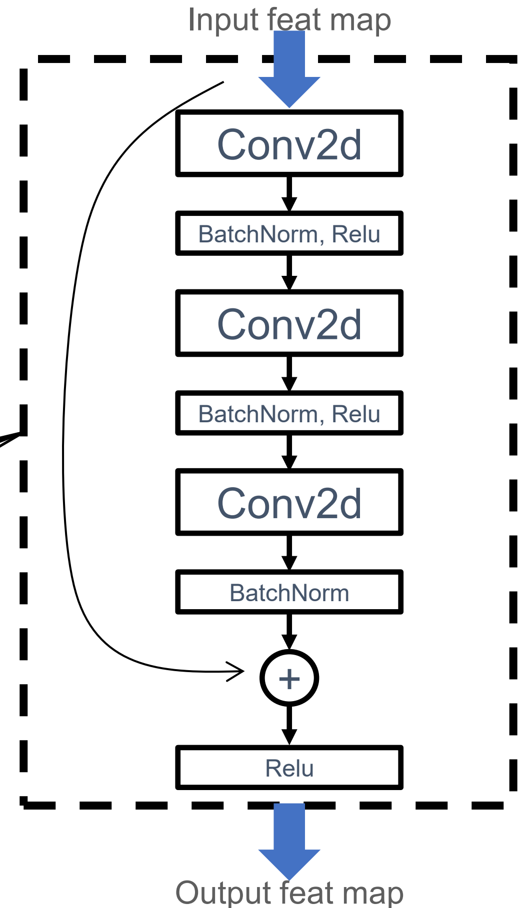
Idea: learn a representation of the input sequence that is good for downstream tasks (eg classification, text generation, etc)



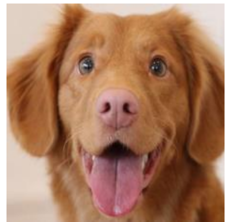
Terminology: NN "blocks"

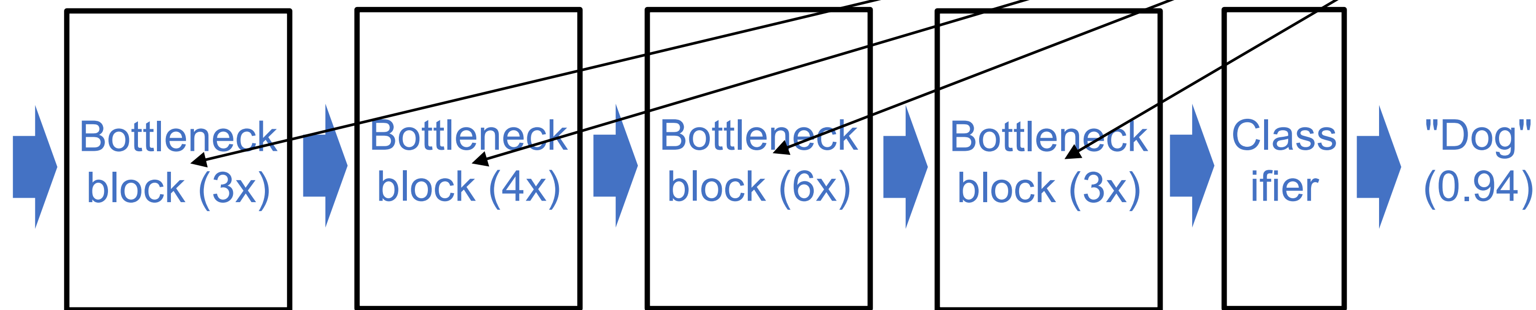
Recall: it's common to define a DNN in terms of "blocks", rather than individual layers

