Vision Transformers

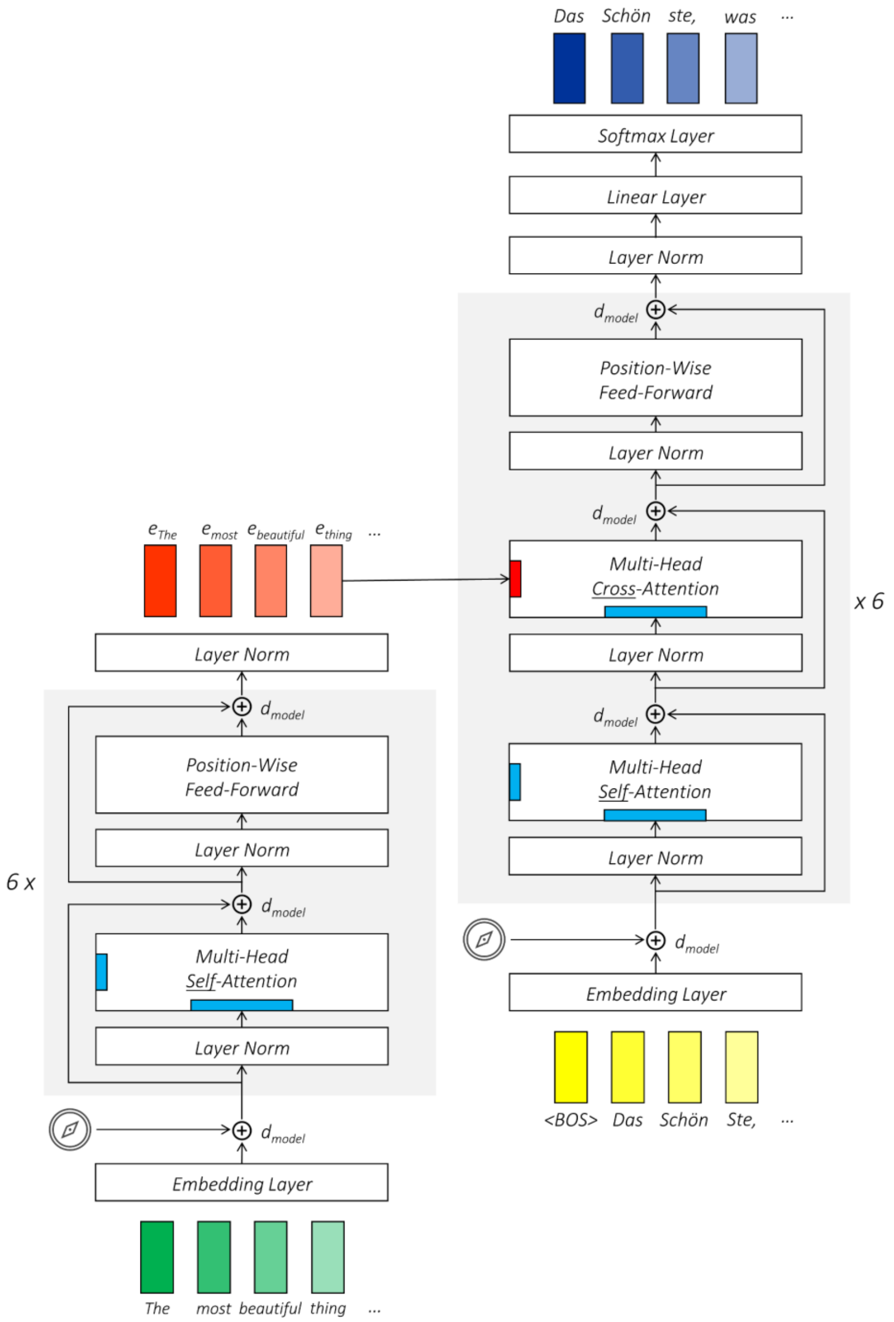
Nilanjan Ray

Plan

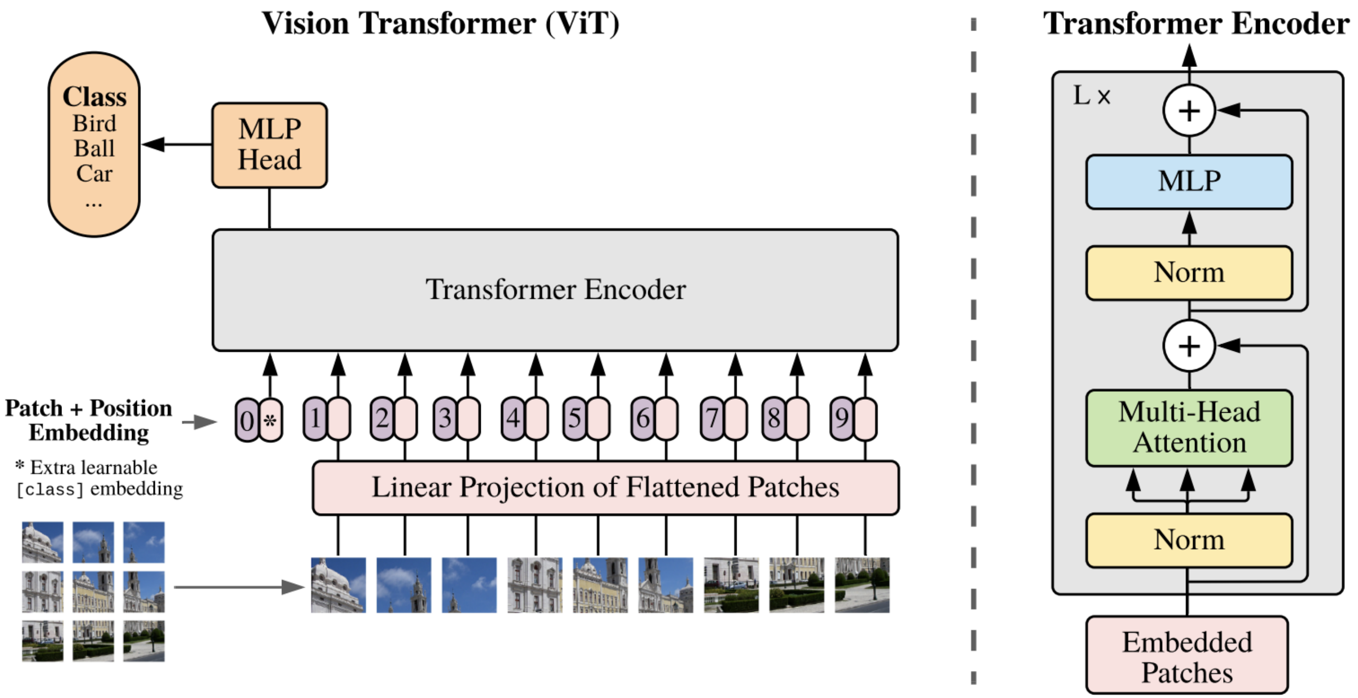
- To understand vision transformers, we need to understand how a transformer works
- There are plenty of tutorials out there. Let's use this one: https://github.com/sgrvinod/a-PyTorch-Tutorial-to-Machine-Translation
- We will take about 15 mins to go through this page. Then, we will come back to describe a vision transformer (ViT) for image classification.
- After studying ViT, we will study how image captions can be generated using transformers.

