浅谈CrossFormer

> 论文名称: CROSSFORMER: A VERSATILE VISION TRANSFORMER BASED ON CROSS-SCALE ATTENTION
> 论文链接: https://arxiv.org/pdf/2108.00154.pdf
> 论文代码：https://github.com/cheerss/CrossFormer

1. 出发点

Transformers模型在处理视觉任务方面已经取得了很大的进展。然而，现有的vision transformers仍然不具备一种对视觉输入很重要的能力：**在不同尺度的特征之间建立注意力。**

- 每层的输入嵌入都是等比例的，没有跨尺度的特征；
- 一些transformers模型为了减少self-attention的计算量，衰减了key和value的部分特征表达。

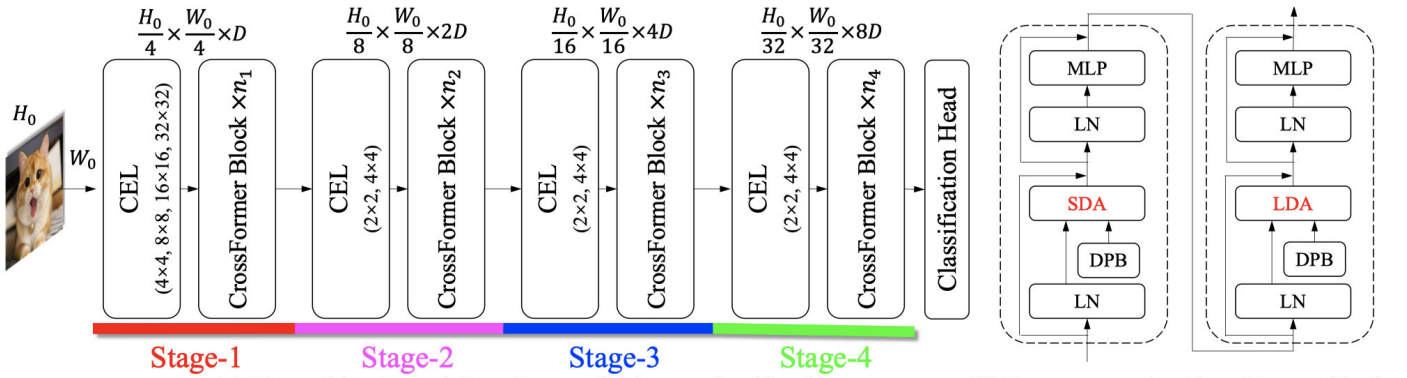
2. 怎么做

为了解决上面的问题，提出了几个模块。

1. Cross-scale Embedding Layer (CEL)
2. Long Short Distance Attention (LSDA)
3. Dynamic Position Bias (DPB)

这里1和2都是为了弥补了以往架构在建立跨尺度注意力方面的缺陷，3的话和上面的问题无关，是为了使相对位置偏差更加灵活，更好的适合不定尺寸的图像和窗口。这篇文章还挺讲究，不仅提出两个模块来解决跨尺度特征attention，还附送了一个模块来搞一个搞位置编码。

3. 模型结构



(a) The architecture of CrossFormer for image classification. (b) Two consecutive CrossFormer blocks.

模型整体的结构图如上所示，与swin-transformers和pvt基本整体结构一致，都是采用了层级的结构，这样的好处是可以迁移到dense任务上去，做检测，分割等。
整体结构由以下组成：