- 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

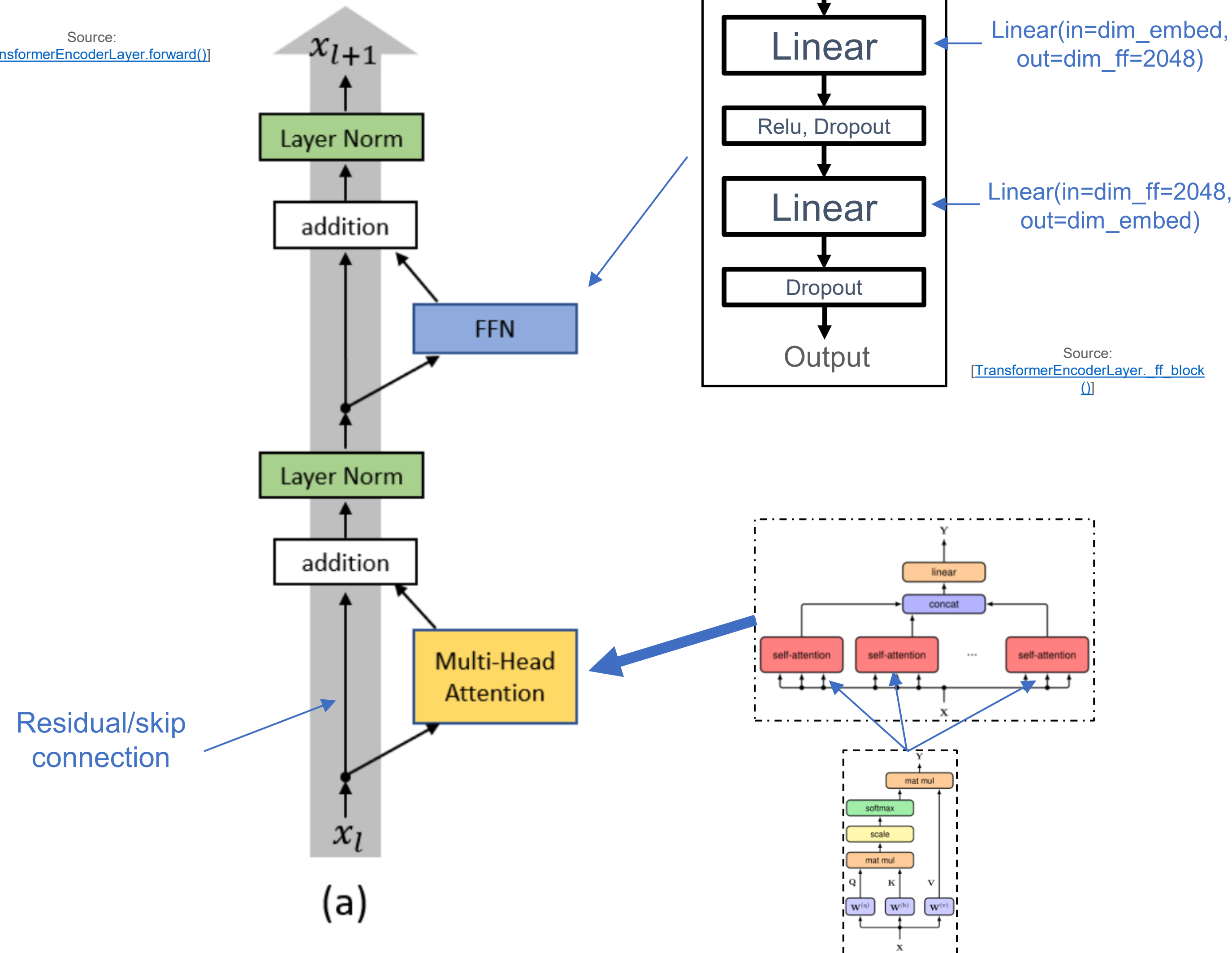




* Some things missing from this resnet50 picture: initial "stem", downsampling (spatial resolution) layers, other minor details

