ML Final Project

December 15, 2023

1 Header material

Paper Title: MaxViT: Multi-Axis Vision Transformer

Name: Yihao Wang

NetID: yw7486

Meta data: [Paper] [arXiv] [GitHub] [Blog]

Citation: Z. Tu, H. Talebi, H. Zhang, F. Yang, P. Milanfar, A. Bovik, and Y. Li, "Maxvit: Multi-axis vision transformer," in *ECCV*, pp. 459-479, 2022.

> Due to some layout issue, such as image size and hyperlink, it is highly recommended to view this report in Colab.

2 Overview of Topic

In this section we are going to introduce the most related topics about MaxViT, including:

- MaxViT and its characteristic Multi-Axes Attention technique,
- Other techniques used in MaxViT, such as MBConv, SE module, and Relative Attention,
- Hybrid Vision Models (a superset of MaxViT, as well as a class of visual models that has become quite popular recently)

> To keep the report neat, clear, organized, and focused, I arrange some necessary background on Vision Transformer and Convlutional Neural Networks in an independent section, named Prerequisite Knowledge.

```
[ ]: #@title environment preparation

     !pip install einops
```

```
Requirement already satisfied: einops in /usr/local/lib/python3.10/dist-packages
(0.7.0)
```

2.1 MaxViT

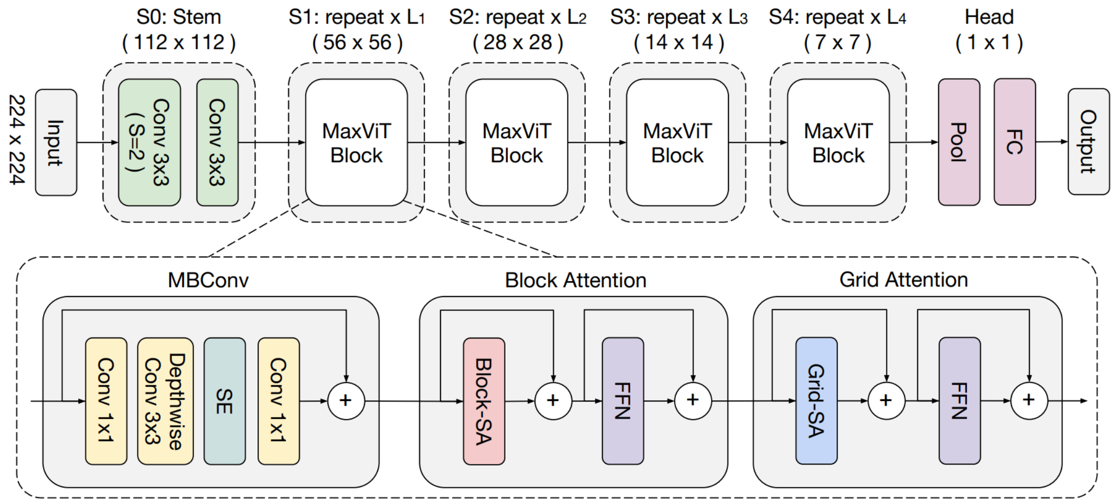
MaxViT [1]., short for Multi-Axis Vision Transformer, is a family of hybrid vision models, that achieves better performances across the board for both parameter and FLOPs efficiency than both state-of-the-art (SOTA) Convolutional Neural Networks (CNNs) [2] and Vision Transformers (ViTs) [3].

“Hybrid” here indicates a combination of CNN and ViT. People are designing hybrid architectures to combine the advantages from both CNNs and ViTs, hopefully. MaxViT has been evaluated on variety of benchmark, such as

- ImageNet [4] for Classification and (unconditional) Image Generation using GAN [5],
- COCO [6] for Object Detection and Instance Segmentation,
- AVA [7] for Image Aesthetic Assessment.

These comprehensive assessments show a great scalibility and generalization abilities of MaxViT, demonstrating the superior potential of MaxViT as a universal vision backbone.

Below is the illustration of MaxViT’s meta-architecture from MaxViT main paper(also in MaxViT blog):



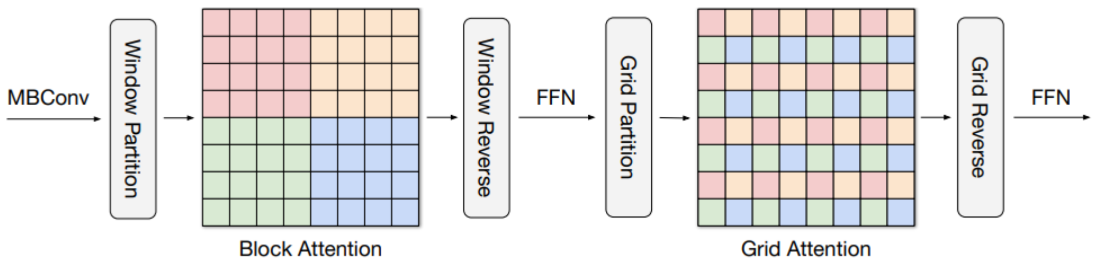
The key conponent leading to MaxViT’s success is a novel attention module called **m**ulti-**ax**is **s**elf-**a**ttention (Max-SA). Compared to full self-attention, Max-SA enjoys greater flexibility and efficiency:

- Max-SA allows to perform both local and global spatial interactions in a single block. As a comparison, both CNNs and standard self-attention are only either local[1] [1] or global.
- Max-SA naturally adapts to different input lengths with merely linear complexity. As a comparison, standard full attention requires quadratic complexity.

2.1.1 Multi-axis Attention

Below is the illustration from MaxViT main paper, demonstrating how Max-SA works. Note the same colors are spatially mixed by the self-attention operation.

[1]Here we only talk about traditional and prevalent small-kernel convolutions, typically 3×3 or 5×5. There has also been a recent surge of works that leverage super large kernels (like 31×31) to achieve a global receptive field in even shallow layers of CNNs. Representative papers are RepLKNet [8] and it’s continuation UniRepLKNet [9].



Note in this sketch, both the block size and gird size are set as 4.

Basicaly, Max-SA is sort of a "decomposation" of stadard full self-attention. There are two fundamental components:

- *block attention*, which is local and dense,
- *grid attention*, which is global and sparse.

This decomposation could be regarded as a proper regularization of full attention, which reintroduces some inductive bias for image data.

Block Attention The window partition operation[2] can be explained with the following tensor shape sequence:

$$\text{Window Partition:} \quad (H, W, C) \rightarrow \left(\frac{H}{P} \times P, \frac{W}{P} \times P, C\right) \rightarrow \left(\frac{HW}{P^2}, P^2, C\right)$$

Below is a plain block attention layer implemented in PyTorch. Note that to focus only on block attention itself, the implementation of relative self-attention, layer normalization, and MLP have been omitted and repalced with a shape-maintaining `nn.Identity`.

```
[ ]: #@title Block Attention implementation

     #@markdown Note this implementation is adapted from the pseudocode in Algo.1 in
      ↪MaxViT paper, with totally unchange logic.

     import torch
     from torch import nn, Tensor
     from einops import rearrange

     class BlockAttn(nn.Module):
         def __init__(self, block_size: int = 7) -> None:
             super().__init__()

             self.block_size = block_size

             self.ln = nn.Identity()
             self.relative_attn = nn.Identity()
             self.ffn = nn.Identity()
```

[2]This operation has two names in the text part —— "window partition" and "block partition". To avoid ambiguity and confusion, I will only use the term of window partition, which is consistent to the name in the figure.

```python
    def forward(self, inp: Tensor) -> Tensor:
        _, H, W = inp.shape[:3]
        x = self.ln(inp)
        x = self._window_partition(x, self.block_size)
        x = self.relative_attn(x)
        x = self._window_reverse(x, H//self.block_size, self.block_size)
        x = inp + x

        x = x + self.ffn(x)
        return x

    @staticmethod
    def _window_partition(
        inp: Tensor,
        block_size: int) -> Tensor:
        return rearrange(inp, "b (h y) (w x) c -> b (h w) (y x) c",
                         y=block_size, x=block_size)

    @staticmethod
    def _window_reverse(
        inp: Tensor,
        block_num: int,
        block_size: int) -> Tensor:
        return rearrange(inp, "b (h w) (y x) c -> b (h y) (w x) c",
                         h=block_num, y=block_size,
                         w=block_num, x=block_size)

inp = torch.randn(1, 56, 56, 4)
block_size = 7
block_num = inp.shape[1] // block_size
print(f'block_num = {block_num}, block_size = {block_size}\n')

print(f'Before window partition:{inp.shape}')
blocks = BlockAttn._window_partition(inp, block_size)
print(f'After window partition:\t{blocks.shape}')
out = BlockAttn._window_reverse(blocks, block_num, block_size)
print(f'After window reverse:\t{out.shape}')


block_attn = BlockAttn()
print(f'\nBefore Block Attention:\t{inp.shape}')
out = block_attn(inp)
print(f'After Block Attention:\t{out.shape}')
```

```
block_num = 8, block_size = 7
```

```
Before window partition:torch.Size([1, 56, 56, 4])
After window partition: torch.Size([1, 64, 49, 4])
After window reverse:   torch.Size([1, 56, 56, 4])

Before Block Attention: torch.Size([1, 56, 56, 4])
After Block Attention:  torch.Size([1, 56, 56, 4])
```

Grid Attention The grid partition operation can be explained with the following tensor shape sequence:

$$\text{Grid Partition:}\quad (H, W, C) \to (G \times \frac{H}{G}, G \times \frac{W}{G}, C) \to \left(G^2, \frac{HW}{G^2}, C\right) \to \left(\frac{HW}{G^2}, G^2, C\right)$$

Note the last $\to$ indicates a `swapaxes` operation that exchange the grid dimension(G^2) with the spatial diemnsion($\frac{HW}{G^2}$). Thus the attention machanism can be applied to those sparse grid, which brings up the global interaction.

The witty thing here is that, since the grid size G is fixed(7 by default), each lattice can have an *adaptive size* of $\frac{H}{G} \times \frac{W}{G}$. Therefore, whatever the input size is, the entire image will always be partitioned with a uniform 7×7 grid, and those pixels on the same relative position within the grid, i.e. with the same color, will interact with each other. This introduce a great property —— MaxViT can process multi-scale images inherently. his property will be proven in the code section.

```
[ ]: #@title Grid Attention implemention

     #@markdown Note this implementation is adapted from the pseudocode(Algo.1) in␣
      ↪MaxViT paper, with totally unchanged logic.

     #@markdown (This implementation is also used in [torchvision](https://github.
      ↪com/pytorch/vision/blob/v0.16.1/torchvision/models/maxvit.py#L333-L336) [10].
      ↪ Mostly for code reuse, I suppose.)

     import torch
     from torch import nn, Tensor
     from einops import rearrange

     class GridAttn(BlockAttn):
         def __init__(self, grid_size: int = 7) -> None:
             super().__init__()

             self.grid_size = grid_size

             self.ln = nn.Identity()
             self.relative_attn = nn.Identity()
             self.ffn = nn.Identity()

         def forward(self, inp: Tensor) -> Tensor:
             _, H, W = inp.shape[:3]
```

```python
        x = self.ln(inp)
        x = self._grid_partition(x, H//self.grid_size)
        x = self.relative_attn(x)
        x = self._grid_reverse(x, self.grid_size, H//self.grid_size)
        x = inp + x

        x = x + self.ffn(x)
        return x

    @staticmethod
    def _grid_partition(
        inp: Tensor,
        grid_num: int) -> Tensor:
        x = GridAttn._window_partition(inp, grid_num)
        x = torch.swapaxes(x, -2, -3)
        return x

    @staticmethod
    def _grid_reverse(
        inp: Tensor,
        grid_size: int,
        grid_num: int) -> Tensor:
        x = torch.swapaxes(inp, -2, -3)
        x = GridAttn._window_reverse(x, grid_size, grid_num)
        return x

inp = torch.randn(1, 56, 56, 4)
grid_size = 7
grid_num = inp.shape[1] // grid_size
print(f'grid_num = {grid_num}, grid_size = {grid_size}\n')

print(f'Before grid partition:\t{inp.shape}')
grids = GridAttn._grid_partition(inp, grid_num)
print(f'After grid partition:\t{grids.shape}')
out = GridAttn._grid_reverse(grids, grid_size, grid_num)
print(f'After grid reverse:\t{out.shape}')


grid_attn = GridAttn()
print(f'\nBefore Grid Attention:\t{inp.shape}')
out = grid_attn(inp)
print(f'After Grid Attention: \t{out.shape}')
```

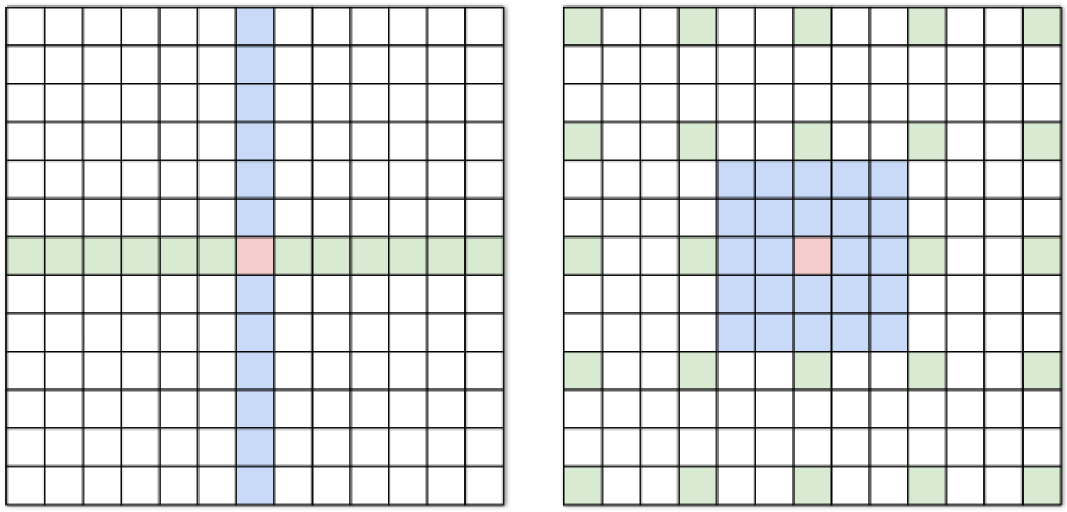
```
grid_num = 8, grid_size = 7

Before grid partition:  torch.Size([1, 56, 56, 4])
After grid partition:   torch.Size([1, 64, 49, 4])
After grid reverse:     torch.Size([1, 56, 56, 4])

Before Grid Attention:  torch.Size([1, 56, 56, 4])
After Grid Attention:   torch.Size([1, 56, 56, 4])
```

[ ]:
```python
#@title Another Grid Attention implementation with einops

#@markdown Note in this implementation `swapaxes` is reduced into `rearrange`.

#@markdown (A similar implementation logic is also used in [timm](https://
↪github.com/huggingface/pytorch-image-models/blob/v0.9.12/timm/models/maxxvit.
↪py#L632-L665) [11]. Mostly for logic clearity, I suppose.)

import torch
from torch import nn, Tensor
from einops import rearrange


class GridAttnAlt(GridAttn):
    def __init__(self, grid_size: int = 7) -> None:
        super().__init__(grid_size)

    @staticmethod
    def _grid_partition(
        inp: Tensor,
        grid_num: int) -> Tensor:
        return rearrange(inp, "b (h y) (w x) c -> b (y x) (h w) c",
                         y=grid_num, x=grid_num)

    @staticmethod
    def _grid_reverse(
        inp: Tensor,
        grid_size: int,
        grid_num: int) -> Tensor:
        return rearrange(inp, "b (y x) (h w) c -> b (h y) (w x) c",
                         h=grid_size, y=grid_num,
                         w=grid_size, x=grid_num)

inp = torch.randn(1, 56, 56, 4)
grid_size = 7
grid_num = inp.shape[1] // grid_size
print(f'grid_num = {grid_num}, grid_size = {grid_size}\n')

print(f'Before grid partition:\t{inp.shape}')
grids = GridAttnAlt._grid_partition(inp, grid_num)
print(f'After grid partition:\t{grids.shape}')
out = GridAttnAlt._grid_reverse(grids, grid_size, grid_num)
```

```
print(f'After grid reverse:\t{out.shape}')


grid_attn_alt = GridAttnAlt()
print(f'\nBefore Grid Attention:\t{inp.shape}')
out = grid_attn_alt(inp)
print(f'After Grid Attention: \t{out.shape}')
```

```
grid_num = 8, grid_size = 7

Before grid partition:  torch.Size([1, 56, 56, 4])
After grid partition:   torch.Size([1, 64, 49, 4])
After grid reverse:     torch.Size([1, 56, 56, 4])

Before Grid Attention:  torch.Size([1, 56, 56, 4])
After Grid Attention:   torch.Size([1, 56, 56, 4])
```

It's important to note that Max-SA is completely different from the axial attention [12]. Axial attention achieves a global receptive field with $O(N\sqrt{N})$ complexity by applying column-wise attention then row-wise. On the contrary, Max-SA shown here first employs dense local attention, then sparse global attention, enjoying global receptive fields with only *linear* ($O(N)$) complexity. See figure below for clearity.