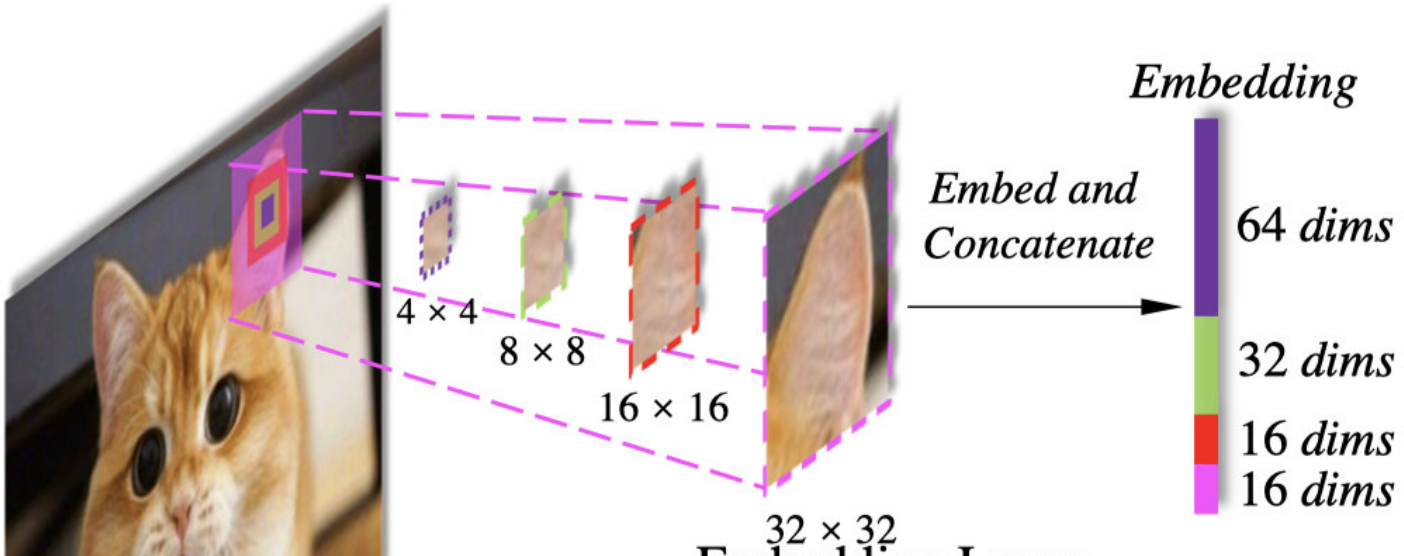
1. Cross-scale embeeding layer (CEL) , 用来做patch embeeding和patch merging(下采样)。
2. CrossFrom block， 看上图(b)，整体看是两个transformer结构的block所组成，其中第一个transformer block采用的是SDA，也就是short distance attention,并且引入了一个DPB模块，第二个transformer block采用的则是LDA，也就是long distance attention，同样也引入了一个DPB模块，两个transformer block串行，组成一个CrossFormer block。
3. Classification Head， 就是常规的分类MLP，没啥可说的。

3.1 Cross-scale embeeding layer (CEL)

Q&A

Question:既然是层级结构，那么就一定会有尺度上的下采样，那crossformers是怎么做的呢?
Answer: 简单回顾一下pvt和swin的做法
pvt: 假设feature map为$B \times H \times W \times C_1$, 那么我们就可以做一个stride为2的一个convolution, 变换为$B \times H//2 \times W//2 \times C_2$，由于patchsize固定，所以，featuremap下采样，对应的就是token的下采样。
swin: swin由于是基于windows做attention，为了达到下采样的效果，选择直接对featuremap上采样，每个4邻域都会分别采样到另一个map里面去，最后则有$B \times H \times W \times C_1$变换为$B \times H//2 \times w//2 \times C_2$，也可以看做是stride为2带有空洞的卷积操作。