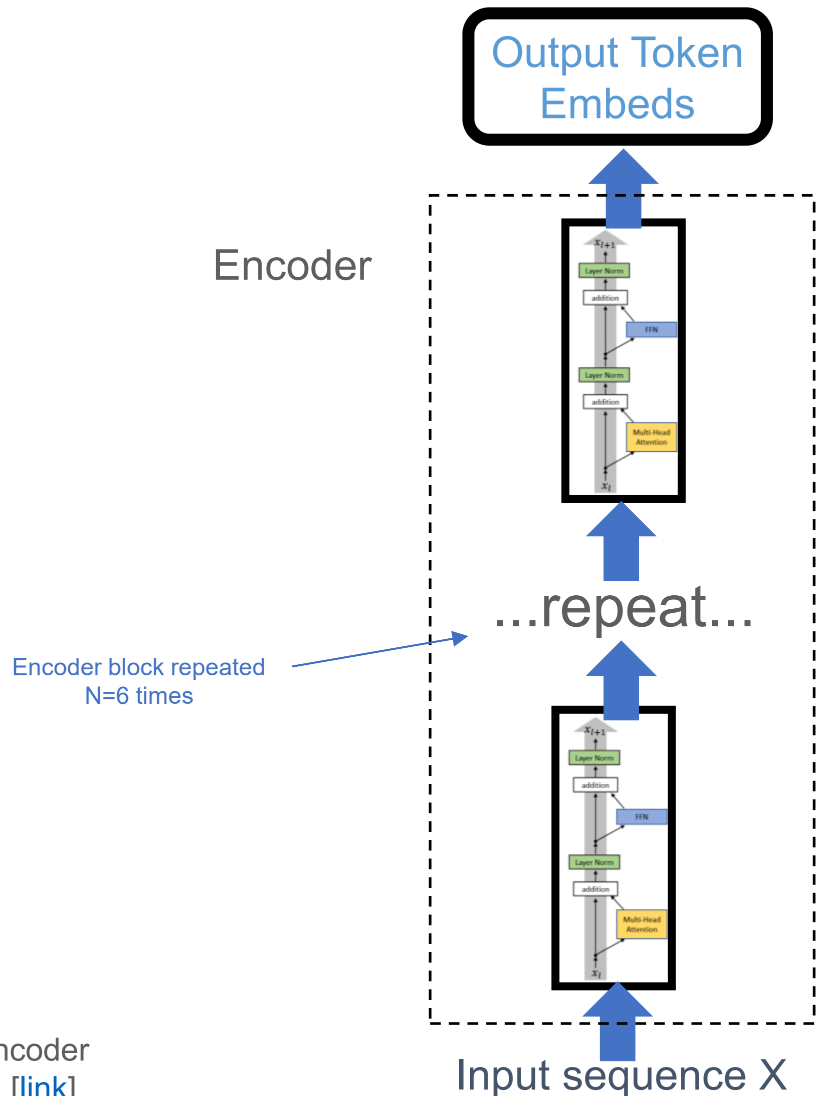
If you're curious here is pytorch's (torchvision) implementation of ResNet50 (in all of its gory detail): [link]
Tip: you should be able to read this code and understand 90% of what's going on! The implementation is complicated because it's super generic+flexible, but with some study you can see what's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