(a) Axial Attention (b) Multi-Axis Attention

2.1.2 Other techniques used in MaxViT

MBConv and SE module MBConv

MBConv[3], short for "**M**obile inverted **B**ottleneck **Conv**olution", is the basic convolutional block in

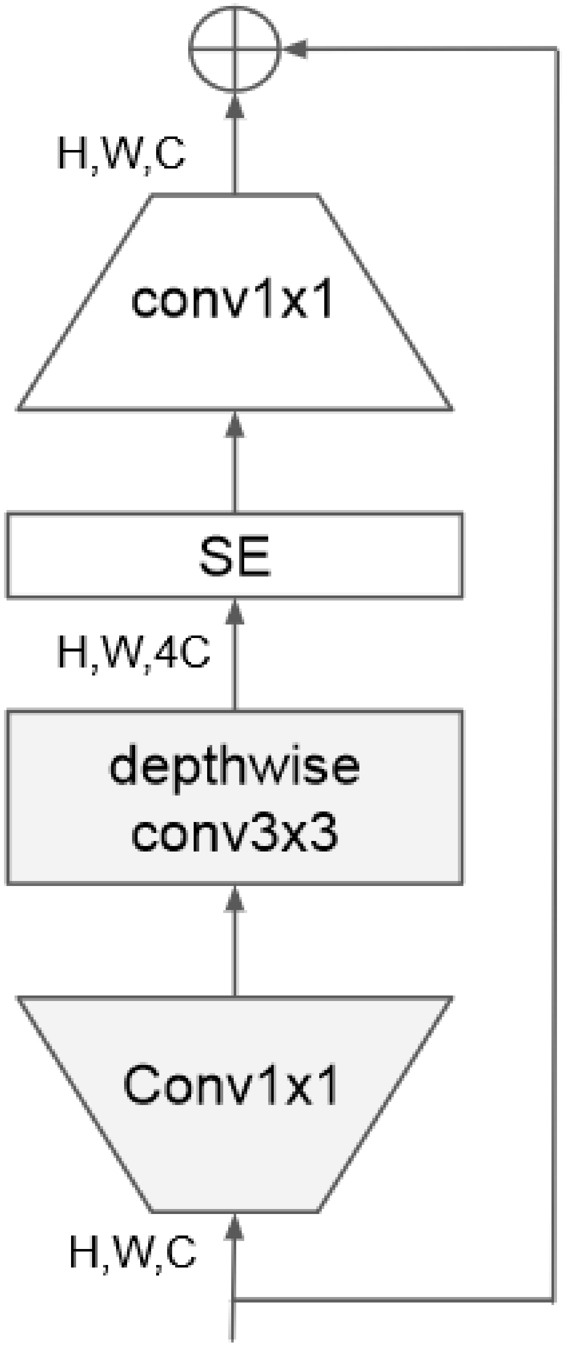
[3]Interestingly, althought the architecture of MBConv was first proposed in MobileNetV2, it was only in EfficientNet that it was first referred to as "MBConv".

MaxViT architecture. A large amount of empirical experience[4] has shown that it's a greate choice to use MBConv if one wants a balance of efficiency[5] and performance.

Note the MBConv block used in MaxViT is the version from EfficientNetV2 [15] (with SE module), rather than the original version from MobileNetV2 [13]. Below is a illustration of MBConv block(figure from EfficientNetV2 paper [15])

[4]Such as EfficientNet [14], EfficientNetV2 [15], ConvNeXt [16], MaxViT [1], RT-1 [17], and EfficientViT [18].

[5]"Efficiency" here is a collective term for memory utilization, computational cost, and inference speed.



MBConv

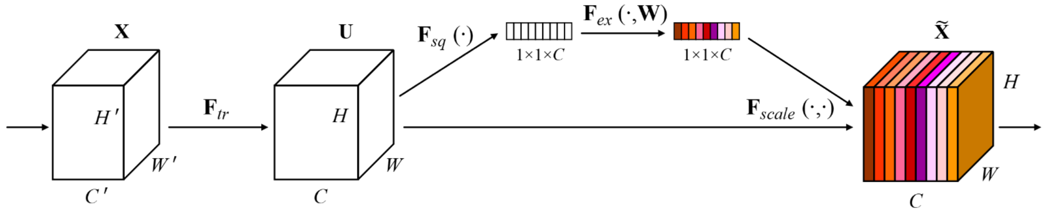
SE

SE module, short for "**S**queeze-and-**E**xcitation" module, was proposed in SENet [19]. As its name, SE module adaptively recalibrates features in all channels via:

1. "squeeze" all channels into channel-wise statistics with global averge pooling, and
2. "excite" original feature maps according to adaptively transformed channel-wise statistics.

SE module could be a plug-and-play module for any single convolutional layer, as long as it doesn't change the number of channels. It "excites" performance without introducing unbearable additional cost.

Below is a illustration of SE module(figure from SENet paper [19]):



Note that in MaxViT, MBConv plays another important role as conditional position encoding (CPE) [20]. Lying in front of attention blocks, MBConv makes MaxViT model free of explicit positional encoding layers as in classic ViTs.

[ ]:
```python
#@title SE implementation

import torch
from torch import nn, Tensor
from einops.layers.torch import Rearrange, Reduce

class SE(nn.Module):
    def __init__(self, dim: int, shrinkage_rate: float = 0.25) -> None:
        super().__init__()
        hidden_dim = int(dim * shrinkage_rate)

        self.gate = nn.Sequential(
            Reduce('b c h w -> b c', 'mean'),
            nn.Linear(dim, hidden_dim, bias = False),
            nn.SiLU(),
            nn.Linear(hidden_dim, dim, bias = False),
            nn.Sigmoid(),
            Rearrange('b c -> b c 1 1')
        )

    def forward(self, x: Tensor) -> Tensor:
        return x * self.gate(x)

inp = torch.randn(1, 4, 56, 56)

se = SE(dim=inp.shape[1])

print(f'Before SE:\t{inp.shape}')
```

```
out = se(inp)
print(f'After SE:\t{out.shape}')
```

```
Before SE:      torch.Size([1, 4, 56, 56])
After SE:       torch.Size([1, 4, 56, 56])
```

[ ]:
```
#@title MBConv implementation

from typing import Callable

import torch
from torch import nn, Tensor

class MBConv(nn.Module):
    def __init__(self, in_dim: int, out_dim: int,
                 expansion_rate: float = 4, shrinkage_rate: float = 0.25,
                 act: Callable = nn.GELU) -> None:
        super().__init__()

        hidden_dim = int(expansion_rate * out_dim)

        self.block = nn.Sequential(
            nn.Conv2d(in_dim, hidden_dim, 1),
            nn.BatchNorm2d(hidden_dim),
            act(),
            nn.Conv2d(hidden_dim, hidden_dim, 3, padding=1, groups=hidden_dim),␣
 ↪ # DWConv
            nn.BatchNorm2d(hidden_dim),
            act(),
            SE(hidden_dim, shrinkage_rate),
            nn.Conv2d(hidden_dim, out_dim, 1),  # PWConv
            nn.BatchNorm2d(out_dim)
        )

    def forward(self, x: Tensor) -> Tensor:
        return x + self.block(x)


inp = torch.randn(1, 4, 56, 56)

mb = MBConv(in_dim=inp.shape[1], out_dim=inp.shape[1])

print(f'Before MBConv:\t{inp.shape}')
out = mb(inp)
print(f'After MBConv:\t{out.shape}')
```

```
Before MBConv:  torch.Size([1, 4, 56, 56])
```

```
After MBConv:   torch.Size([1, 4, 56, 56])
```

Relative Attention In MaxViT, all the attention operators use the relative attention as in CoAtNet [21] and Swin [22], a variant of vanilla attention characterized by its learnable relative attention bias. Relative attention can be defined as:

$$\text{RelativeAttention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q} \cdot \mathbf{K}^T}{\sqrt{d}} + \mathbf{B}\right) \cdot \mathbf{V},$$

where $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ are the query, key, and value and d is the embedding dimension. Except for the scaled dot-product result $\mathbf{Q} \cdot \mathbf{K}^T / \sqrt{d}$, the attention weights are co-decided by a location-aware matrix $\mathbf{B}$. Since the relative position along each axis lies in the range $[-H+1, H-1]$ (or $[-W+1, W-1]$), $\mathbf{B}$ can be parameterized with a learnable bias "codebook" $\hat{\mathbf{B}} \in \mathbb{R}^{(2H-1)\times(2W-1)}$, and values in $\mathbf{B}$ can be taken from $\hat{\mathbf{B}}$ during inference.

```
[ ]: #@title Relative Attention implementation

     #@markdown This implementation is adapted from [vit_pytorch](https://github.com/
      ↪lucidrains/vit-pytorch/blob/1.6.4/vit_pytorch/max_vit.py#L148) [23].

     import torch
     from torch import nn, Tensor
     from einops import rearrange

     class RelativeAttention(nn.Module):
         def __init__(self, n_heads: int, window_size: int) -> None:
             super().__init__()

             # trainable attention bias
             self.rel_pos_bias = nn.Embedding((2 * window_size - 1) ** 2, n_heads)

             pos = torch.arange(window_size)
             grid = torch.stack(torch.meshgrid(pos, pos, indexing = 'ij'))
             grid = rearrange(grid, 'c i j -> (i j) c')
             rel_pos = rearrange(grid, 'i ... -> i 1 ...') - rearrange(grid, 'j ...␣
      ↪-> 1 j ...')
             rel_pos += window_size - 1
             rel_pos_indices = (rel_pos * torch.tensor([2 * window_size - 1, 1])).
      ↪sum(dim = -1)

             # untrainable indeces of attention bias
             self.register_buffer('rel_pos_indices', rel_pos_indices)

         def forward(self, x: Tensor) -> Tensor:
             bias = self.rel_pos_bias(self.rel_pos_indices)
             return x + rearrange(bias, 'i j h -> 1 h i j')
```

```
n_heads = 8
window_size = 7
seq_len = window_size ** 2
scaled_dot_product_result = torch.randn(1, n_heads, seq_len, seq_len)

rel_attn = RelativeAttention(n_heads=8, window_size=7)

print(f'Before Relative Attn:\t{scaled_dot_product_result.shape}')
out = rel_attn(scaled_dot_product_result)
print(f'After Relative Attn:\t{out.shape}')
```

```
Before Relative Attn:   torch.Size([1, 8, 49, 49])
After Relative Attn:    torch.Size([1, 8, 49, 49])
```

2.2 Hybrid Vision Models

2.2.1 Why Hybrid?

For the past decade, CNNs have been the dominating model architecture for almost all vision tasks. Thanks to their strong prior[6], convolutional layers tend to have better generalization with faster converging speed (especially on relatively small dataset).

More recently, ViTs have shown great success on computer vision tasks due to their great scalability[7]. Unlike convolutiona layers, the self-attention layers in vision transformers provide a global context by modeling long-range dependencies(see section of Vision Transformer). Unfortunately, this global scope often comes at a high computational price. What's more, vanilla Transformer layers may lack certain desirable inductive biases possessed by CNNs, and thus require significant amount of data and computational resource to compensate [21].

In short (but not perfectly accurately), CNN is a synonym of efficiency while ViT is that of performance. A key challenge here is how to effectively combine convolution and attention to achieve better trade-offs between efficiency and performance. There are two typical roadmaps to achieve "hybrid":

- One is to implicitly exploit the idea behind convolution. Swin Transformer [22] is one of the representative works of this type.
- The other one is to explicitly use convolutional layers directly, just like MaxViT [1]. As in the next section, a variety of works will deliver their own answers.

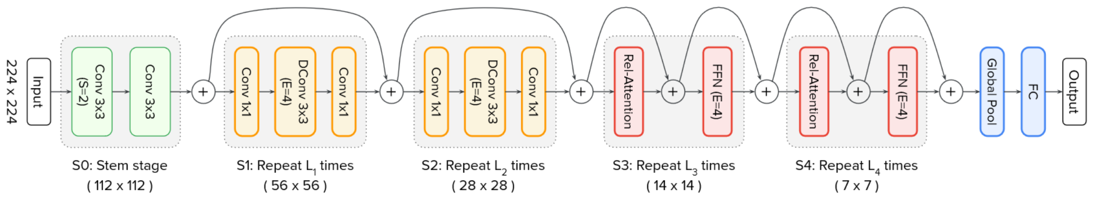
2.2.2 Other Hybrid Models

CoAtNet [21] (NeurIPS '21)

This work is the first to take a systematic approach to the vertical layout design and show how and why different network stages prefer different types of layers. This remarkable work by Google Brain paved the way for almost all subsequent research on hybrid architectures.

[6]The convolution operation introduces some inductive biases natually, such as *Translation Equivariance* and *Locality*.

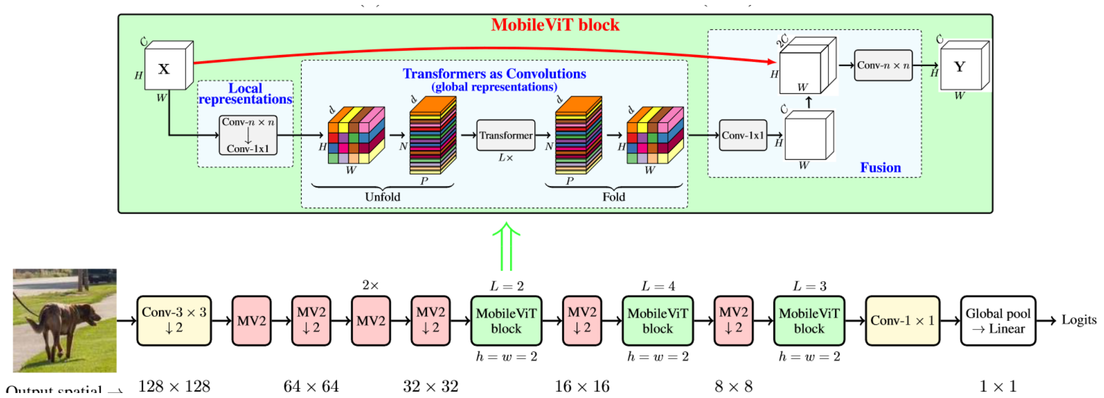
[7]As pointed out in the ConvNeXt paper, larger datasets and modern training recipe (e.g., AdamW optimizer, data augmentation techniques, and regularization schemes) also play a role.