Transformer: Encoder and Decoder

https://github.com/sgrvinod/a-PyTorch-Tutorial-to-Transformers?tab=readme-ov-file



What are ViTs?



Only encoders are used in ViT, no decoders are needed for image classification

Tutorial: https://uvadlc-notebooks.readthedocs.io/en/latest/tutorial_notebooks/tutorial15/Vision_Transformer.html

Let’s dissect “VisionTransformer” module

- Image shape
  - print(CIFAR_images.shape)
  - torch.Size([4, 3, 32, 32]): batch size, #channels, image height, image width
- Image to patches:
  - img_patches = img_to_patch(CIFAR_images, patch_size=4, flatten_channels=True)
  - print(img_patches.shape)
  - torch.Size([4, 64, 48]): batch size, #patches, (#channels)x(patch height)x(patch width)
- Preprocess patches by linear layer
  - Input shape to linear layer: [4, 64, 48]
  - Output shape from the linear layer: [4, 64, 256]. 256 is the embedding dimension
- CLS tokens shape: [1, 1, 256] but it is repeated for batches, so shape becomes [4, 1, 256]
- Shape after CLS tokens concatenated to output of linear embedding: [4, 65, 256].
- This tensor is added to the positional embedding.
- Positional embedding shape: [1, 65, 256]
- Transformer input and output - note that 0^{th} and 1^{st} dimensions are exchanged
  - Input shape to transformer: [65, 4, 256]
  - Output shape from transformer: [65, 4, 256]
- MLP head input and output
  - Input shape to MLP head: [4, 256]
  - Output shape from MLP head: [4, 10]

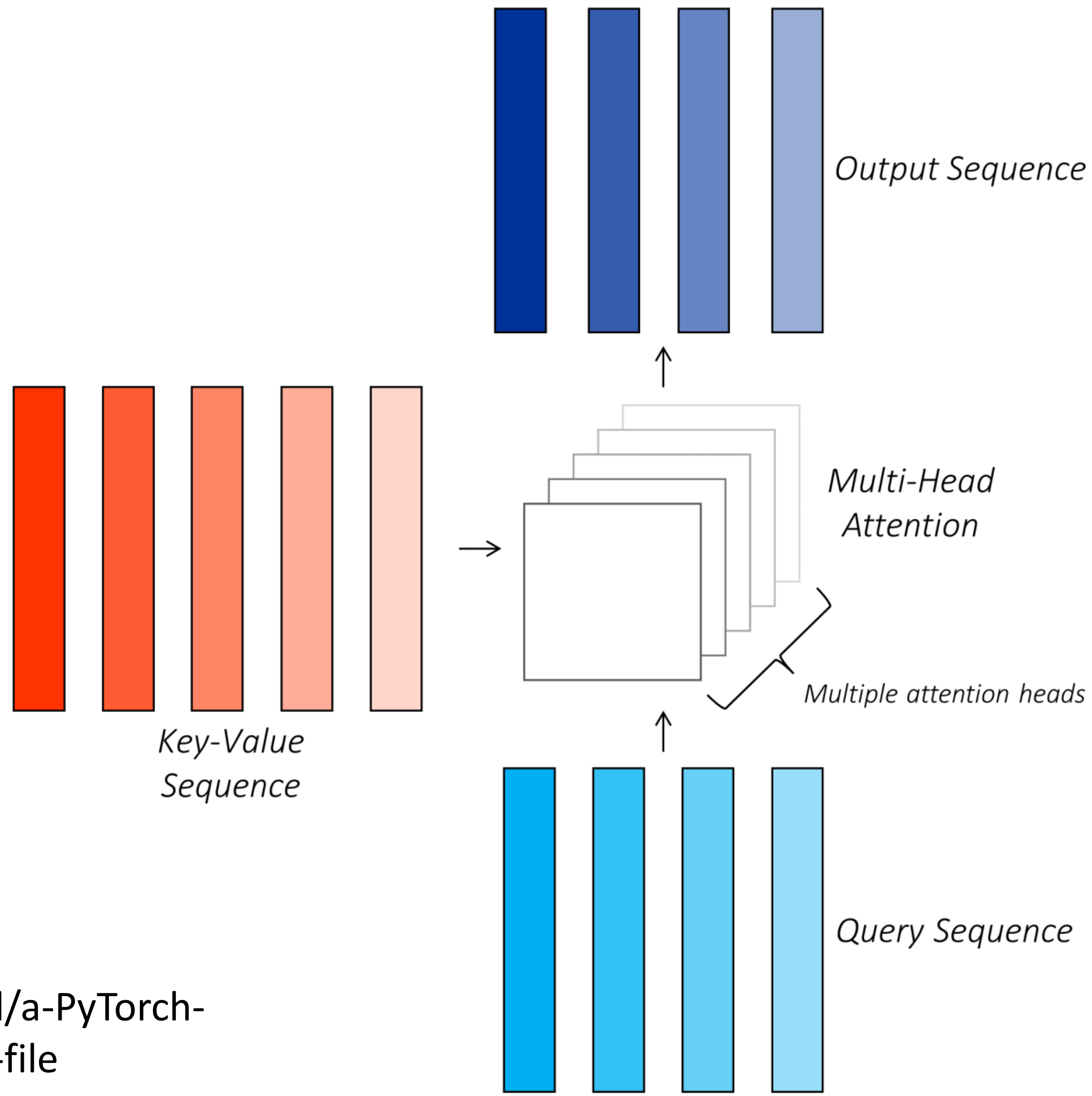
```python
def forward(self, x):
    # Preprocess input
    x = img_to_patch(x, self.patch_size)
    B, T, _ = x.shape
    x = self.input_layer(x)

    # Add CLS token and positional encoding
    cls_token = self.cls_token.repeat(B, 1, 1)
    x = torch.cat([cls_token, x], dim=1)
    x = x + self.pos_embedding[:,:T+1]

    # Apply Transforrmer
    x = self.dropout(x)
    x = x.transpose(0, 1)
    x = self.transformer(x)

    # Perform classification prediction
    cls = x[0]
    out = self.mlp_head(cls)
    return out
```