Encoder block

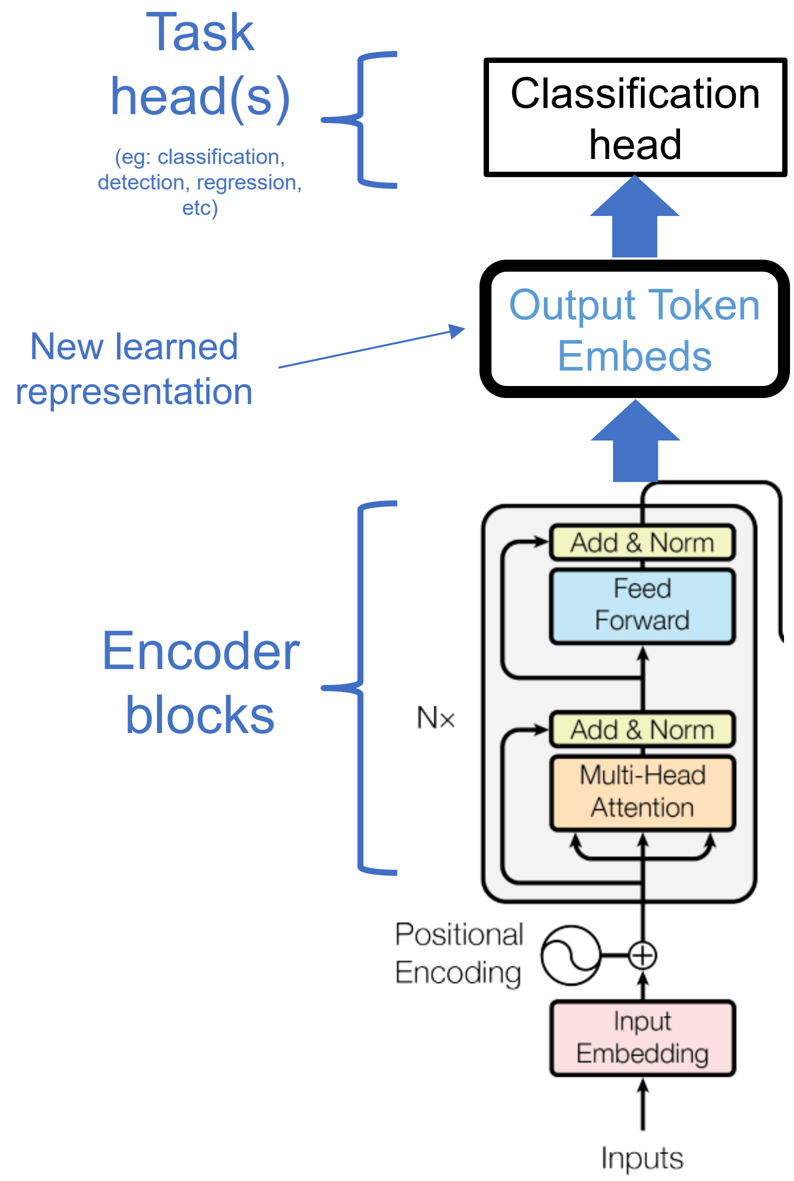


Important: FFN is broadcasted to each token, independently.
In other words, the same Linear layer(s) are applied to each token independently (not a different Linear layer for each token position, nor a giant Linear layer that mixes all token embeddings, that would be too expensive!)

Encoder Arch



(drawn another way)



TransformerEncoder
pytorch code: [link]

Transformer encoder takeaways

Input: X with shape=$[B, n, d]$

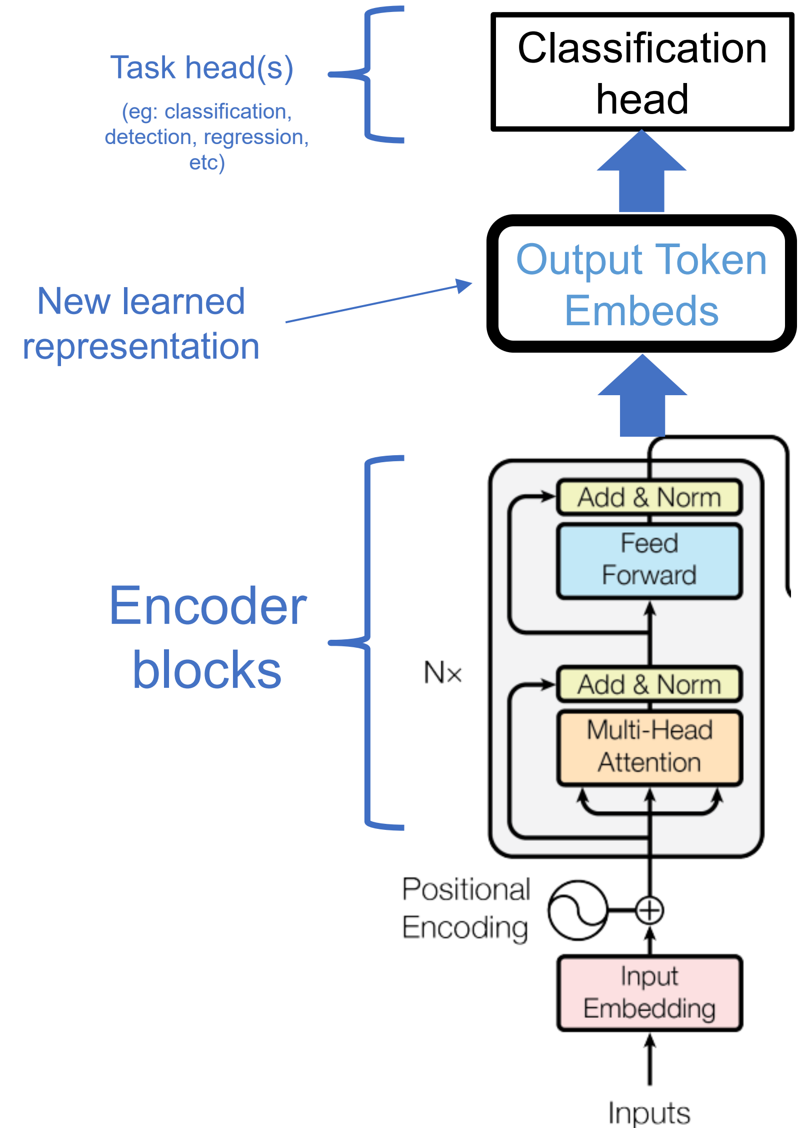
Output: Z with shape=$[B, n, d]$

Idea: learn a representation of the input sequence that is good for downstream tasks (eg classification, text generation, etc)

MHA: capture cross-token interactions via scaled-dot product attention.