224 x 224
Input
Conv 3x3 (S=2)
Conv 3x3
Conv 1x1
DConv 3x3 (E=4)
Conv 1x1
Rel-Attention
FFN (E=4)
Global Pool
FC
Output
S0: Stem stage (112 x 112)
S1: Repeat L1 times (56 x 56)
S2: Repeat L2 times (28 x 28)
S3: Repeat L3 times (14 x 14)
S4: Repeat L4 times (7 x 7)

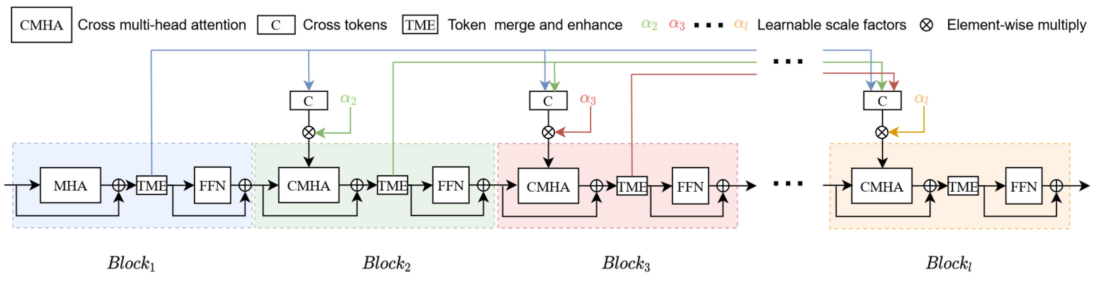
MobileViT [25] (ICLR '22)

In this work Apple aims to design a light-weight, general-purpose, and low latency network for mobile vision tasks. They presents a different perspective to learn global representations via replacing the local processing (using convolution) with global processing (using Transformer).



FcaFormer [26] (ICCV '23)

In this work the authors proposed to aggregate tokens from previous blocks with DWConv and Cross-Attention. This operation calibrates the statistics of tokens from different semantic levels, thus can better fuse information from them.



CMHA Cross multi-head attention
C Cross tokens
TME Token merge and enhance
α2 α3 ... αl Learnable scale factors
⊗ Element-wise multiply
MHA
TME
FFN
CMHA
C
Block1
Block2
Block3
Blockl

EfficientViT [27] (CVPR '23)

To distinguish, note that this EfficientViT(CVPR) was proposed by Microsoft.

In this work the authors proposed to use DWConv before FFN, which serves as token interaction and introduces inductive bias of the local structural information. Moreover, they proposed Cascaded Group Attention – add the output of each head to the subsequent head to refine the feature representations progressively.

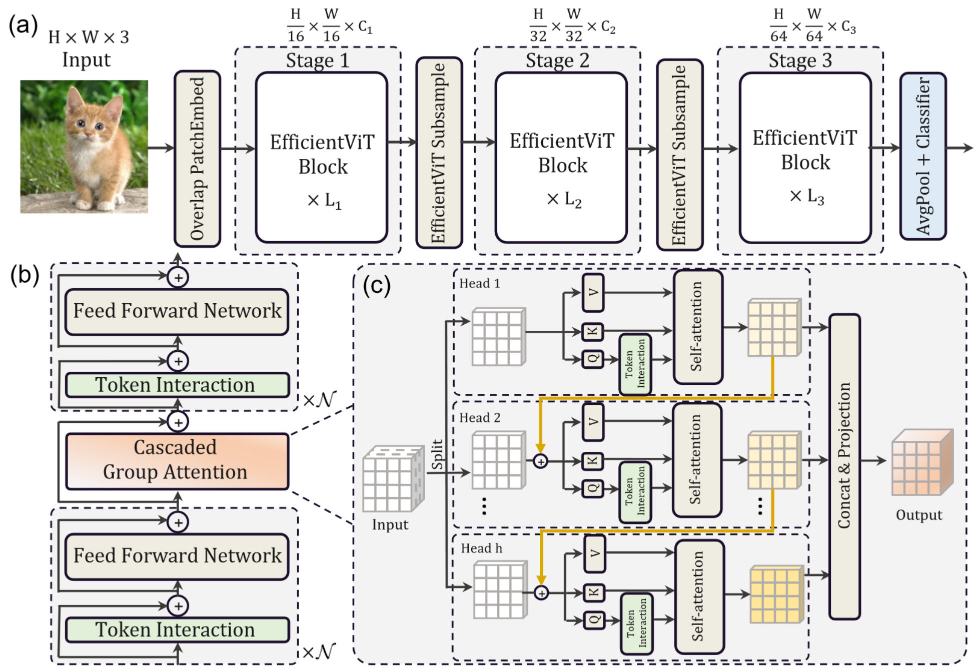
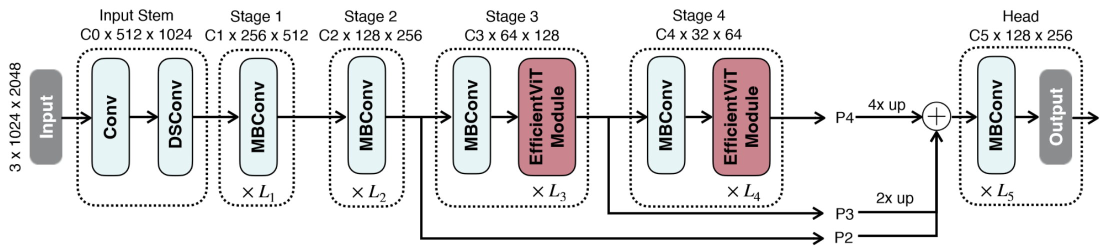


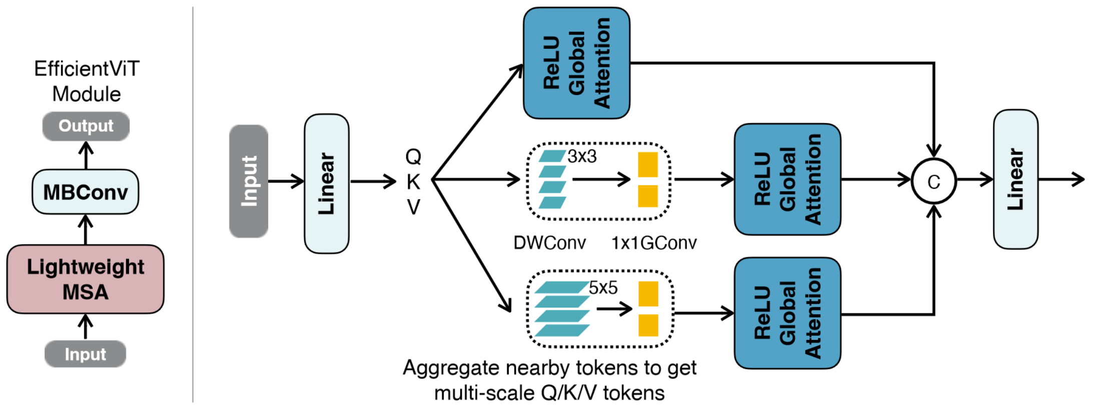
Figure 6. Overview of EfficientViT. (a) Architecture of EfficientViT; (b) Sandwich Layout block; (c) Cascaded Group Attention.

EfficientViT [18] (ICCV '23)

To distinguish, note that this EfficientViT(ICCV) was proposed by MIT.

In this work the author made an observation that large kernel convolution and SoftMax operation are hardware-unfriendly. Therefore, they suggested to generate multi-scale tokens with small-size kernel convolutions and replace traditional SoftMax-based attention with ReLU-based attention. These effort achieved impressive latency reduction.





FastViT [28] (ICCV '23)

In this work Apple introduced RepMixer, a novel token mixer with operation rearrangement to allow inference-time reparameterization to lower computational cost. They also use large kernel convolutions in train-time overparameterization[8] branch and Conv-FFN, which make the model more robust by benefiting from larger receptive field.

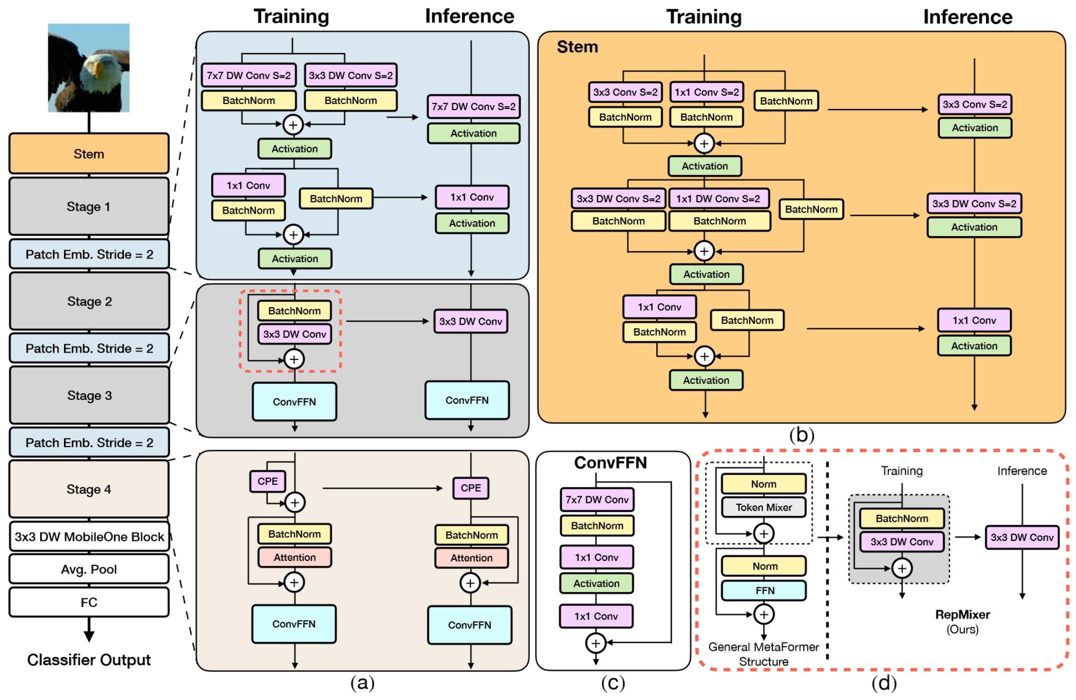


Figure 2: (a) Overview of FastViT architecture which decouples train-time and inference-time architecture. Stages 1, 2, and 3 have the same architecture and uses RepMixer for token mixing. In stage 4, self attention layers are used for token mixing. (b) Architecture of the convolutional stem. (c) Architecture of convolutional-FFN (d) Overview of *RepMixer* block, which reparameterizes a skip connection at inference.

[8]Although it is not mentioned in the paper, I suppose the spirit of "train-time overparameterization" and "inference-time reparameterization" is to increase variance, thus decrease bias, during training, while painlessly reduce computational overhead, with performance maintained, during inference.

3 Prerequisite Knowledge

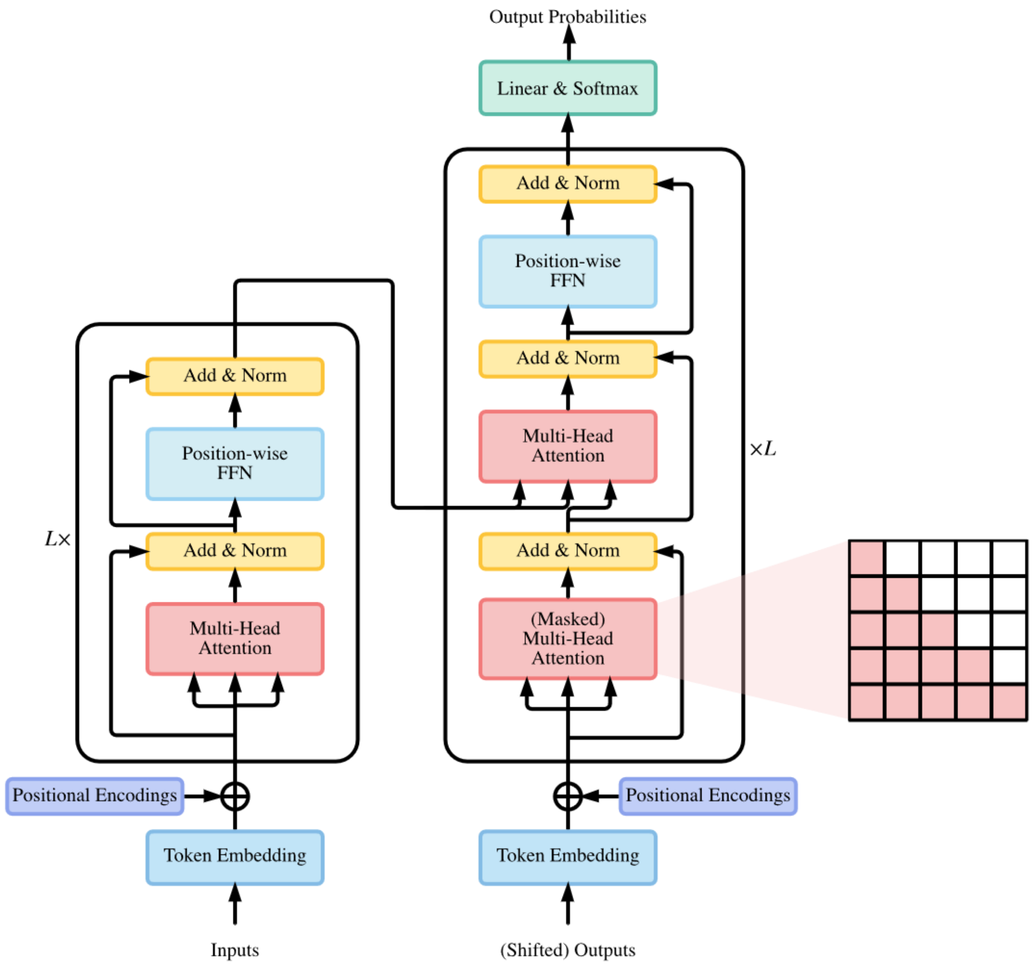
```
[ ]: #@title environment preparation

     !pip install einops
```

```
Requirement already satisfied: einops in /usr/local/lib/python3.10/dist-packages
(0.7.0)
```

3.1 (Vanilla) Transformer

Once proposed, Transformer [29] soonly became the model of choice in natural language processing (NLP), replacing the dominate Recurrent Neural Networks(RNNs) [30] representative by Long-Short Term Memery(LSTM) [31] and Gate Recurrent Unit(GRU) [32]. s scalability enable it to gain performance as data size scales up [33]. A plenty of well-known models are built on Transformer architecture, such as GPT [34–36] from OpenAI, BERT [37] from Google, and Llama [38,39] from Meta. The vanilla Transformer is a sequence-to-sequence model and consists of an encoder and a decoder, each of which is a stack of L identical blocks.

Below is the classic Transformer architecture for ”A Survey of Transformers” [40].



Next, we are introducing several key components of the vanilla Transformer one-by-one:

- Multi-Head Self-Attention(MHSA),
- Feed-Forward Network(FFN)
- Residual Connection [41,42] and Layer Normalization(LayerNorm or LN) [43], and
- Positional Encodings(PE)[9].

Then we will introduce the structures of encoder and decoder, as well as their difference.

3.1.1 From Scaled Dot-Product Attention to MHSA

Scaled Dot-Product Attention

Transformer adopts the scaled dot-product of Query-Key-Value (QKV) as the implementation of attention mechanism, which is given by

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q} \cdot \mathbf{K}^T}{\sqrt{D_k}}\right) \cdot \mathbf{V} = \mathbf{A} \cdot \mathbf{V},$$

[9]PE is also referred to as Position Embeddings.

where $\mathbf{Q} \in \mathbb{R}^{N \times D_k}$ is queries of length N, $\mathbf{K} \in \mathbb{R}^{M \times D_k}$ is keys of length M, and $\mathbf{V} \in \mathbb{R}^{M \times D_v}$ is values of length M. D_k and D_v are embedding dimensions of keys (or queries) and values respectively.