Self.transformer: Sequentially applies a few attention blocks

- Input shape to the Attn block: [65, 4, 256]
- Output shape from the Attn block: [65, 4, 256]
- Residual connections, linear layers, layer normalizations only work on the last dimensions of the tensor [65, 4, 256].
- So where is the mixing happening?
    - It must be happening inside the nn.MultiheadAttention module

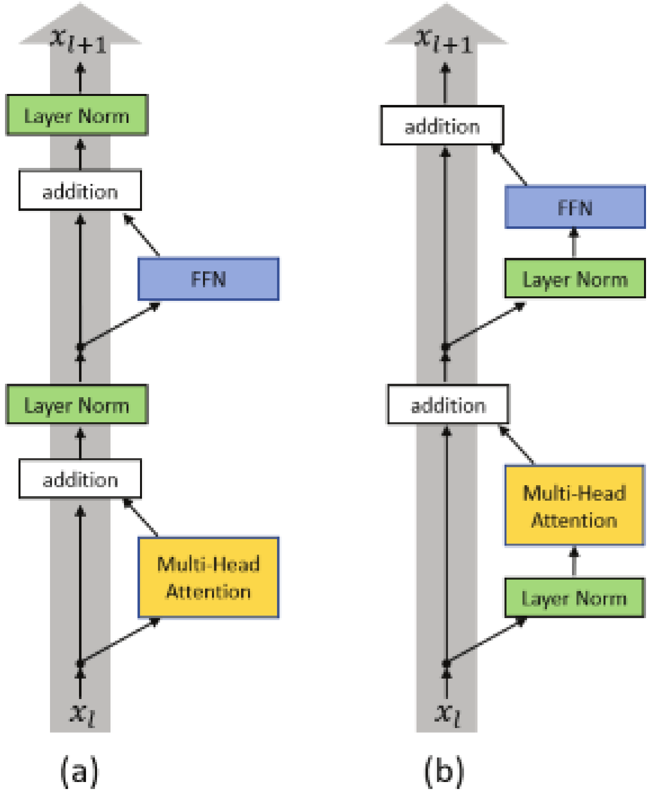
```python
class AttentionBlock(nn.Module):

    def __init__(self, embed_dim, hidden_dim, num_heads, dropout=0.0):
        super().__init__()

        self.layer_norm_1 = nn.LayerNorm(embed_dim)
        self.attn = nn.MultiheadAttention(embed_dim, num_heads,
                                          dropout=dropout)
        self.layer_norm_2 = nn.LayerNorm(embed_dim)
        self.linear = nn.Sequential(
            nn.Linear(embed_dim, hidden_dim),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_dim, embed_dim),
            nn.Dropout(dropout)
        )


    def forward(self, x):
        print("Input shape to Attn block:",x.shape)
        inp_x = self.layer_norm_1(x)
        x = x + self.attn(inp_x, inp_x, inp_x)[0]
        x = x + self.linear(self.layer_norm_2(x))
        return x
```

Multi-head Attention



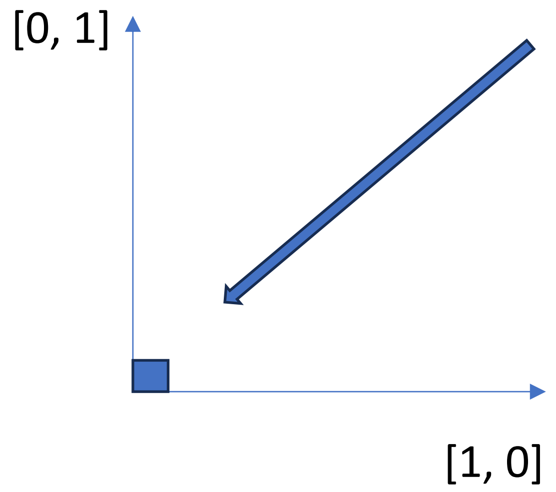
Details here: https://github.com/sgrvinod/a-PyTorch-Tutorial-to-Transformers?tab=readme-ov-file