Learn multiple heads in parallel to increase model **width**.

Stack encoder blocks sequentially to increase model **depth**.



Detail: Pre-norm vs Post-norm

Original transformers paper did "Post-LayerNorm" (Post-LN)

But, people found out that "Pre-LayerNorm" (Pre-LN) works better

- Better training stability, can use larger learning rate, etc
- For more details, see: [link]

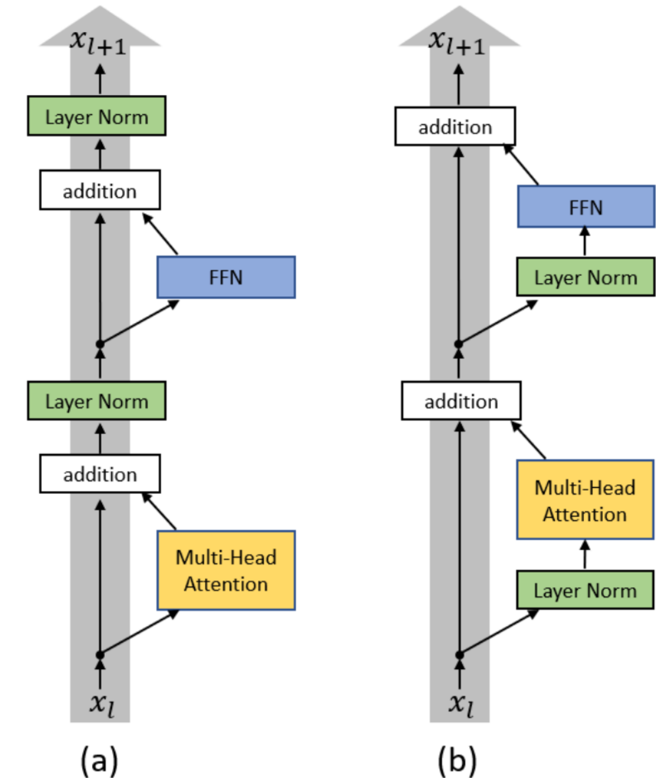


Figure 1. (a) Post-LN Transformer layer; (b) Pre-LN Transformer layer.

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

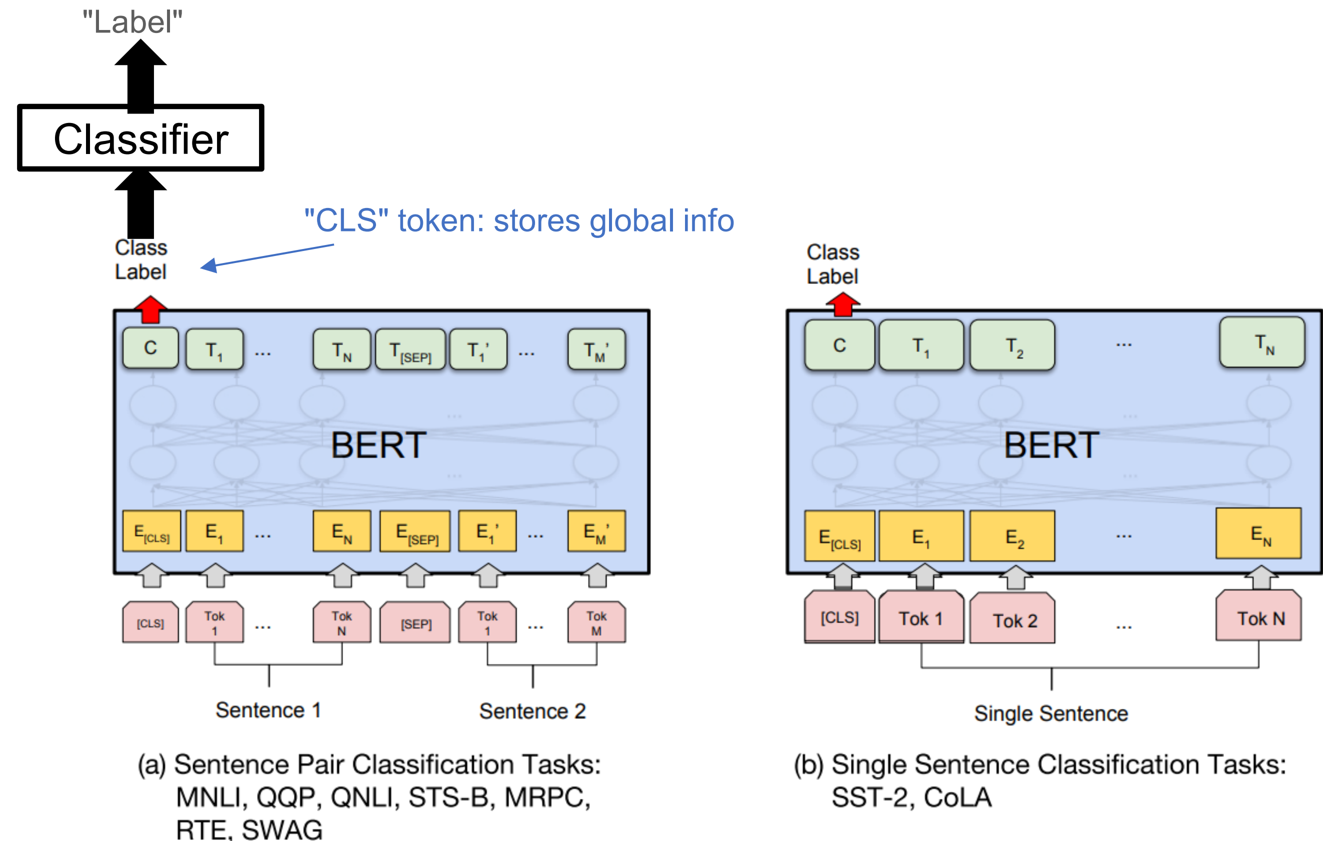
Application: text classification (BERT)