Here are some notes on QKV scaled dot-product attention:

- The softmax is applied in a row-wise manner, i.e., on the last dimension of a given tensor. - The intermediate result $\mathbf{A} = \text{softmax}\left(\frac{\mathbf{Q} \cdot \mathbf{K}^T}{\sqrt{D_k}}\right)$ is often called *attention matrix*, which describes how much attention each query pays to each key.
- The dot-products of queries and keys are divided by $\sqrt{D_k}$ to alleviate gradient vanishing problem of the softmax function.

```
[ ]: #@title Scaled Dot-Product Attention implemeatation
     #@markdown <a name="SDPA_impl"></a>

     #@markdown In this implementation, we use `torch.einsum` to calculate matrix␣
      ↪multiplication, which clearly demonstrates the process of how tensors change.

     #@markdown And to show the essence of the attention mechanism, we omit the␣
      ↪temperature parameter and dropout layer, which usually exist to add␣
      ↪regularization stabilize training.

     import torch
     from torch import nn, Tensor
     import torch.nn.functional as F

     class ScaledDotProductAttention(nn.Module):

         def __init__(self, temperature: float):
             super().__init__()

             self.temperature = temperature

         def forward(self, q: Tensor, k: Tensor, v: Tensor) -> Tensor:
             # q.shape: [..., n, k]
             # k.shape: [..., m, k]
             # v.shape: [..., m, v]

             # Q @ K.T / (d_k ** 0.5)
             attn = torch.einsum("... n k, ... m k -> ... n m", q, k)
             attn /= self.temperature
             # softmax
             attn = F.softmax(attn, dim=-1)
             # A @ V
             output = torch.einsum("... n m, ... m v -> ... n v", attn, v)

             return output, attn

     d_k = 8
```

```
d_v = 16

q = torch.randn(1, 10, d_k)
k = torch.randn(1, 11, d_k)
v = torch.randn(1, 11, d_v)

attention = ScaledDotProductAttention(temperature=d_k ** 0.5)
output, attn = attention(q, k, v)
print(f"Attention:\t{attn.shape}")
print(f"Output: \t{output.shape}")
```

```
Attention:      torch.Size([1, 10, 11])
Output:         torch.Size([1, 10, 16])
```

Multi-Head Attention

In order to allow the model jointly attend to information from different representation subspaces, Transformer adopts multi-head trick. Instead of directly applying attention on the output of the previous layer, we project the original queries, keys and values from the embedding space of dimension $D_{\{k,v\}}$ into h subspaces of dimension $d_{\{k,v\}}$ before applying the scaled dot-product attention on each of them:

$$\begin{aligned}\text{MultiHead}(\mathbf{Q},\mathbf{K},\mathbf{V}) &= \text{Concat}(\text{head}_1,\dots,\text{head}_h)\mathbf{W}^O \\ \text{where head}_i &= \text{Attention}(\mathbf{Q}\mathbf{W}_i^Q,\mathbf{K}\mathbf{W}_i^K,\mathbf{V}\mathbf{W}_i^V),\end{aligned}$$

where $\mathbf{W}_i^Q \in \mathbb{R}^{D_k\times d_k}, \mathbf{W}_i^K \in \mathbb{R}^{D_k\times d_k}, \mathbf{W}_i^V \in \mathbb{R}^{D_v\times d_v}$ are input projection matrices, and $\mathbf{W}^O \in \mathbb{R}^{hd_v\times D_v}$ is the output projection matrix.

Here are some notes on Multi-Head attention:

- In practice, $d_{\{k,v\}}$ is usually set to $D_{\{k,v\}}/h$, which preserves the total embedding dimension and has similar computational cost. D_k and D_v are often set to the same value D_m.
- When implementing, we can use linear layers to project QKV into spaces of dimension $h\cdot d_{\{k,v\}}$ and then split them along embedding dimension into h pieces.

[ ]:
```
#@title Multi-Head Attention implemeatation
#@markdown <a name="MHA_impl"></a>

#@markdown Thanks to the `...` syntax in `torch.einsum`, this implementation␣
↪can directly adopt the previous [Scaled Dot-Product Attention](#SDPA_impl),␣
↪whihc supports input with different `ndim`.

import torch
from torch import nn, Tensor
import torch.nn.functional as F

from einops import rearrange

class MultiHeadAttention(nn.Module):
```

```python
    def __init__(self, n_head: int, D_k: int, D_v:int, d_k: int, d_v: int):
        super().__init__()

        self.n_head = n_head

        self.w_q = nn.Linear(D_k, n_head * d_k, bias=False)
        self.w_k = nn.Linear(D_k, n_head * d_k, bias=False)
        self.w_v = nn.Linear(D_v, n_head * d_v, bias=False)
        self.w_o = nn.Linear(n_head * d_v, D_v, bias=False)

        self.attention = ScaledDotProductAttention(temperature=d_k ** 0.5)

    def forward(self, q: Tensor, k: Tensor, v: Tensor) -> Tensor:

        # project original q,k,v into different heads
        q = rearrange(self.w_q(q), "b n (h k) -> b h n k", h=self.n_head)
        k = rearrange(self.w_k(k), "b m (h k) -> b h m k", h=self.n_head)
        v = rearrange(self.w_v(v), "b m (h v) -> b h m v", h=self.n_head)

        # apply attention to each head seperately
        output, attn = self.attention(q, k, v)

        # concat all heads back together
        output = rearrange(output, "b h n v -> b n (h v)")
        output = self.w_o(output)

        return output, attn


n_head = 8
D_k = 128
D_v = 256
d_k = 32
d_v = 64

q = torch.randn(1, 10, D_k)
k = torch.randn(1, 11, D_k)
v = torch.randn(1, 11, D_v)

multi_head_attention = MultiHeadAttention(n_head, D_k, D_v, d_k, d_v)
output, attn = multi_head_attention(q, k, v)
print(f"Attention:\t{attn.shape}")
print(f"Output: \t{output.shape}")
```

```
Attention:      torch.Size([1, 8, 10, 11])
Output:         torch.Size([1, 10, 256])
```

Self-Attention

Among all attention mechanisms, self-attention is characterized of "applying attention to itself". Given a sequence of token embeddings $\mathbf{X} \in \mathbb{R}^{N \times D}$, we set $\mathbf{Q} = \mathbf{K} = \mathbf{V} = \mathbf{X}$ when applying self-attention. Combined with Multi-head technique, we can also regard those projection matrices $\mathbf{W}_i^{\{Q,K,V\}}$ as projection from data(or feature) space to QKV spaces.

[ ]:
```
#@title Multi-Head Self-Attention implemeatation

import torch
from torch import Tensor


class MultiHeadSelfAttention(MultiHeadAttention):

    def __init__(self, n_head: int, d_model: int, d_kv: int):
        super().__init__(n_head, d_model, d_model, d_kv, d_kv)

    def forward(self, x: Tensor) -> Tensor:
        return super().forward(q=x, k=x, v=x)


n_head = 8
D_model = 512
d_kv = 64

x = torch.randn(1, 12, D_model)

multi_head_self_attention = MultiHeadSelfAttention(n_head, D_model, d_kv)
output, attn = multi_head_self_attention(x)
print(f"Attention:\t{attn.shape}")
print(f"Output: \t{output.shape}")
```

```
Attention:      torch.Size([1, 8, 12, 12])
Output:         torch.Size([1, 12, 512])
```

3.1.2 FFN

FFN is simply a two-layer MLP, which is given by

$$\mathrm{FFN}(\mathbf{X}) = \mathrm{ReLU}\left(\mathbf{X}\mathbf{W}^1 + \mathbf{b}^1\right)\mathbf{W}^2 + \mathbf{b}^2,$$

where $\mathbf{W}^1 \in \mathbb{R}^{D_m \times D_f}, \mathbf{W}^2 \in \mathbb{R}^{D_f \times D_m}, \mathbf{b}^1 \in \mathbb{R}^{D_f}, \mathbf{b}^2 \in \mathbb{R}^{D_m}$ are weights and biases of MLP.

Here are some notes on FFN:

- Typically the intermediate dimension D_f of the FFN is set to be larger than D_m, e.g., $D_f = 4D_m$[10].

[10]This setting shares a similar insights of inverted bottleneck structure in CNNs.

- Since FFN operates independently and identically on each spatial position, it is also known as *position-wise* FFN. And that's exactly why FFN could be understood as two point-wise(1×1) convolutions[11].

3.1.3 Residual Connection and LayerNorm

In order to build and train a deep model effectively, Transformer employs a residual connection around each module, followed by LayerNorm[12], which can be written as

$$\mathbf{X} = \text{LayerNorm}(\text{MHSA}(\mathbf{X}) + \mathbf{X})$$
$$\mathbf{X} = \text{LayerNorm}(\text{FFN}(\mathbf{X}) + \mathbf{X}).$$

3.1.4 Positional Encodings

Unlike CNNs or RNNs, in Transformer there is no information about position in any operation till now. Therefore, we need to manually inject some information about the relative or absolute position of the tokens in the sequence. Transformer choose to use sine and cosine functions of different frequencies:

$$\text{PE}_{(pos,2i)} = \sin(pos/10000^{2i/D_m})$$
$$\text{PE}_{(pos,2i+1)} = \cos(pos/10000^{2i/D_m}),$$

where pos is the position, i is the dimension, and D_m is the embedding dimension through the model. This way, Transformer uses a geometric wavelength to introduce additional positional information to the input embeddings.

[ ]:
```
#@title Positional Encoding implemeatation

import torch
from torch import nn, Tensor

class PositionalEncoding(nn.Module):

    def __init__(self, d_model: int, max_len: int = 1000):
        super().__init__()

        position = torch.arange(max_len).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2) * (-torch.log(torch.
↪tensor(10000.0)) / d_model))
        pe = torch.zeros(1, max_len, d_model)
        pe[..., 0::2] = torch.sin(position * div_term)
        pe[..., 1::2] = torch.cos(position * div_term)
        self.register_buffer('pe', pe)
```

[11]Now it is natural to extent FFN into Conv-FFN, e.g., in FastViT [28].

[12]Note this way that vanilla Transformer uses LayerNorm behind attention module is known as post-norm. There is another popular way of applying LayerNorm ahead of attention module, called pre-norm [44].

```
    def forward(self, x: Tensor) -> Tensor:
        # x.shape: (batch_size, seq_len, embed_dim)
        return x + self.pe[:, :x.shape[1]]

d_model = 256
seq_len = 10

seq = torch.randn(1, seq_len, d_model)
pe = PositionalEncoding(d_model)
output = pe(seq)

print(f"PE:\t{pe.pe.shape}")
print(f"Input:\t{seq.shape}")
print(f"Output:\t{output.shape}")
```

```
PE:     torch.Size([1, 1000, 256])
Input:  torch.Size([1, 10, 256])
Output: torch.Size([1, 10, 256])
```

3.1.5 Encoder vs. Decoder

Encoder

Each encoder block is mainly composed of a MHSA module and a FFN. For building a deeper model, a residual connection is employed around each module, followed by LayerNorm.

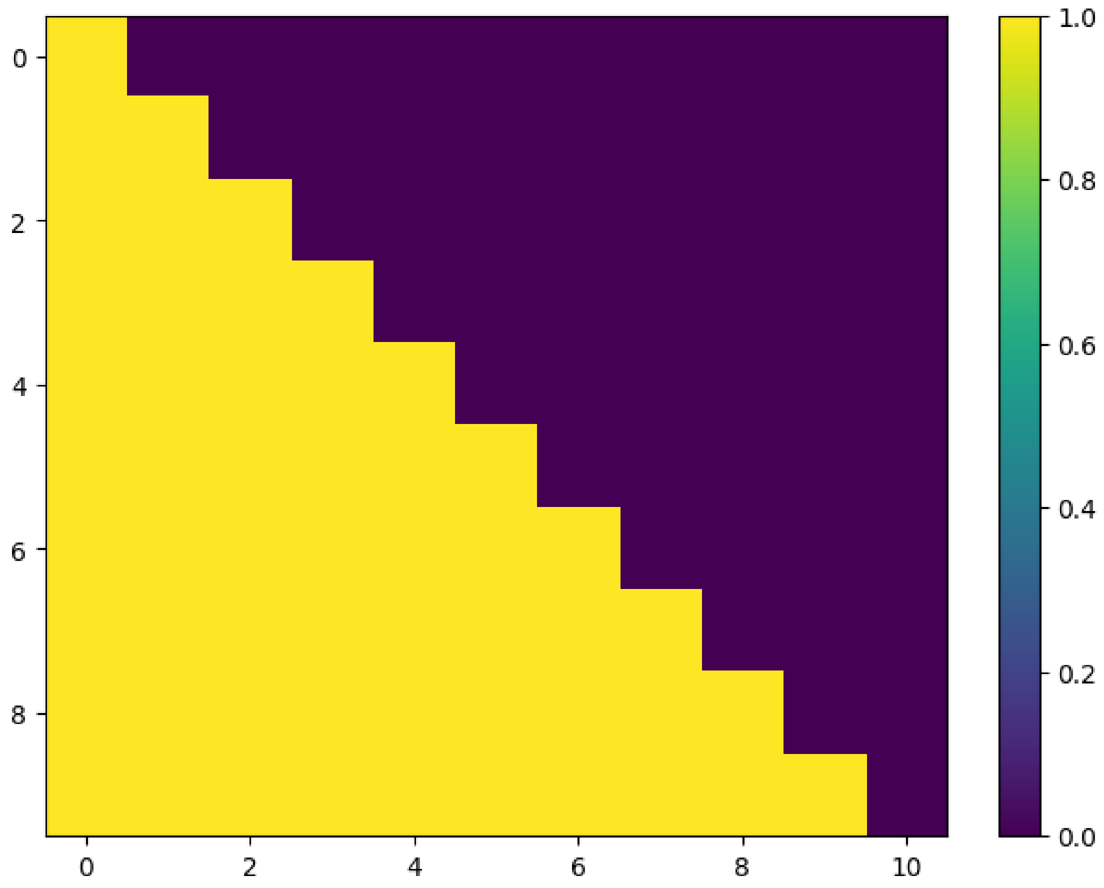
Decoder

Compared to the encoder blocks, things become a little bit different in decoder blocks. Between the MHSA modules and the FFNs, decoder blocks additionally insert MHSA modules of cross-attention version. In these modules, Q-s are projected from the outputs of the previous (decoder) layer, whereas the K-s and V-s are projected using the outputs of the encoder. Furthermore, considering the causal character of language, the MHSA modules in the decoder are applied with attention masks to prevent each position from attending to subsequent positions. These lower triangle-like masks are often referred to as causal masks.