Transformer encoder



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

Vectors and their relationships



The relationship between 2 vectors is represented by their angle

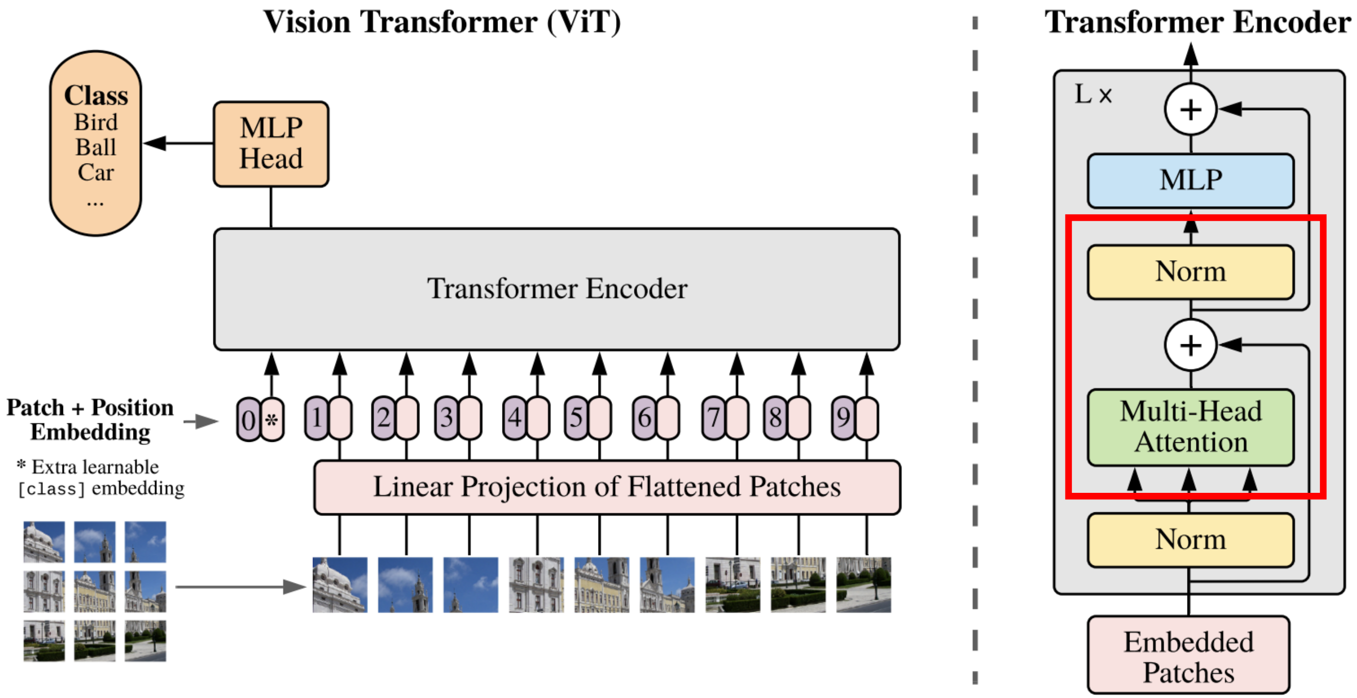
Consine similarity formula

$$\cos\theta = \frac{\vec{a}\cdot\vec{b}}{\|\vec{a}\|\|\vec{b}\|} \qquad \frac{\textit{dot product}}{\textit{Norms}}$$

$$\|\vec{a}\| = \sqrt{a_1^2 + a_2^2 + a_3^2 + \cdots + a_n^2}$$

$$\|\vec{b}\| = \sqrt{b_1^2 + b_2^2 + b_3^2 + \cdots + b_n^2}$$

Attention



$$\frac{\textit{dot product}}{\textit{Norms}}$$

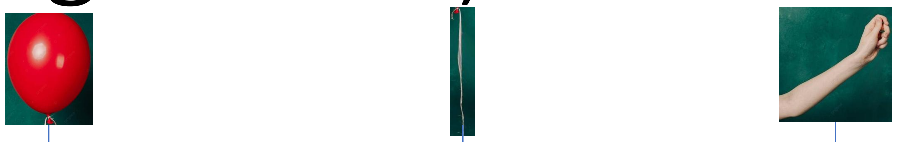
Attention

Query
(Question)

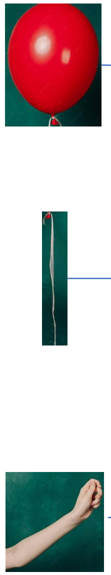
Key
(Matching)

Value
(Answer)

Attention (Single-head)

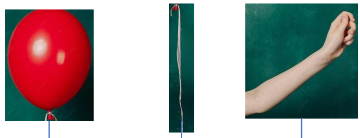


Linear



Linear

	$\boxed{Q_1}$	Q_2	Q_3
K_1	Q_1K_1	Q_2K_1	Q_3K_1
$\boxed{K_2}$	Strong relationship	Q_2K_2	Q_3K_2
K_3	Positive relationship	Strong relationship	Q_3K_3

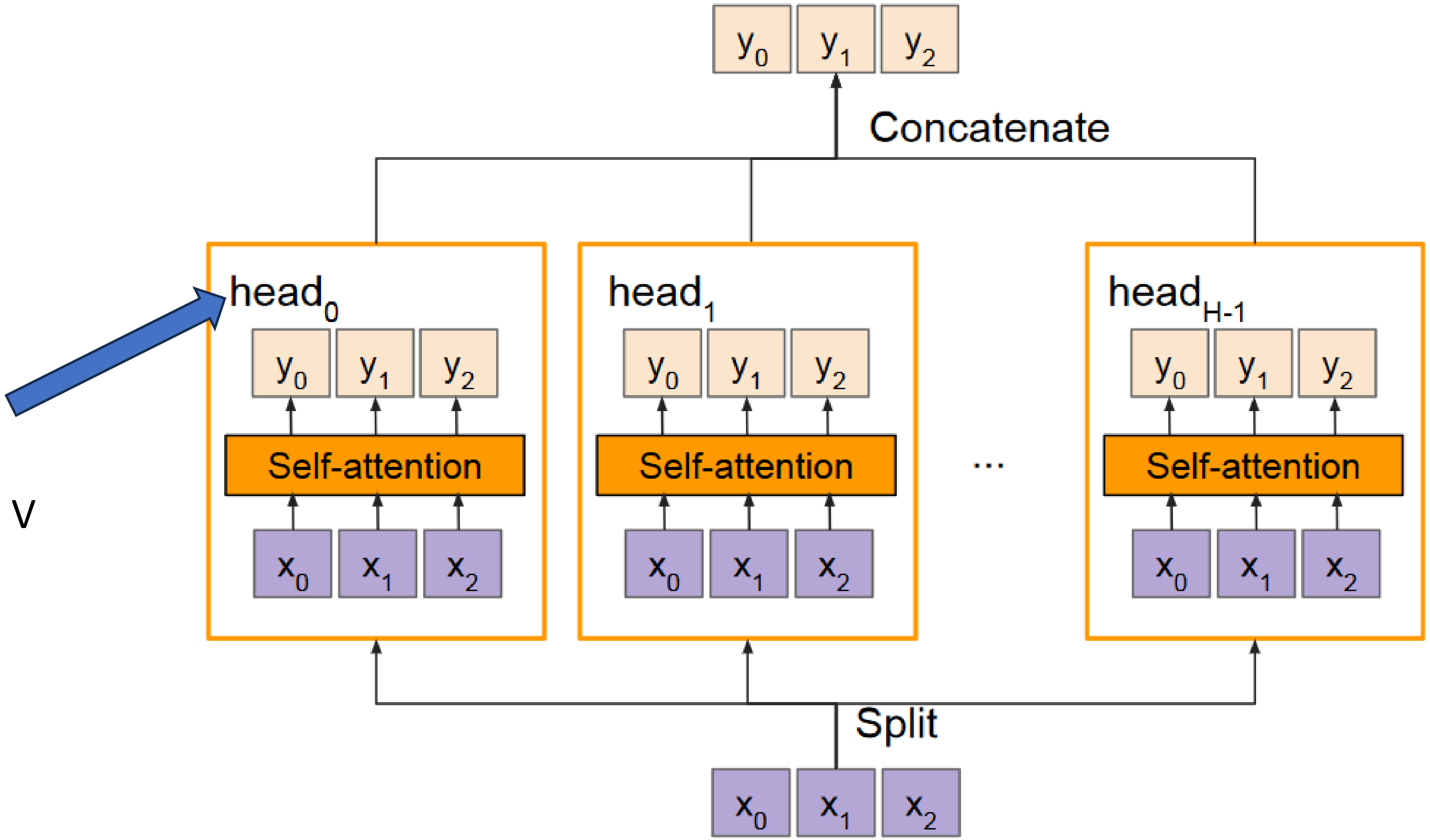


Linear

	V_1	V_2	V_3
Q_1K_2	Bloon attached to string	Bloon attached to string	Hand holds bloon
Q_1K_3	Hand holds bloon	Hand holds bloon	Hand holds bloon
Q_3K_2	Hand holds bloon	Hand holds string	Hand holds string
...	...	...	...