BERT ("Bidirectional Encoder Representations from Transformers", 2018): [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/summary" sentence representation

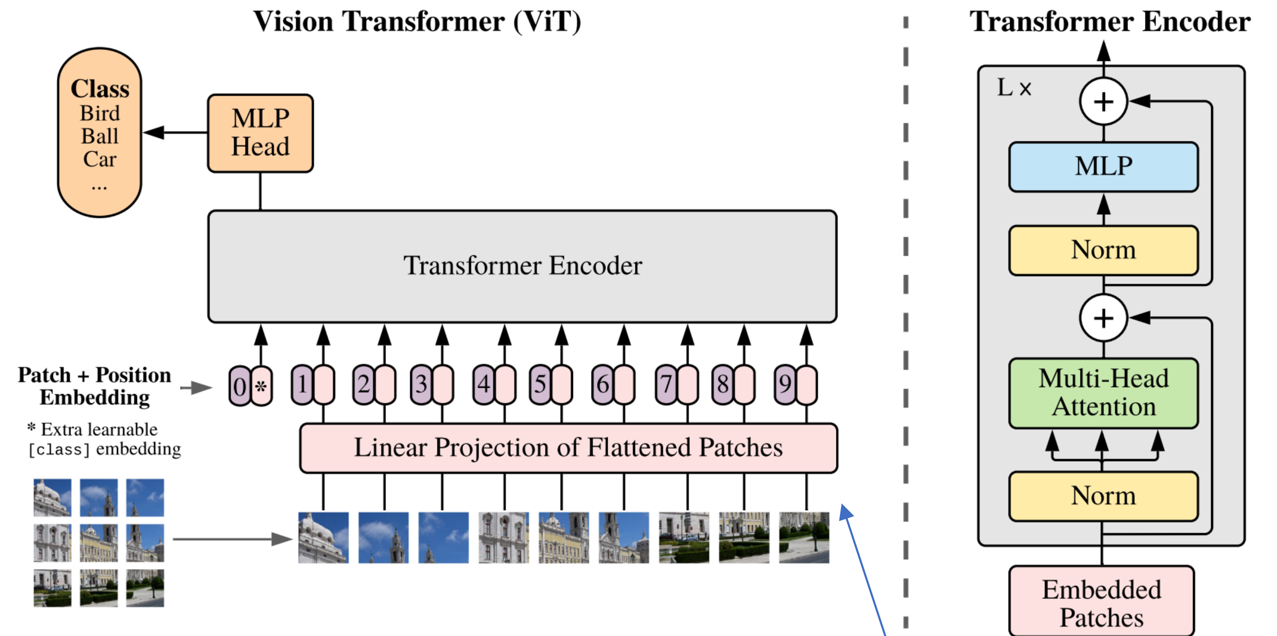


Application: image classification (ViT)

"An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale" [link] (2021).

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