[ ]:
```
#@markdown <a name="causal"></a>
#@title Masked (Causal) Self-Attention implemeatation

from torch import nn, Tensor
import torch.nn.functional as F

class MaskedScaledDotProductAttention(ScaledDotProductAttention):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    def forward(self,
                q: Tensor, k: Tensor, v: Tensor,
```

```python
                mask: Tensor = None) -> Tensor:
        # q.shape: [..., n, k]
        # k.shape: [..., m, k]
        # v.shape: [..., m, v]
        # mask.shape: [..., n, m]

        # Q @ K.T / (d_k ** 0.5)
        attn = torch.einsum("... n k, ... m k -> ... n m", q, k)
        attn /= self.temperature
        # apply causal mask
        if mask is not None:
            attn = attn.masked_fill(mask == 0, -1e9)
        # softmax
        attn = F.softmax(attn, dim=-1)
        # A @ V
        output = torch.einsum("... n m, ... m v -> ... n v", attn, v)

        return output, attn

def get_causal_mask(m: int, n: int) -> Tensor:
    mask = torch.tril(torch.ones(m, n))
    return mask


d_k = 8
d_v = 16
q_len = 10
k_len = v_len = 11

q = torch.randn(1, q_len, d_k)
k = torch.randn(1, k_len, d_k)
v = torch.randn(1, v_len, d_v)

masked_attention = MaskedScaledDotProductAttention(temperature=d_k ** 0.5)
causal_mask = get_causal_mask(q_len, k_len)
output, attn = masked_attention(q, k, v, causal_mask)
print(f"Attention:\t{attn.shape}")
print(f"Output: \t{output.shape}")

print("\nCausal mask:")
import matplotlib.pyplot as plt

fig, ax = plt.subplots(layout='constrained')
im = ax.imshow(causal_mask)
cbar = fig.colorbar(im)
plt.show()
```

```
Attention:     torch.Size([1, 10, 11])
Output:        torch.Size([1, 10, 16])

Causal mask:
```



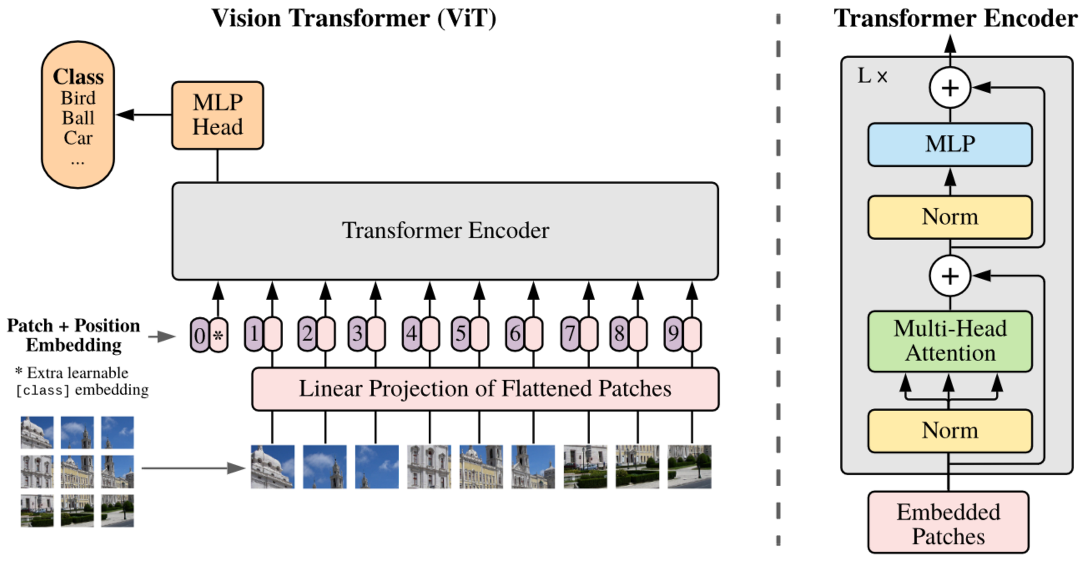
3.2 Vision Transformer (ViT)

3.2.1 From Language to Vision

There is always a domain gap between NLP and CV [45]:

- CV aims to deal with images, which are regular grids with *continous* pixel values. And these natural signals usually have heavy spatial redundancy.
- NLP aims to deal with words, which are predefined *discret* tokens. And these human-generated signals are highly semantic and information-dense.

This gap introduces some extra difficulty for CV to draw lessons from NLP, e.g., directly use already successful models and architectures, such as Transformer. Nevertheless, Google experimented with applying a standard Transformer directly to images, with the fewest possible modifications [3]. The model they proposed is Vision Transformer(ViT), whose overview is shown below.



3.2.2 Adaptation from Vanilla Transformer

As most of vision backbones, ViT's mission is to encoder images into a representation space. Therefore, ViT only exploits the encoder of vanilla Transformer. To handle 2D images, ViT patchifies the original image $\mathbf{x} \in \mathbb{R}^{H\times W\times C}$ before inputting to the Transformer encoder:

$$(H, W, C) \rightarrow \left(\frac{H}{P} \times P, \frac{W}{P} \times P, C\right) \rightarrow \left(\frac{HW}{P^2}, P^2C\right),$$

where (H, W) is the resolution of the original image, C is the number of channels, (P, P) is the resolution of each image patch, and $N = HW/P^2$ is the resulting number of patches, which also serves as the effective input sequence length for the Transformer. These patches are further projected from pixel space to embedding space. The projection is done by a single linear layer, and this operation is often referred to as Patch Embedding.

Similar to BERT [37]'s `[class]` token, ViT prepends a learnable embedding $\mathbf{x}_{\text{class}}$ to the sequence of embedded patches, whose state at the output of the Transformer encoder will serve as the image representation. Thus classification can be done by appending a classification head, usually a two-layer MLP.

Although it's natural to extend the 1D positional embeddings in NLP tasks to 2D positional embeddings in CV tasks, no significant performance gain has been observed in ViT's experiment.

What's more, ViT changes the original ReLU non-linear function to a more advanced GELU [46] activation. And ViT adopts the pre-norm [44], instead of the post-norm in original Transformer.

As described in ViT paper [3], Vision Transformer has much less image-specific inductive bias than CNNs. Convolutional layers bring the prior knowledge of two-dimensional neighborhood structure and translation equivariance, while self-attention layers are global. It is the hierarchical Transformers (e.g., Swin Transformers [22]) that reintroduced several CNN inductive biases, making Transformers practically viable as a generic vision backbone and demonstrating remarkable performance on a wide variety of vision tasks.

3.3 Design of CNNs

Depthwise Separatable Convolution

The key design of MBConv block is so-called "Depthwise Separable Convolutions" proposed in MobileNet [47]. By "depthwise separable", it emphasizes to factorize a standard convolution into a depthwise convolution (DWConv) and a pointwise convolution (also called PWConv or 1×1 convolution).

DWConv is a special case of group convolution where the group number equals to channel number exactly. Since each channel is processed separately, DWConv is more a spatial information mixer.

In contrast, PWConv doesn't mix any spatial inforamtion due to its 1×1 kernel size. It is responsible for performing inter-channel feature fusion.

> Note that a simple stack of a DWConv and a PWConv could be not as expressive as a regular convolutional layer, but this could be regarded as a reasonable regularization.

Bottleneck vs. Inverted Bottleneck

To reduce computational cost and introduce more non-linearity, ResNet [41] introduced Bottleneck as a CNN building block design. As its name, "Bottleneck" block uses two PWConv layers to decrease down and increase back the channel number at input and output respectively. See figure below from ResNet paper [41].

However, in MobileNetV2 [13], from a perspective of manifold, the authors proposed that, although the Bottleneck block introduce more non-linearity, the decrease of channels also leads to excessive loss of features, which undermines the model's representative ability. Therefore, they introduced Inverted Bottleneck, which increase channel number before decreasing it.

Macro Design in CNNs

From experiences of building a vision model, typically there are 1 stem, 4 stages, and 1 head in a model.

The stem will downsample the original image, typically by a factor of 4, to reduce subsequent computational overheads.

Each of the following 4 stages contains a sequence of convolutional blocks. The resolution of the feature maps is reduced via downsampling between stages, while it remains constant within each stage. As the stages going deeper, the number of channels also increases accordingly. This "staged" design allows the model to progressively extract information at each semantic level, which also benefits the application of downstream tasks, e.g., Feature Pyramids Network (FPN) [48] in object detection.

> We can say that the stem and all stages together consist of the so-called "backbone".

Finally, the head is used to take on various downstream tasks, such as classification head or detection head[13].

[13]In some models for detection, there is even a "neck" to aggregate features from different semantic levels.

4 Main Claims

The MaxViT paper evaluated different variants of MaxViT (with a spectrum of parameter number) on a variety of tasks, including image classification, object detection, instance segmentation, image aesthetic assessment, and image generation.

Below are some main claims:

- On image classification tasks, MaxViT generally outperformed other SOTA models, such as EfficientNetV2 [15], Swin Transformer V2 [49], ConvNeXt [16], and CoAtNet [21].
- On object detection and instance segmentation tasks with COCO dataset [6], MaxViT backbone outperform other SOTA backbones with respect to both accuracy and efficiency.
- On image aesthetics/quality assessment with AVA benchmark [7], MaxViT outperforms and shows better linear corre- lation than the SOTA method MUSIQ.
- On unconditional image generation task with Generative adversarial network (GAN) [5], MaxViT achieves better Fréchet Inception Distance (FID) and Inception Score (IS) with significantly less parameters. I choose the first one as a primary claim and make some discussion on it.

4.1 Primary Claim: On image classification tasks, MaxViT generally outperformed the most recent strong hybrid model CoAtNet.

4.1.1 Evidence

MaxViT has been evaluated thoroughly on image classification task. In the paper of MaxViT, the comprehensive metric table (shown below) demonstrates the SOTA performance of MaxViT when training under ImageNet-1K [4] setting. (table from MaxViT paper)

Table 2: **Performance comparison under ImageNet-1K setting.** Throughput is measured on a single V100 GPU with batch size 16, following [56, 57, 80].

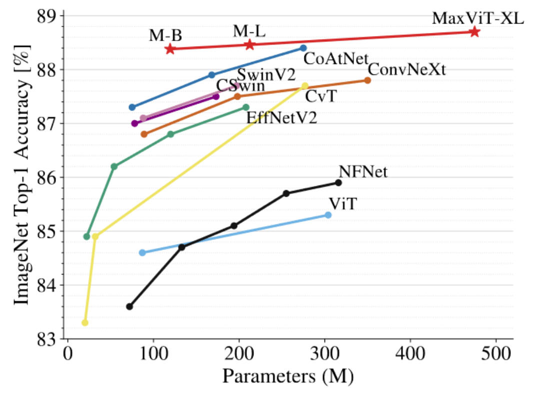
	Model	Eval size	Params	FLOPs	Throughput (image/s)	IN-1K top-1 acc.
ConvNets	●EffNet-B6 [79]	528	43M	19.0G	96.9	84.0
	●EffNet-B7 [79]	600	66M	37.0G	55.1	84.3
	●RegNetY-16 [62]	224	84M	16.0G	334.7	82.9
	●NFNet-F0 [5]	256	72M	12.4G	533.3	83.6
	●NFNet-F1 [5]	320	132M	35.5G	228.5	84.7
	●EffNetV2-S [80]	384	24M	8.8G	666.6	83.9
	●EffNetV2-M [80]	480	55M	24.0G	280.7	85.1
	●ConvNeXt-S [57]	224	50M	8.7G	447.1	83.1
	●ConvNeXt-B [57]	224	89M	15.4G	292.1	83.8
	●ConvNeXt-L [57]	224	198M	34.4G	146.8	84.3
ViTs	○ViT-B/32 [22]	384	86M	55.4G	85.9	77.9
	○ViT-B/16 [22]	384	307M	190.7G	27.3	76.5
	○DeiT-B [81]	384	86M	55.4G	85.9	83.1
	○CaiT-M24 [82]	224	186M	36.0G	-	83.4
	○CaiT-M24 [82]	384	186M	116.1G	-	84.5
	○DeepViT-L [105]	224	55M	12.5G	-	83.1
	○T2T-ViT-24 [101]	224	64M	15.0G	-	82.6
	○Swin-S [56]	224	50M	8.7G	436.9	83.0
	○Swin-B [56]	384	88M	47.0G	84.7	84.5
	○CSwin-B [21]	224	78M	15.0G	250	84.2
	○CSwin-B [21]	384	78M	47.0G	-	85.4
	○Focal-S [99]	224	51M	9.1G	-	83.5
	○Focal-B [99]	224	90M	16.0G	-	83.8
Hybrid	◇CvT-21 [93]	384	32M	24.9G	-	83.3
	◇CoAtNet-2 [19]	224	75M	15.7G	247.7	84.1
	◇CoAtNet-3 [19]	224	168M	34.7G	163.3	84.5
	◇CoAtNet-3 [19]	384	168M	107.4G	48.5	85.8
	◇CoAtNet-3 [19]	512	168M	203.1G	22.4	86.0
	◇MaxViT-T	224	31M	5.6G	349.6	83.62
	◇MaxViT-S	224	69M	11.7G	242.5	84.45
	◇MaxViT-B	224	120M	23.4G	133.6	84.95
	◇MaxViT-L	224	212M	43.9G	99.4	85.17
	◇MaxViT-T	384	31M	17.7G	121.9	85.24
	◇MaxViT-S	384	69M	36.1G	82.7	85.74
	◇MaxViT-B	384	120M	74.2G	45.8	86.34
	◇MaxViT-L	384	212M	133.1G	34.3	86.40
	◇MaxViT-T	512	31M	33.7G	63.8	85.72
	◇MaxViT-S	512	69M	67.6G	43.3	86.19
	◇MaxViT-B	512	120M	138.5G	24.0	86.66
	◇MaxViT-L	512	212M	245.4G	17.8	**86.70**

The following table further demonstrates the great scalibility of MaxViT when pre-training on large scale datasets (ImageNet-21K and JFT-300M [50]) and then fine-tuning on ImageNet-1K.

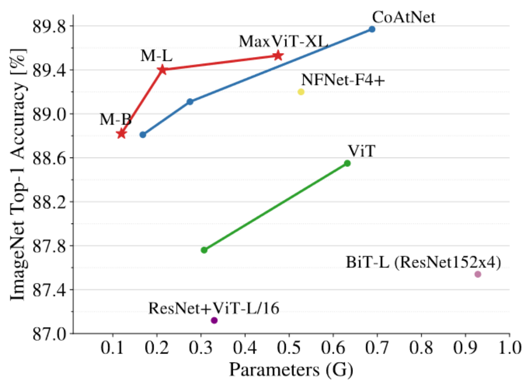
Table 3: **Performance comparison for large-scale data regimes**: ImageNet-21K and JFT pretrained models.

	Model	Eval size	Params	FLOPs	IN-1K top-1 acc. 21K→1K	IN-1K top-1 acc. JFT→1K
ConvNets	●BiT-R-101x3 [46]	384	388M	204.6G	84.4	-
	●BiT-R-152x4 [46]	480	937M	840.5G	85.4	-
	●EffNetV2-L [80]	480	121M	53.0G	86.8	-
	●EffNetV2-XL [80]	512	208M	94.0G	87.3	-
	●ConvNeXt-L [57]	384	198M	101.0G	87.5	-
	●ConvNeXt-XL [57]	384	350M	179.0G	87.8	-
	●NFNet-F4+ [5]	512	527M	367G	-	89.20
ViTs	○ViT-B/16 [22]	384	87M	55.5G	84.0	-
	○ViT-L/16 [22]	384	305M	191.1G	85.2	
	○ViT-L/16 [22]	512	305M	364G	-	87.76
	○ViT-H/14 [22]	518	632M	1021G	-	88.55
	○HaloNet-H4 [84]	512	85M	-	85.8	-
	○SwinV2-B [56]	384	88M	-	87.1	-
	○SwinV2-L [56]	384	197M	-	87.7	-
Hybrid	◇CvT-W24 [93]	384	277M	193.2G	87.7	-
	◇R+ViT-L/16 [22]	384	330M	-	-	87.12
	◇CoAtNet-3 [19]	384	168M	107.4G	87.6	88.52
	◇CoAtNet-3 [19]	512	168M	214G	87.9	88.81
	◇CoAtNet-4 [19]	512	275M	360.9G	88.1	89.11
	◇CoAtNet-5 [19]	512	688M	812G	-	**89.77**
	◇MaxViT-B	384	119M	74.2G	88.24	88.69
	◇MaxViT-L	384	212M	128.7G	88.32	89.12
	◇MaxViT-XL	384	475M	293.7G	**88.51**	89.36
	◇MaxViT-B	512	119M	138.3G	88.38	88.82
	◇MaxViT-L	512	212M	245.2G	88.46	89.41
	◇MaxViT-XL	512	475M	535.2G	**88.70**	**89.53**

Additionally, according to the following Accuracy vs. Parameters line plot, we can clearly see that MaxViT has better performance than other SOTA models when their number of parameters are comparable.



(a) Accuracy *vs.* Params performances for ImageNet-21K pre-trained models.



(b) Accuracy *vs.* Params scaling curve for JFT-300M pre-trained models.