Multi-head attention

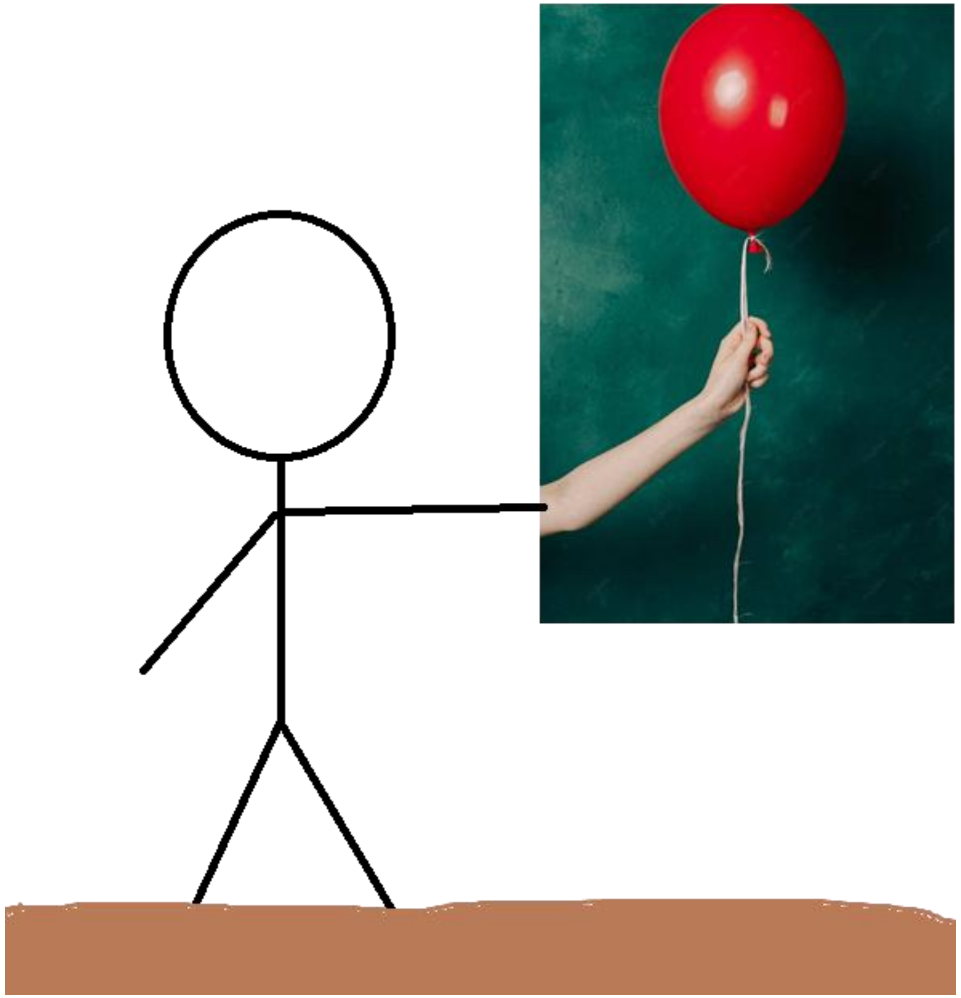
Each head is a separate Q, K, V pair



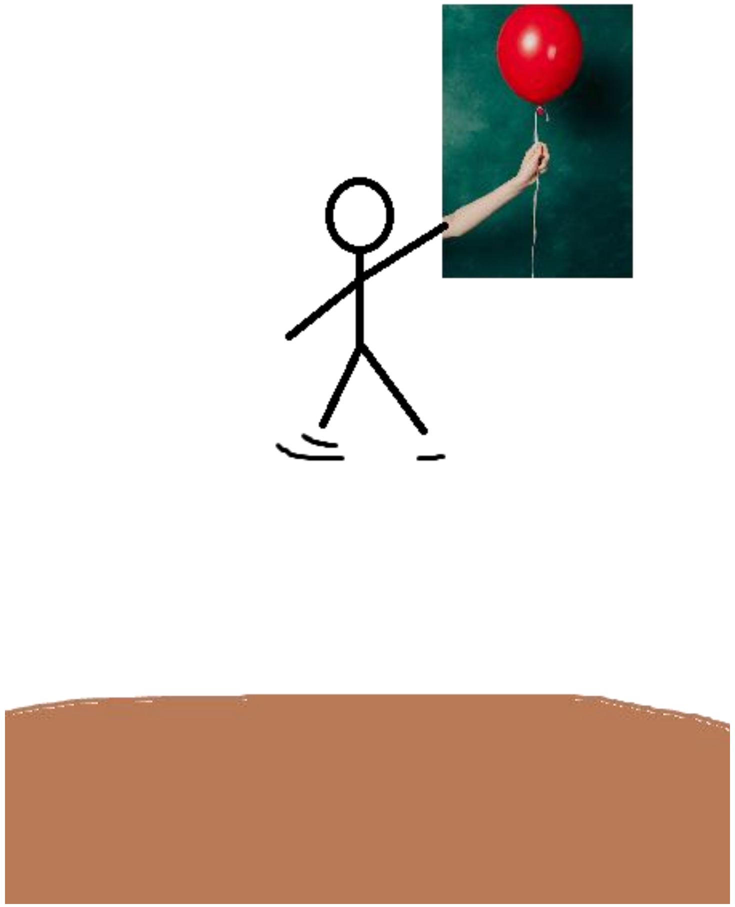
Multiple self-attention modules in parallel