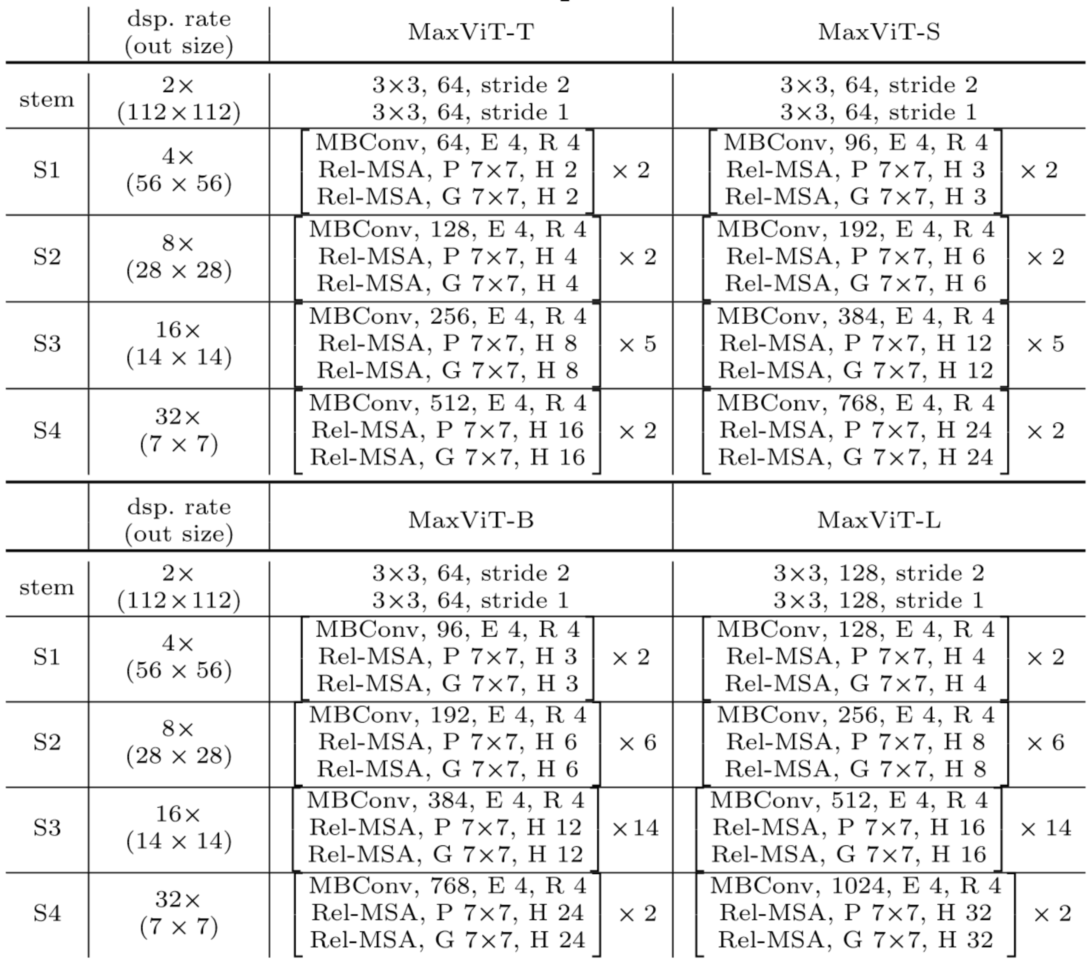
Fig. 4: **Performance comparison on large-scale pre-trained models.** MaxViT shows superior scaling performance under both ImageNet-21K and JFT-300M pre-trained settings.

4.1.2 Details of Experiments

Note the paper evaluated MaxViT with different common resolutions, including 224×224, 384×384, and 512×512.

Below are different variants of MaxViT with a spectrum of parameter numbers. (table from MaxViT paper)

Table 11: **Detailed architectural specifications** for MaxViT families.



	dsp. rate (out size)	MaxViT-T		MaxViT-S	
stem	2× (112×112)	3×3, 64, stride 2 3×3, 64, stride 1		3×3, 64, stride 2 3×3, 64, stride 1	
S1	4× (56 × 56)	MBConv, 64, E 4, R 4 Rel-MSA, P 7×7, H 2 Rel-MSA, G 7×7, H 2	× 2	MBConv, 96, E 4, R 4 Rel-MSA, P 7×7, H 3 Rel-MSA, G 7×7, H 3	× 2
S2	8× (28 × 28)	MBConv, 128, E 4, R 4 Rel-MSA, P 7×7, H 4 Rel-MSA, G 7×7, H 4	× 2	MBConv, 192, E 4, R 4 Rel-MSA, P 7×7, H 6 Rel-MSA, G 7×7, H 6	× 2
S3	16× (14 × 14)	MBConv, 256, E 4, R 4 Rel-MSA, P 7×7, H 8 Rel-MSA, G 7×7, H 8	× 5	MBConv, 384, E 4, R 4 Rel-MSA, P 7×7, H 12 Rel-MSA, G 7×7, H 12	× 5
S4	32× (7 × 7)	MBConv, 512, E 4, R 4 Rel-MSA, P 7×7, H 16 Rel-MSA, G 7×7, H 16	× 2	MBConv, 768, E 4, R 4 Rel-MSA, P 7×7, H 24 Rel-MSA, G 7×7, H 24	× 2

	dsp. rate (out size)	MaxViT-B		MaxViT-L	
stem	2× (112×112)	3×3, 64, stride 2 3×3, 64, stride 1		3×3, 128, stride 2 3×3, 128, stride 1	
S1	4× (56 × 56)	MBConv, 96, E 4, R 4 Rel-MSA, P 7×7, H 3 Rel-MSA, G 7×7, H 3	× 2	MBConv, 128, E 4, R 4 Rel-MSA, P 7×7, H 4 Rel-MSA, G 7×7, H 4	× 2
S2	8× (28 × 28)	MBConv, 192, E 4, R 4 Rel-MSA, P 7×7, H 6 Rel-MSA, G 7×7, H 6	× 6	MBConv, 256, E 4, R 4 Rel-MSA, P 7×7, H 8 Rel-MSA, G 7×7, H 8	× 6
S3	16× (14 × 14)	MBConv, 384, E 4, R 4 Rel-MSA, P 7×7, H 12 Rel-MSA, G 7×7, H 12	×14	MBConv, 512, E 4, R 4 Rel-MSA, P 7×7, H 16 Rel-MSA, G 7×7, H 16	× 14
S4	32× (7 × 7)	MBConv, 768, E 4, R 4 Rel-MSA, P 7×7, H 24 Rel-MSA, G 7×7, H 24	× 2	MBConv, 1024, E 4, R 4 Rel-MSA, P 7×7, H 32 Rel-MSA, G 7×7, H 32	× 2

The hyperparameters they use for training are as follows: (table from MaxViT appendix)

Table 12: **Detailed hyperparameters used in ImageNet-1K experiments.** Multiple values separated by '/' are for each model size respectively.

Hyperparameter	ImageNet-1K Pre-training (MaxViT-T/S/B/L)	ImageNet-1K Fine-tuning (MaxViT-T/S/B/L)	ImageNet-21K Pre-training (MaxViT-B/L/XL)	ImageNet-21K Fine-tuning (MaxViT-B/L/XL)	JFT-300M Pre-training (MaxViT-B/L/XL)	JFT-300M Fine-tuning (MaxViT-B/L/XL)
Stochastic depth	0.2/0.3/0.4/0.6		0.3/0.4/0.6	0.4/0.5/0.9	0.0/0.0/0.0	0.1/0.2/0.2
Center crop	True	False	True	False	True	False
RandAugment	2, 15	2, 15	2, 5	2, 15	2, 5	2, 15
Mixup alpha	0.8	0.8	None	None	None	None
Loss type	Softmax	Softmax	Sigmoid	Softmax	Sigmoid	Softmax
Label smoothing	0.1	0.1	0.0001	0.1	0	0.1
Train epochs	300	30	90	30	14	30
Train batch size	4096	512	4096	512	4096	512
Optimizer type	AdamW	AdamW	AdamW	AdamW	AdamW	AdamW
Peak learning rate	3e-3	5e-5	1e-3	5e-5	1e-3	5e-5
Min learning rate	1e-5	5e-5	1e-5	5e-5	1e-5	5e-5
Warm-up	10K steps	None	5 epochs	None	20K steps	None
LR decay schedule	Cosine	None	Linear	None	Linear	None
Weight decay rate	0.05	1e-8	0.01	1e-8	0.01	1e-8
Gradient clip	1.0	1.0	1.0	1.0	1.0	1.0
EMA decay rate	None	0.9999	None	0.9999	None	0.9999

4.1.3 Validation & Reproduction

One of the large scale dataset they used to pre-train MaxViT is JFT-300M established by Google, which is not public available. Moreover, due to the incredibly huge computational resources and cost it requires to follow the posted training recipe exactly, we can not directly validate the primary claim (performance on accuracy part) via pre-training and testing all models.

However, since MaxViT has published some pre-training weights on GitHub, it could be practical to evaluate these weights directly on the validation split of ImageNet-1K dataset.

As for the throughput part, we can not directly run inference on unavailable V100 GPU. However, we can run inference on T4 GPU provided by Colab, and we expect to have a similar relative rank.

5 Code

In the following sub-sections, we are going to:

- setup the environment and install the MaxViT Python Package from official repository (in TensorFlow version)
- select a variant of MaxViT model and benchmark its throughput (images per second)
- run some inferences with other variants on …
  - example images (same resolution as training configuration)
  - example images (higher resolution than training configuration)

Attribution: Codes in this section are directly copy-pasted from the official MaxViT_tutorial, with minor modifications in titles.

[ ]:
```
#@title Import and install
import time
from IPython import display
import tensorflow as tf
import tensorflow.compat.v1 as tf1
import tensorflow_datasets as tfds
!git clone https://github.com/google-research/maxvit
%cd /content/maxvit
# set up module
!python setup.py install

# imports
import maxvit.models.hparams as hparams
import maxvit.models.maxvit as layers

# Checkpoints location
CKPTS_DIRS = {
    'MaxViTTiny_i1k_224': 'gs://gresearch/maxvit/ckpts/maxvittiny/i1k/224',
    'MaxViTTiny_i1k_384': 'gs://gresearch/maxvit/ckpts/maxvittiny/i1k/384',
    'MaxViTTiny_i1k_512': 'gs://gresearch/maxvit/ckpts/maxvittiny/i1k/512',
    'MaxViTSmall_i1k_224': 'gs://gresearch/maxvit/ckpts/maxvitsmall/i1k/224',
    'MaxViTSmall_i1k_384': 'gs://gresearch/maxvit/ckpts/maxvitsmall/i1k/384',
    'MaxViTSmall_i1k_512': 'gs://gresearch/maxvit/ckpts/maxvitsmall/i1k/512',
    'MaxViTBase_i1k_224': 'gs://gresearch/maxvit/ckpts/maxvitbase/i1k/224',
    'MaxViTBase_i1k_384': 'gs://gresearch/maxvit/ckpts/maxvitbase/i1k/384',
    'MaxViTBase_i1k_512': 'gs://gresearch/maxvit/ckpts/maxvitbase/i1k/512',
    'MaxViTBase_i21k_i1k_224': None,
    'MaxViTBase_i21k_i1k_384': 'gs://gresearch/maxvit/ckpts/maxvitbase/i21k_i1k/
  ↪384',
    'MaxViTBase_i21k_i1k_512': 'gs://gresearch/maxvit/ckpts/maxvitbase/i21k_i1k/
  ↪512',
    'MaxViTLarge_i1k_224': 'gs://gresearch/maxvit/ckpts/maxvitlarge/i1k/224',
    'MaxViTLarge_i1k_384': 'gs://gresearch/maxvit/ckpts/maxvitlarge/i1k/384',
    'MaxViTLarge_i1k_512': 'gs://gresearch/maxvit/ckpts/maxvitlarge/i1k/512',
    'MaxViTLarge_i21k_i1k_224': None,
    'MaxViTLarge_i21k_i1k_384': 'gs://gresearch/maxvit/ckpts/maxvitlarge/
  ↪i21k_i1k/384',
    'MaxViTLarge_i21k_i1k_512': 'gs://gresearch/maxvit/ckpts/maxvitlarge/
  ↪i21k_i1k/512',
    'MaxViTXLarge_i21k_i1k_224': None,
    'MaxViTXLarge_i21k_i1k_384': 'gs://gresearch/maxvit/ckpts/maxvitxlarge/
  ↪i21k_i1k/384',
    'MaxViTXLarge_i21k_i1k_512': 'gs://gresearch/maxvit/ckpts/maxvitxlarge/
  ↪i21k_i1k/512',
}

DATASET_MAP = {
    'ImageNet-1K': 'i1k',
    'ImageNet-21K': 'i21k_i1k',
}
```

```
Cloning into 'maxvit'…
remote: Enumerating objects: 90, done.
remote: Counting objects: 100% (90/90), done.
remote: Compressing objects: 100% (62/62), done.
remote: Total 90 (delta 44), reused 66 (delta 26), pack-reused 0
Receiving objects: 100% (90/90), 1.48 MiB | 4.91 MiB/s, done.
Resolving deltas: 100% (44/44), done.
/content/maxvit
running install
/usr/local/lib/python3.10/dist-packages/setuptools/_distutils/cmd.py:66:
SetuptoolsDeprecationWarning: setup.py install is deprecated.
!!

********************************************************************************
        Please avoid running ``setup.py`` directly.
        Instead, use pypa/build, pypa/installer, pypa/build or
        other standards-based tools.

        See https://blog.ganssle.io/articles/2021/10/setup-py-deprecated.html
for details.
********************************************************************************

!!
  self.initialize_options()
/usr/local/lib/python3.10/dist-packages/setuptools/_distutils/cmd.py:66:
EasyInstallDeprecationWarning: easy_install command is deprecated.
!!

********************************************************************************
        Please avoid running ``setup.py`` and ``easy_install``.
        Instead, use pypa/build, pypa/installer, pypa/build or
        other standards-based tools.

        See https://github.com/pypa/setuptools/issues/917 for details.
********************************************************************************

!!
  self.initialize_options()
running bdist_egg
running egg_info
creating maxvit.egg-info
writing maxvit.egg-info/PKG-INFO
writing dependency_links to maxvit.egg-info/dependency_links.txt
writing requirements to maxvit.egg-info/requires.txt
writing top-level names to maxvit.egg-info/top_level.txt
writing manifest file 'maxvit.egg-info/SOURCES.txt'
reading manifest file 'maxvit.egg-info/SOURCES.txt'
adding license file 'LICENSE'
```

```
writing manifest file 'maxvit.egg-info/SOURCES.txt'
installing library code to build/bdist.linux-x86_64/egg
running install_lib
running build_py
creating build
creating build/lib
creating build/lib/maxvit
copying maxvit/test_maxvit.py -> build/lib/maxvit
copying maxvit/__init__.py -> build/lib/maxvit
creating build/lib/maxvit/models
copying maxvit/models/maxvit.py -> build/lib/maxvit/models
copying maxvit/models/hparam_configs.py -> build/lib/maxvit/models
copying maxvit/models/__init__.py -> build/lib/maxvit/models
copying maxvit/models/common_ops.py -> build/lib/maxvit/models
copying maxvit/models/utils.py -> build/lib/maxvit/models
copying maxvit/models/attention_utils.py -> build/lib/maxvit/models
copying maxvit/models/hparams.py -> build/lib/maxvit/models
copying maxvit/models/eval_ckpt.py -> build/lib/maxvit/models
copying maxvit/models/hparams_registry.py -> build/lib/maxvit/models
creating build/lib/maxvit/models/hp
copying maxvit/models/hp/__init__.py -> build/lib/maxvit/models/hp
copying maxvit/models/hp/vision_i1k.py -> build/lib/maxvit/models/hp
copying maxvit/models/hp/vision.py -> build/lib/maxvit/models/hp
creating build/bdist.linux-x86_64
creating build/bdist.linux-x86_64/egg
creating build/bdist.linux-x86_64/egg/maxvit
creating build/bdist.linux-x86_64/egg/maxvit/models
copying build/lib/maxvit/models/maxvit.py ->
build/bdist.linux-x86_64/egg/maxvit/models
copying build/lib/maxvit/models/hparam_configs.py ->
build/bdist.linux-x86_64/egg/maxvit/models
copying build/lib/maxvit/models/__init__.py ->
build/bdist.linux-x86_64/egg/maxvit/models
copying build/lib/maxvit/models/common_ops.py ->
build/bdist.linux-x86_64/egg/maxvit/models
copying build/lib/maxvit/models/utils.py ->
build/bdist.linux-x86_64/egg/maxvit/models
creating build/bdist.linux-x86_64/egg/maxvit/models/hp
copying build/lib/maxvit/models/hp/__init__.py ->
build/bdist.linux-x86_64/egg/maxvit/models/hp
copying build/lib/maxvit/models/hp/vision_i1k.py ->
build/bdist.linux-x86_64/egg/maxvit/models/hp
copying build/lib/maxvit/models/hp/vision.py ->
build/bdist.linux-x86_64/egg/maxvit/models/hp
copying build/lib/maxvit/models/attention_utils.py ->
build/bdist.linux-x86_64/egg/maxvit/models
copying build/lib/maxvit/models/hparams.py ->
build/bdist.linux-x86_64/egg/maxvit/models
```

```
copying build/lib/maxvit/models/eval_ckpt.py ->
build/bdist.linux-x86_64/egg/maxvit/models
copying build/lib/maxvit/models/hparams_registry.py ->
build/bdist.linux-x86_64/egg/maxvit/models
copying build/lib/maxvit/test_maxvit.py -> build/bdist.linux-x86_64/egg/maxvit
copying build/lib/maxvit/__init__.py -> build/bdist.linux-x86_64/egg/maxvit
byte-compiling build/bdist.linux-x86_64/egg/maxvit/models/maxvit.py to
maxvit.cpython-310.pyc
byte-compiling build/bdist.linux-x86_64/egg/maxvit/models/hparam_configs.py to
hparam_configs.cpython-310.pyc
byte-compiling build/bdist.linux-x86_64/egg/maxvit/models/__init__.py to
__init__.cpython-310.pyc
byte-compiling build/bdist.linux-x86_64/egg/maxvit/models/common_ops.py to
common_ops.cpython-310.pyc
byte-compiling build/bdist.linux-x86_64/egg/maxvit/models/utils.py to
utils.cpython-310.pyc
byte-compiling build/bdist.linux-x86_64/egg/maxvit/models/hp/__init__.py to
__init__.cpython-310.pyc
byte-compiling build/bdist.linux-x86_64/egg/maxvit/models/hp/vision_i1k.py to
vision_i1k.cpython-310.pyc
byte-compiling build/bdist.linux-x86_64/egg/maxvit/models/hp/vision.py to
vision.cpython-310.pyc
byte-compiling build/bdist.linux-x86_64/egg/maxvit/models/attention_utils.py to
attention_utils.cpython-310.pyc
byte-compiling build/bdist.linux-x86_64/egg/maxvit/models/hparams.py to
hparams.cpython-310.pyc
byte-compiling build/bdist.linux-x86_64/egg/maxvit/models/eval_ckpt.py to
eval_ckpt.cpython-310.pyc
byte-compiling build/bdist.linux-x86_64/egg/maxvit/models/hparams_registry.py to
hparams_registry.cpython-310.pyc
byte-compiling build/bdist.linux-x86_64/egg/maxvit/test_maxvit.py to
test_maxvit.cpython-310.pyc
byte-compiling build/bdist.linux-x86_64/egg/maxvit/__init__.py to
__init__.cpython-310.pyc
creating build/bdist.linux-x86_64/egg/EGG-INFO
copying maxvit.egg-info/PKG-INFO -> build/bdist.linux-x86_64/egg/EGG-INFO
copying maxvit.egg-info/SOURCES.txt -> build/bdist.linux-x86_64/egg/EGG-INFO
copying maxvit.egg-info/dependency_links.txt ->
build/bdist.linux-x86_64/egg/EGG-INFO
copying maxvit.egg-info/requires.txt -> build/bdist.linux-x86_64/egg/EGG-INFO
copying maxvit.egg-info/top_level.txt -> build/bdist.linux-x86_64/egg/EGG-INFO
zip_safe flag not set; analyzing archive contents…
creating dist
creating 'dist/maxvit-1.0.0-py3.10.egg' and adding
'build/bdist.linux-x86_64/egg' to it
removing 'build/bdist.linux-x86_64/egg' (and everything under it)
Processing maxvit-1.0.0-py3.10.egg
Copying maxvit-1.0.0-py3.10.egg to /usr/local/lib/python3.10/dist-packages
```

```
Adding maxvit 1.0.0 to easy-install.pth file

Installed /usr/local/lib/python3.10/dist-packages/maxvit-1.0.0-py3.10.egg
Processing dependencies for maxvit==1.0.0
Searching for ml-collections==0.1.0
Reading https://pypi.org/simple/ml-collections/
Downloading https://files.pythonhosted.org/packages/03/d4/9ab1a8c2aebf78c348404c
464733974dc4e7088174d6272ed09c2fa5a8fa/ml_collections-0.1.0-py3-none-
any.whl#sha256=0f30752d52fae1c09c10a406fc5d405716da2b60fa5f13dd15498615cb44d3c9
Best match: ml-collections 0.1.0
Processing ml_collections-0.1.0-py3-none-any.whl
Installing ml_collections-0.1.0-py3-none-any.whl to
/usr/local/lib/python3.10/dist-packages
Adding ml-collections 0.1.0 to easy-install.pth file

Installed /usr/local/lib/python3.10/dist-
packages/ml_collections-0.1.0-py3.10.egg
Searching for clu>=0.0.3
Reading https://pypi.org/simple/clu/
Downloading https://files.pythonhosted.org/packages/a5/4f/3a65478c297046dc678202
a468a4b3479d124936fe77ea7f3d6fed49baee/clu-0.0.10-py3-none-
any.whl#sha256=9c286c591c5207b7beed856da6e1f016c462d1234a46807802c573e0da3ea7c9
Best match: clu 0.0.10
Processing clu-0.0.10-py3-none-any.whl
Installing clu-0.0.10-py3-none-any.whl to /usr/local/lib/python3.10/dist-
packages
Adding clu 0.0.10 to easy-install.pth file

Installed /usr/local/lib/python3.10/dist-packages/clu-0.0.10-py3.10.egg
error: numpy 1.23.5 is installed but numpy==1.23.1 is required by {'clu'}
```

[ ]:
```
#@title Benchmark inference time

MODEL_NAME = "MaxViTTiny" #@param ["MaxViTTiny", "MaxViTSmall", "MaxViTBase",␣
 ↪"MaxViTLarge"] {type:"string"}
IMAGE_SIZE = "224" #@param [224, 384, 512] {type:"string"}
BATCH_SIZE = 16 #@param {type:"integer"}
MIXED_PRECISION = True #@param {type:"boolean"}

IMAGE_SIZE = int(IMAGE_SIZE)


class MaxViTModel(tf.keras.Model):
  """class to build MaxViT family model."""
  def __init__(self,
               model_name='',
               model_input_size=224,
               input_specs=tf.keras.layers.InputSpec(
                   shape=[None, None, None, 3]),
               training=True):
    """VisionTransformer initialization function."""
    inputs = tf.keras.Input(shape=input_specs.shape[1:])
    config = hparams.lookup(model_name)

    if model_input_size == 224:
      config.model.window_size = 7
      config.model.grid_size = 7
      config.model.scale_ratio = None
    elif model_input_size == 384:
      config.model.window_size = 12
      config.model.grid_size = 12
      config.model.scale_ratio = '384/224'
    elif model_input_size == 512:
      config.model.window_size = 16
      config.model.grid_size = 16
      config.model.scale_ratio = '512/224'

    model = layers.MaxViT(config.model)
    out = model(inputs, training=training)

    super(MaxViTModel, self).__init__(inputs=inputs, outputs=out)


def build_tf2_model():
  """Build the tf2 model."""
  if MIXED_PRECISION:
    # Use 'mixed_float16' if running on GPUs.
    policy = tf.keras.mixed_precision.Policy('mixed_float16')
    tf.keras.mixed_precision.set_global_policy(policy)
  model = MaxViTModel(model_name=MODEL_NAME,
                      model_input_size=IMAGE_SIZE,
                      input_specs=tf.keras.layers.InputSpec(
                      shape=[BATCH_SIZE, IMAGE_SIZE, IMAGE_SIZE, 3]),
                      training=False)
  return model


def run_tf_benchmark():
  """Run benchmark."""
  model = build_tf2_model()
  imgs = tf.ones((BATCH_SIZE, IMAGE_SIZE, IMAGE_SIZE, 3), dtype=tf.float16)

  @tf.function
  def f(x):
    return model(x, training=False)

  print('starting warmup.')
  for _ in range(10):  # warmup runs.
    f(imgs)

  print('start benchmark.')
  start = time.perf_counter()
  for _ in range(10):
    f(imgs)
  end = time.perf_counter()
  inference_time = (end - start) / 10

  print('Per batch inference time: ', inference_time)
  print('FPS: ', BATCH_SIZE / inference_time)

run_tf_benchmark()
```

```
starting warmup.
start benchmark.
Per batch inference time:  0.12835027870000032
FPS:  124.65886449220433
```

5.0.1 Inference on images

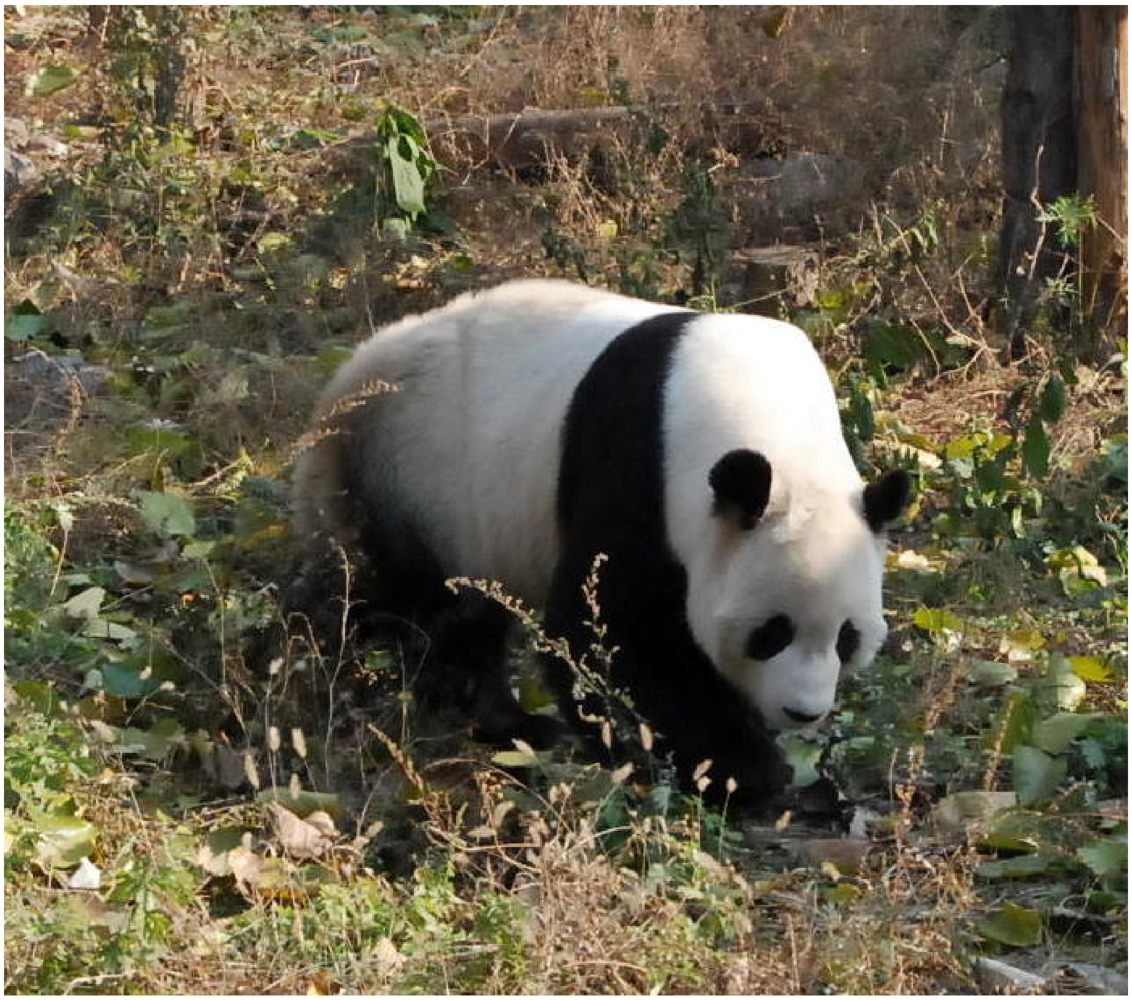
Example image (same resolution)

[ ]:
```
#@title Set model and params

MODEL_NAME = "MaxViTBase" #@param ["MaxViTTiny", "MaxViTSmall", "MaxViTBase",␣
 ↪"MaxViTLarge"] {type:"string"}
TRAIN_SET = "ImageNet-1K" #@param ["ImageNet-1K"] {type:"string"}
TRAIN_IMAGE_SIZE = "224" #@param [224, 384, 512] {type:"string"}
MIXED_PRECISION = False #@param {type:"boolean"}

CKPT_DIR =␣
 ↪CKPTS_DIRS[f'{MODEL_NAME}_{DATASET_MAP[TRAIN_SET]}_{TRAIN_IMAGE_SIZE}']
```

[ ]:
```
#@title Inference on example image
#@markdown Note this code block runs quite slowly (takes ~5 minutes), so please␣
 ↪be patient.
import maxvit.models.eval_ckpt as eval_ckpt

#@markdown ### Enter a file path:
file_path = "https://upload.wikimedia.org/wikipedia/commons/f/fe/
 ↪Giant_Panda_in_Beijing_Zoo_1.JPG" #@param {type:"string"}
INFER_IMAGE_SIZE = "224" #@param [224, 384, 448, 512, 672, 768, 896, 1024]␣
 ↪{type:"string"}
```

```
# Download label map file and image
labels_map_file = 'gs://cloud-tpu-checkpoints/efficientnet/eval_data/labels_map.
 ↪json'
image_file = 'panda.jpg'

!wget {file_path} -O {image_file}

image_files = [image_file]

eval_driver = eval_ckpt.MaxViTDriver(
    model_name=MODEL_NAME,
    model_input_size=TRAIN_IMAGE_SIZE,
    batch_size=1,
    image_size=int(INFER_IMAGE_SIZE),
    include_background_label=False,
    advprop_preprocessing=False,)

print(f"Input image:")
display.display(display.Image(image_file, width=INFER_IMAGE_SIZE))

print(f"MaxViT prediction:")
pred_idx, pred_prob = eval_driver.eval_example_images(
    CKPT_DIR, image_files, labels_map_file)
```

```
--2023-12-14 21:17:45--  https://upload.wikimedia.org/wikipedia/commons/f/fe/Gia
nt_Panda_in_Beijing_Zoo_1.JPG
Resolving upload.wikimedia.org (upload.wikimedia.org)… 103.102.166.240,
2001:df2:e500:ed1a::2:b
Connecting to upload.wikimedia.org
(upload.wikimedia.org)|103.102.166.240|:443… connected.
HTTP request sent, awaiting response… 200 OK
Length: 116068 (113K) [image/jpeg]
Saving to: ‘panda.jpg’

panda.jpg           100%[===================>] 113.35K  --.-KB/s    in 0.006s

2023-12-14 21:17:45 (18.8 MB/s) - ‘panda.jpg’ saved [116068/116068]

Input image:
```



MaxViT prediction:

```
WARNING:tensorflow:From /usr/local/lib/python3.10/dist-
packages/tensorflow/python/util/dispatch.py:1260: resize_bicubic (from
tensorflow.python.ops.image_ops_impl) is deprecated and will be removed in a
future version.
Instructions for updating:
Use `tf.image.resize(…method=ResizeMethod.BICUBIC…)` instead.
WARNING:tensorflow:From /content/maxvit/maxvit/models/eval_ckpt.py:182:
DatasetV1.make_one_shot_iterator (from tensorflow.python.data.ops.dataset_ops)
is deprecated and will be removed in a future version.
Instructions for updating:
This is a deprecated API that should only be used in TF 1 graph mode and legacy
TF 2 graph mode available through `tf.compat.v1`. In all other situations --
namely, eager mode and inside `tf.function` -- you can consume dataset elements
using `for elem in dataset: …` or by explicitly creating iterator via
`iterator = iter(dataset)` and fetching its elements via `values =
next(iterator)`. Furthermore, this API is not available in TF 2. During the
transition from TF 1 to TF 2 you can use
`tf.compat.v1.data.make_one_shot_iterator(dataset)` to create a TF 1 graph mode
style iterator for a dataset created through TF 2 APIs. Note that this should be
a transient state of your code base as there are in general no guarantees about
the interoperability of TF 1 and TF 2 code.

predicted class for image panda.jpg:
  -> top_0 (90.62%): giant panda, panda, panda bear, coon bear, Ailuropoda
melanoleuca
  -> top_1 (0.21%): lesser panda, red panda, panda, bear cat, cat bear, Ailurus
fulgens
  -> top_2 (0.07%): earthstar
  -> top_3 (0.05%): soccer ball
  -> top_4 (0.04%): sloth bear, Melursus ursinus, Ursus ursinus
```