Source: http://cs231n.stanford.edu/slides/2022/lecture_11_ruohan.pdf

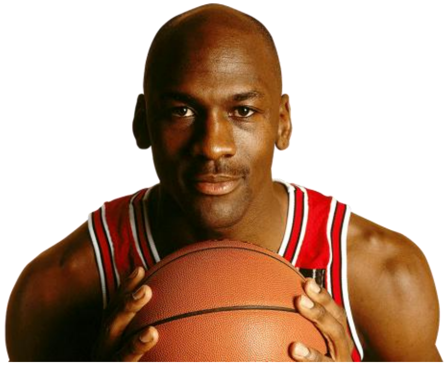
Heads



Different pictures describe different scenarios for the same object



Heads (Michael, depending on the context)



nn.MultiheadAttention module

Multihead attention is defined as:

$$\text{Multihead}(Q,K,V) = \text{Concat}(\text{head}_1,\ldots,\text{head}_h)W^O$$
$$\text{where } \text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, V\boxed{W_i^V})$$

A separate Linear layer learned by each head

W's are learnable parameters:

$W_{1\ldots h}^Q \in \mathbb{R}^{D\times d_k}$, $W_{1\ldots h}^K \in \mathbb{R}^{D\times d_k}$, $W_{1\ldots h}^V \in \mathbb{R}^{D\times d_v}$, and $W^O \in \mathbb{R}^{h\cdot d_k \times d_{out}}$

Attention function is defined as:

$$\text{Attention}(Q,K,V) = \text{softmax}\left(\frac{QK^T}{\boxed{\sqrt{d_k}}}\right)V$$

Scaling factor to resolve *curse of dimensionality*

where

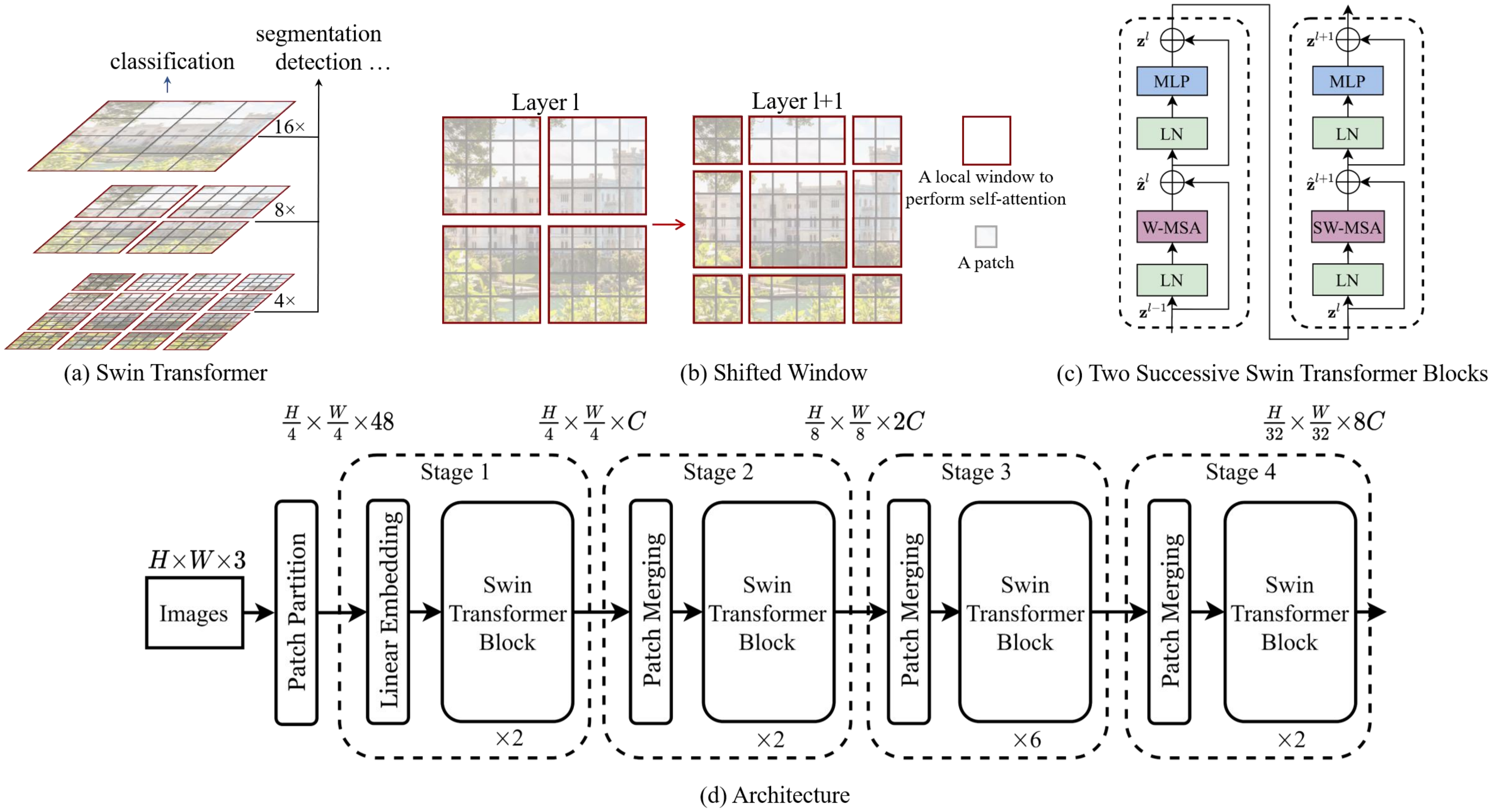
queries $Q \in \mathbb{R}^{T\times d_k}$, keys $K \in \mathbb{R}^{T\times d_k}$ and values $V \in \mathbb{R}^{T\times d_v}$

Source: https://uvadlc-notebooks.readthedocs.io/en/latest/tutorial_notebooks/tutorial6/Transformers_and_MHAttention.html

A simple implementation using PyTorch modules

```python
import torch, torch.nn as nn

class PatchEmbed(nn.Module):
    def __init__(self, in_ch=3, dim=128, patch=4):
        super().__init__()
        self.proj = nn.Conv2d(in_ch, dim, patch, patch)
    def forward(self, x):
        return self.proj(x).flatten(2).transpose(1,2)  # [B, N, D]

class ViT(nn.Module):
    def __init__(self, img=32, patch=4, dim=128, heads=4, depth=4, n_cls=10):
        super().__init__()
        self.patch_embed = PatchEmbed(3, dim, patch)
        n_p = (img//patch)**2
        self.cls = nn.Parameter(torch.zeros(1,1,dim))
        self.pos = nn.Parameter(torch.randn(1,n_p+1,dim))
        enc = nn.TransformerEncoderLayer(dim, heads, dim*4, batch_first=True)
        self.enc = nn.TransformerEncoder(enc, depth)
        self.head = nn.Linear(dim, n_cls)
    def forward(self, x):
        B = x.size(0)
        x = self.patch_embed(x)
        x = torch.cat([self.cls.expand(B,-1,-1), x], 1) + self.pos
        return self.head(self.enc(x)[:,0])
```