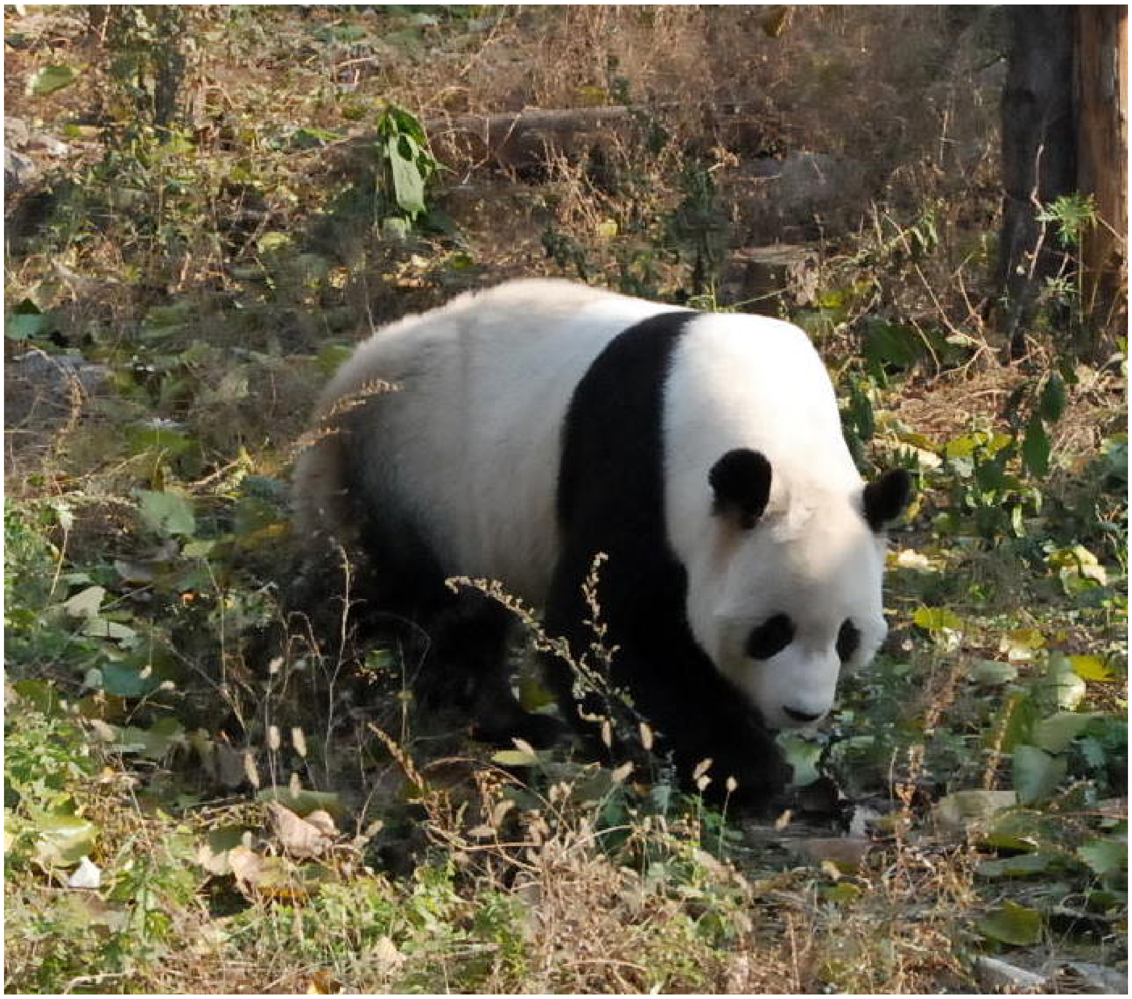
Directly inference on higher resolution Note INFER_IMAGE_SIZE needs to be multipliers of TRAIN_IMAGE_SIZE

[ ]:
```
#@title Set model and params

MODEL_NAME = "MaxViTTiny" #@param ["MaxViTTiny", "MaxViTSmall", "MaxViTBase",␣
 ↪"MaxViTLarge"] {type:"string"}
TRAIN_SET = "ImageNet-1K" #@param ["ImageNet-1K"] {type:"string"}
TRAIN_IMAGE_SIZE = "224" #@param [224, 384, 512] {type:"string"}
MIXED_PRECISION = False #@param {type:"boolean"}

CKPT_DIR =␣
 ↪CKPTS_DIRS[f'{MODEL_NAME}_{DATASET_MAP[TRAIN_SET]}_{TRAIN_IMAGE_SIZE}']
```

[ ]:
```
#@title Inference on example image
#@markdown Note this code block runs quite slowly (takes ~2 minutes), so please␣
 ↪be patient.
import maxvit.models.eval_ckpt as eval_ckpt

#@markdown ### Enter a file path:
file_path = "https://upload.wikimedia.org/wikipedia/commons/f/fe/
 ↪Giant_Panda_in_Beijing_Zoo_1.JPG" #@param {type:"string"}
INFER_IMAGE_SIZE = "672" #@param [224, 384, 448, 512, 672, 768, 896, 1024]␣
 ↪{type:"string"}

# Download label map file and image
labels_map_file = 'gs://cloud-tpu-checkpoints/efficientnet/eval_data/labels_map.
 ↪json'
image_file = 'panda.jpg'

!wget {file_path} -O {image_file}
```

```
image_files = [image_file]

eval_driver = eval_ckpt.MaxViTDriver(
    model_name=MODEL_NAME,
    model_input_size=TRAIN_IMAGE_SIZE,
    batch_size=1,
    image_size=int(INFER_IMAGE_SIZE),
    include_background_label=False,
    advprop_preprocessing=False,)

print(f"Input image:")
display.display(display.Image(image_file, width=INFER_IMAGE_SIZE))

print(f"MaxViT prediction:")
pred_idx, pred_prob = eval_driver.eval_example_images(
    CKPT_DIR, image_files, labels_map_file)
```

```
--2023-12-14 21:25:29--  https://upload.wikimedia.org/wikipedia/commons/f/fe/Gia
nt_Panda_in_Beijing_Zoo_1.JPG
Resolving upload.wikimedia.org (upload.wikimedia.org)… 103.102.166.240,
2001:df2:e500:ed1a::2:b
Connecting to upload.wikimedia.org
(upload.wikimedia.org)|103.102.166.240|:443… connected.
HTTP request sent, awaiting response… 200 OK
Length: 116068 (113K) [image/jpeg]
Saving to: ‘panda.jpg’

panda.jpg           100%[===================>] 113.35K  --.-KB/s    in 0.004s

2023-12-14 21:25:29 (30.7 MB/s) - ‘panda.jpg’ saved [116068/116068]

Input image:
```



```
MaxViT prediction:
predicted class for image panda.jpg:
  -> top_0 (90.04%): giant panda, panda, panda bear, coon bear, Ailuropoda
melanoleuca
  -> top_1 (0.10%): lesser panda, red panda, panda, bear cat, cat bear, Ailurus
fulgens
  -> top_2 (0.08%): Arctic fox, white fox, Alopex lagopus
  -> top_3 (0.07%): American black bear, black bear, Ursus americanus, Euarctos
americanus
  -> top_4 (0.07%): sloth bear, Melursus ursinus, Ursus ursinus
```

6 Reference List

[1] Z. Tu, H. Talebi, H. Zhang, F. Yang, P. Milanfar, A. Bovik, and Y. Li, "Maxvit: Multi-axis vision transformer," in ECCV, pp. 459–479, 2022.

[2] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel,

"Backpropagation applied to hand-written zip code recognition," Neural Computation, vol. 1, no. 4, pp. 541–551, 1989.

[3] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An image is worth 16x16 words: Transformers for image recognition at scale," in ICLR, 2021.

[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in NeurIPS, pp. 84–90, 2012.

[5] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative Adversarial Nets," in NeurIPS, pp. 2672–2680, 2014.

[6] T. Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Doll´ar, and C. L. Zitnick, "Microsoft coco: Common objects in context," in ECCV, pp. 740–755, 2014.

[7] N. Murray, L. Marchesotti, and F. Perronnin, "Ava: A large-scale database for aesthetic visual analysis," in CVPR, pp. 2408–2415, 2012.

[8] X. Ding, X. Zhang, J. Han, and G. Ding, "Scaling Up Your Kernels to 31x31: Revisiting Large Kernel Design in CNNs," in CVPR, pp. 11963–11975, 2022.

[9] X. Ding, Y. Zhang, Y. Ge, S. Zhao, L. Song, X. Yue, and Y. Shan, "Unireplknet: A universal perception large-kernel convnet for audio, video, point cloud, time-series and image recognition," arXiv preprint arXiv:2311.15599, 2023.

[10] TorchVision maintainers and contributors, "TorchVision: PyTorch's Computer Vision library", GitHub repository, https://github.com/pytorch/vision/blob/v0.16.1/torchvision/models/maxvit.py#L333-L336, 2023.

[11] R. Wightman, "timm: Pytorch image models." GitHub repository, https://github.com/huggingface/pytorch-image-models/blob/v0.9.12/timm/models/maxxvit.py#L632-L665, 2023.

[12] J. Ho, N. Kalchbrenner, D. Weissenborn, and T. Salimans, "Axial attention in multidimensional transformers," arXiv preprint arXiv:1912.12180, 2019.

[13] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted Residuals and Linear Bottlenecks," in CVPR, pp. 4510–4520, 2018.

[14] M. Tan and Q. Le, "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks," in ICML, pp. 6105–6114, May 2019.

[15] M. Tan and Q. V. Le, "EfficientNetV2: Smaller Models and Faster Training," in ICML, pp. 10096–10106, 2021.

[16] Z. Liu, H. Mao, C.-Y. Wu, C. Feichtenhofer, T. Darrell, and S. Xie, "A ConvNet for the 2020s," in CVPR, pp. 11966–11976, 2022.

[17] A. Brohan, N. Brown, J. Carbajal, Y. Chebotar, J. Dabis, C. Finn, K. Gopalakrishnan, K. Hausman, A. Herzog, J. Hsu, et al., "Rt-1: Robotics transformer for real-world control at scale," arXiv preprint arXiv:2212.06817, 2022.

[18] H. Cai, J. Li, M. Hu, C. Gan, and S. Han, "EfficientViT: Lightweight Multi-Scale Attention for On-Device Semantic Segmentation," in ICCV, pp. 17302–17313, 2023.

[19] J. Hu, L. Shen, and G. Sun, “Squeeze-and-excitation networks,” in CVPR, pp. 7132–7141, 2018.

[20] X. Chu, Z. Tian, B. Zhang, X. Wang, and C. Shen, “Conditional positional encodings for vision transformers,” in ICLR, 2023.

[21] Z. Dai, H. Liu, Q. V. Le, and M. Tan, “CoAtNet: Marrying Convolution and Attention for All Data Sizes,” in NeurIPS, pp. 3965–3977, 2021.

[22] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo, “Swin transformer: Hierarchical vision transformer using shifted windows,” in ICCV, pp. 10012–10022, 2021.

[23] P. Wang (lucidrains), “vit-pytorch”, GitHub repository, https://github.com/lucidrains/vit-pytorch/blob/1.6.4/vit_pytorch/max_vit.py#L148, 2023.

[24] I. Loshchilov and F. Hutter, “Decoupled weight decay regularization,” in ICLR, 2019.

[25] S. Mehta and M. Rastegari, “MobileViT: Light-weight, General-purpose, and Mobile-friendly Vision Transformer,” in ICLR, 2022.

[26] H. Zhang, W. Hu, and X. Wang, “Fcaformer: Forward cross attention in hybrid vision transformer,” in ICCV, pp. 6060–6069, 2023.

[27] X. Liu, H. Peng, N. Zheng, Y. Yang, H. Hu, and Y. Yuan, “EfficientViT: Memory Efficient Vision Transformer with Cascaded Group Attention,” in CVPR, pp. 14420–14430, 2023.

[28] P. K. A. Vasu, J. Gabriel, J. Zhu, O. Tuzel, and A. Ranjan, “FastViT: A Fast Hybrid Vision Transformer using Structural Reparame- terization,” in ICCV, pp. 5785–5795, 2023.

[29] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is All you Need,” in NeurIPS, pp. 6000–6010, 2017.

[30] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning internal representations by error propagation,” Neural Computation, pp. 318–362, 1986.

[31] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” Neural Computation, vol. 9, no. 8, pp. 1735–1780, 1997.

[32] K. Cho, B. van Merri¨enboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, “Learning phrase representations using RNN encoder–decoder for statistical machine translation,” in EMNLP, pp. 1724–1734, 2014.

[33] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, “Scaling laws for neural language models,” arXiv preprint arXiv:2001.08361, 2020.

[34] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever, et al., “Improving language understanding by generative pre-training,” OpenAI, 2018.

[35] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, et al., “Language models are unsupervised multitask learners,” OpenAI, 2019.

[36] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al., “Language models are few-shot learners,” in NeurIPS, pp. 1877–1901, 2020.

[37] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of deep bidirectional transformers for language understanding,” in NAACL, pp. 4171–4186, 2019.

[38] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozi‘ere, N. Goyal, E. Hambro, F. Azhar, et al., “Llama: Open and efficient foundation language models,” arXiv preprint arXiv:2302.13971, 2023.

[39] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, et al., “Llama 2: Open foundation and fine-tuned chat models,” arXiv preprint arXiv:2307.09288, 2023

[40] T. Lin, Y. Wang, X. Liu, and X. Qiu, “A survey of transformers,” AI Open, vol. 3, pp. 111–132, 2022.

[41] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in CVPR, pp. 770–778, 2016.

[42] R. K. Srivastava, K. Greff, and J. Schmidhuber, “Training very deep networks,” in NeurIPS, pp. 2377–2385, 2015.

[43] J. Ba, J. R. Kiros, and G. E. Hinton, “Layer normalization,” arXiv preprint arXiv:1607.06450, 2016.

[44] R. Xiong, Y. Yang, D. He, K. Zheng, S. Zheng, C. Xing, H. Zhang, Y. Lan, L. Wang, and T. Liu, “On layer normalization in the transformer architecture,” in ICML, pp. 10524–10533, 2020.

[45] K. He, X. Chen, S. Xie, Y. Li, P. Doll´ar, and R. Girshick, “Masked Autoencoders Are Scalable Vision Learners,” in CVPR, pp. 16000– 16009, 2021.

[46] D. Hendrycks and K. Gimpel, “Gaussian error linear units (gelus),” arXiv preprint arXiv:1606.08415, 2016.

[47] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, “Mobilenets: Efficient convolutional neural networks for mobile vision applications,” arXiv preprint arXiv:1704.04861, 2017.

[48] T.-Y. Lin, P. Doll´ar, R. Girshick, K. He, B. Hariharan, and S. Belongie, “Feature pyramid networks for object detection,” in CVPR, pp. 936–944, 2017.

[49] Z. Liu, H. Hu, Y. Lin, Z. Yao, Z. Xie, Y. Wei, J. Ning, Y. Cao, Z. Zhang, L. Dong, F. Wei, and B. Guo, “Swin transformer v2: Scaling up capacity and resolution,” in CVPR, pp. 12009–12019, 2022.

[50] C. Sun, A. Shrivastava, S. Singh, and A. Gupta, “Revisiting unreasonable effectiveness of data in deep learning era,” in ICCV, pp. 843–852, 2017.