SOTA architecture: Swin transformers



(a) Swin Transformer (b) Shifted Window (c) Two Successive Swin Transformer Blocks

(d) Architecture

https://github.com/microsoft/Swin-Transformer

Revisiting Convnet: ConvNeXt

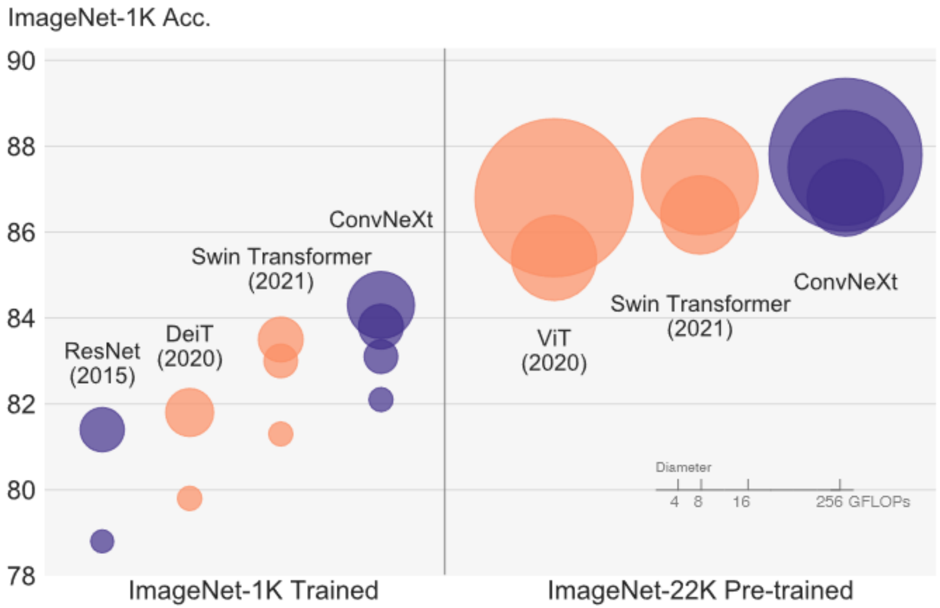


Figure 1. **ImageNet-1K classification** results for • ConvNets and ○ vision Transformers. Each bubble's area is proportional to FLOPs of a variant in a model family. ImageNet-1K/22K models here take $224^2/384^2$ images respectively. ResNet and ViT results were obtained with improved training procedures over the original papers. We demonstrate that a standard ConvNet model can achieve the same level of scalability as hierarchical vision Transformers while being much simpler in design.

Pure conv net:

- Very few non-linear functions
- LN instead of BN
- 7x7 conv
- Gelu instead of Relu

https://arxiv.org/pdf/2201.03545.pdf

And the revenge of ViT!

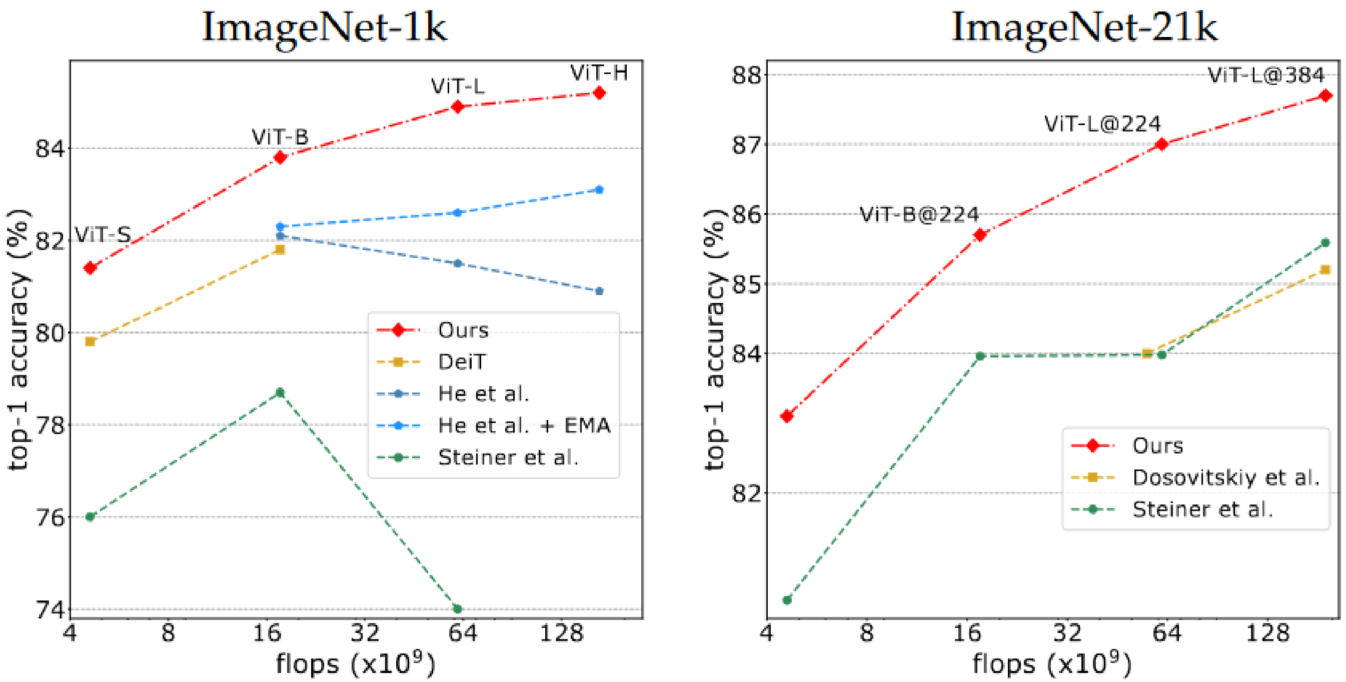


Figure 1: Comparison of training recipes for (*left*) vanilla vision transformers trained on ImageNet-1k and evaluated at resolution 224×224, and (*right*) pre-trained on ImageNet-21k at 224×224 and fine-tuned on ImageNet-1k at resolution 224×224 or 384×384.

https://arxiv.org/pdf/2204.07118.pdf

Image Captioning

Flickr8k

A child in a pink dress is climbing up a set of stairs in an entry way .

A girl going into a wooden building .

A little girl climbing into a wooden playhouse .

A little girl climbing the stairs to her playhouse .

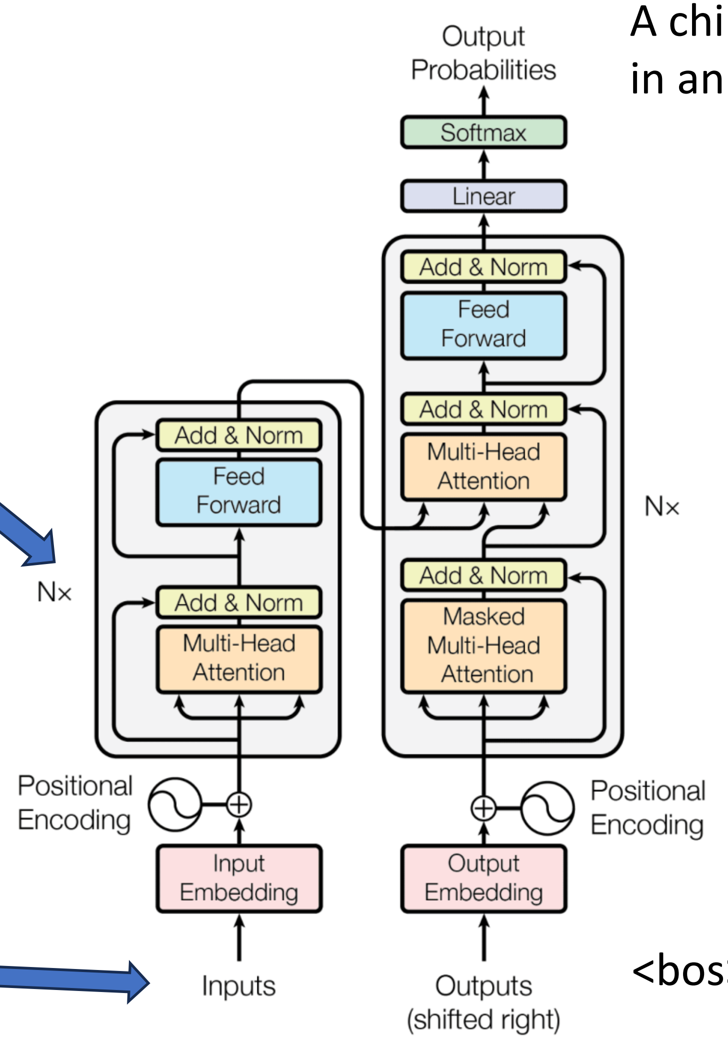
A little girl in a pink dress going into a wooden cabin .

Transformer Encoder-Decoder (Generation)

A child in a pink dress is climbing up a set of stairs in an entry way . <eos> <pad> <pad> ..

ViT

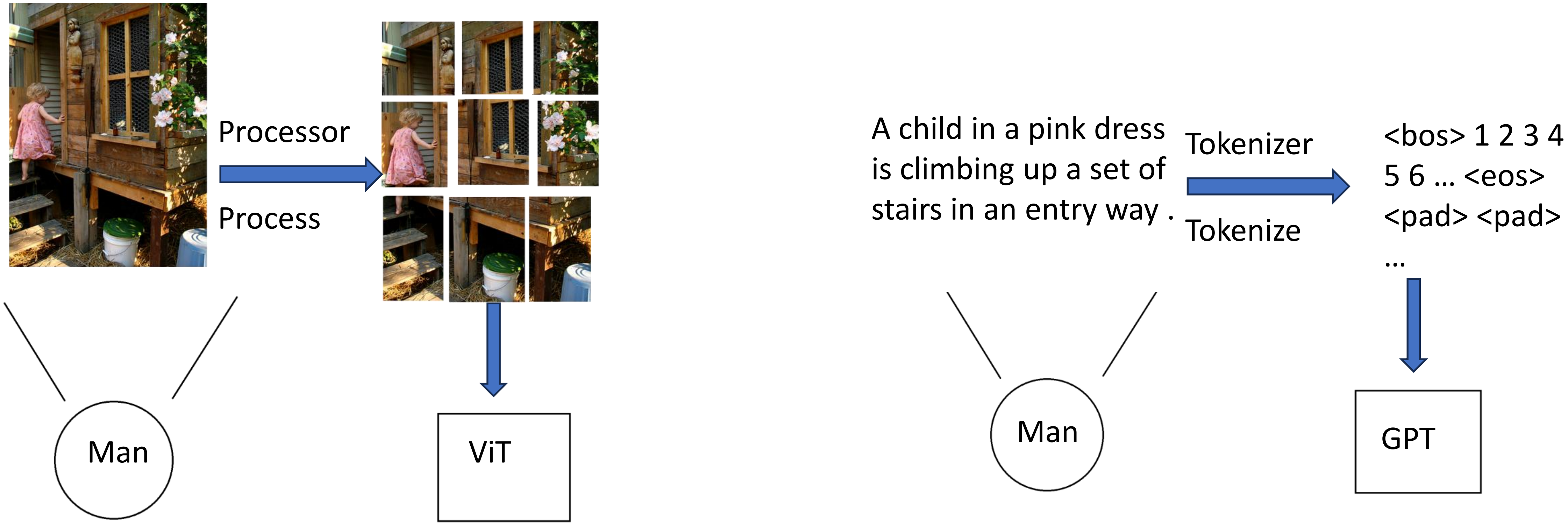
GPT2



<bos>

Tokens

- ViT reads image in batch, GPT reads words (and punctuations) in tokens



Tokens

- Usually positive numbers
- <cls>
- <pad> : Pad to unify the size of (maybe truncated) input output. Usually, this token should not be learned by the model. So, during training, this should be replaced with a random token that does not exist in the vocab.
- <eos>
- <sep>
- <bos>
- 1 sentence is around 30-40 tokens (English)

Masking

- Attention in transformers, visually explained | Chapter 6, Deep Learning (youtube.com)

Seq2SeqTrainer

- Trainer (huggingface.co)
- google/vit-base-patch16-224 · Hugging Face
- openai-community/gpt2 · Hugging Face
- OpenAI GPT2 (huggingface.co)
- Vision Encoder Decoder Models (huggingface.co)