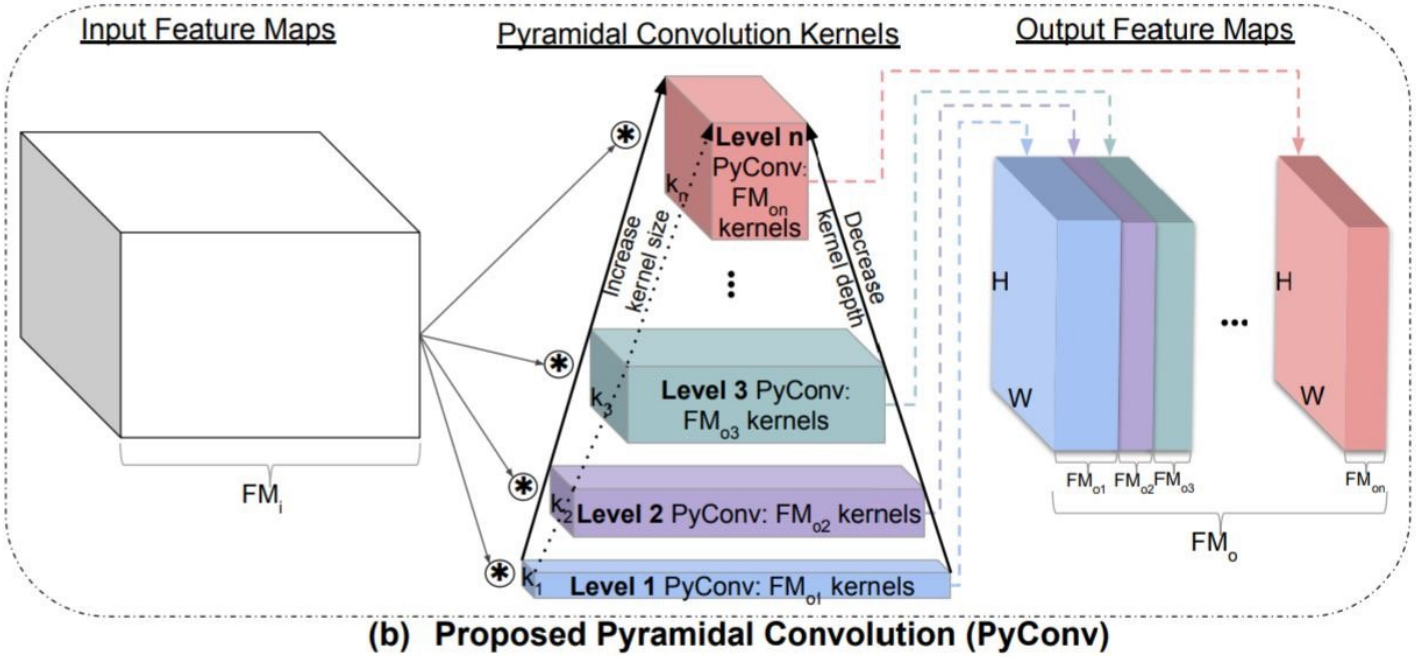
Question: 万变不离其宗，所以为了达到下采样的效果，用*卷积*其实就可以了。那么CrossFormer为了实现下采样是怎么做的呢?
Answer:



Embedding Layer

Type	Kernel	Stride	Dim
Conv.	4×4	4×4	$\frac{D}{2}$
Conv.	8×8	4×4	$\frac{D}{4}$
Conv.	16×16	4×4	$\frac{D}{8}$
Conv.	32×32	4×4	$\frac{D}{8}$

看上图，很明显，直接用不同卷积核来对输入的图片做卷积，得到卷积后的结果，直接concat一起，作为我们的patch embeeding。想法很简单，实现的话也很朴素，通过不同卷积核的卷积，来获取不同尺度特征的信息，对于变化尺度的物体相对来说是比较友好的，这个可行性其实在很多paper里面都有用到过，比如**Pyramidal Convolution**, 如下图所示。



(b) Proposed Pyramidal Convolution (PyConv)

ps: 这里除了patch embeeding，也就是第一个CEL用的是4个卷积核stride为4来做多尺度，其余的CEL也就是patch merge用的都是2个卷积核stride为2来做的多尺度。两个操作基本相同，只看一份代码即可，核心代码如下:

```
class PatchEmbed(nn.Module):
    def __init__(self, img_size=224, patch_size=[4], in_chans=3, embed_dim=96, norm_layer=None):
        super().__init__()
        ...

        self.projs = nn.ModuleList()
        for i, ps in enumerate(patch_size):
            if i == len(patch_size) - 1:
                dim = embed_dim // 2 ** i
            else:
                dim = embed_dim // 2 ** (i + 1)
            stride = patch_size[0]
            padding = (ps - patch_size[0]) // 2
            self.projs.append(nn.Conv2d(in_chans, dim, kernel_size=ps, stride=stride, padding=pa
        if norm_layer is not None:
            self.norm = norm_layer(embed_dim)
        else:
            self.norm = None

    def forward(self, x):
        B, C, H, W = x.shape
        # FIXME look at relaxing size constraints
        assert H == self.img_size[0] and W == self.img_size[1], \
            f"Input image size ({H}*{W}) doesn't match model ({self.img_size[0]}*{self.img_size[
        xs = []
        for i in range(len(self.projs)):
            tx = self.projs[i](x).flatten(2).transpose(1, 2)
            xs.append(tx)  # B Ph*Pw C
        x = torch.cat(xs, dim=2)
        if self.norm is not None:
            x = self.norm(x)
        return x
```

代码做了两件事情:

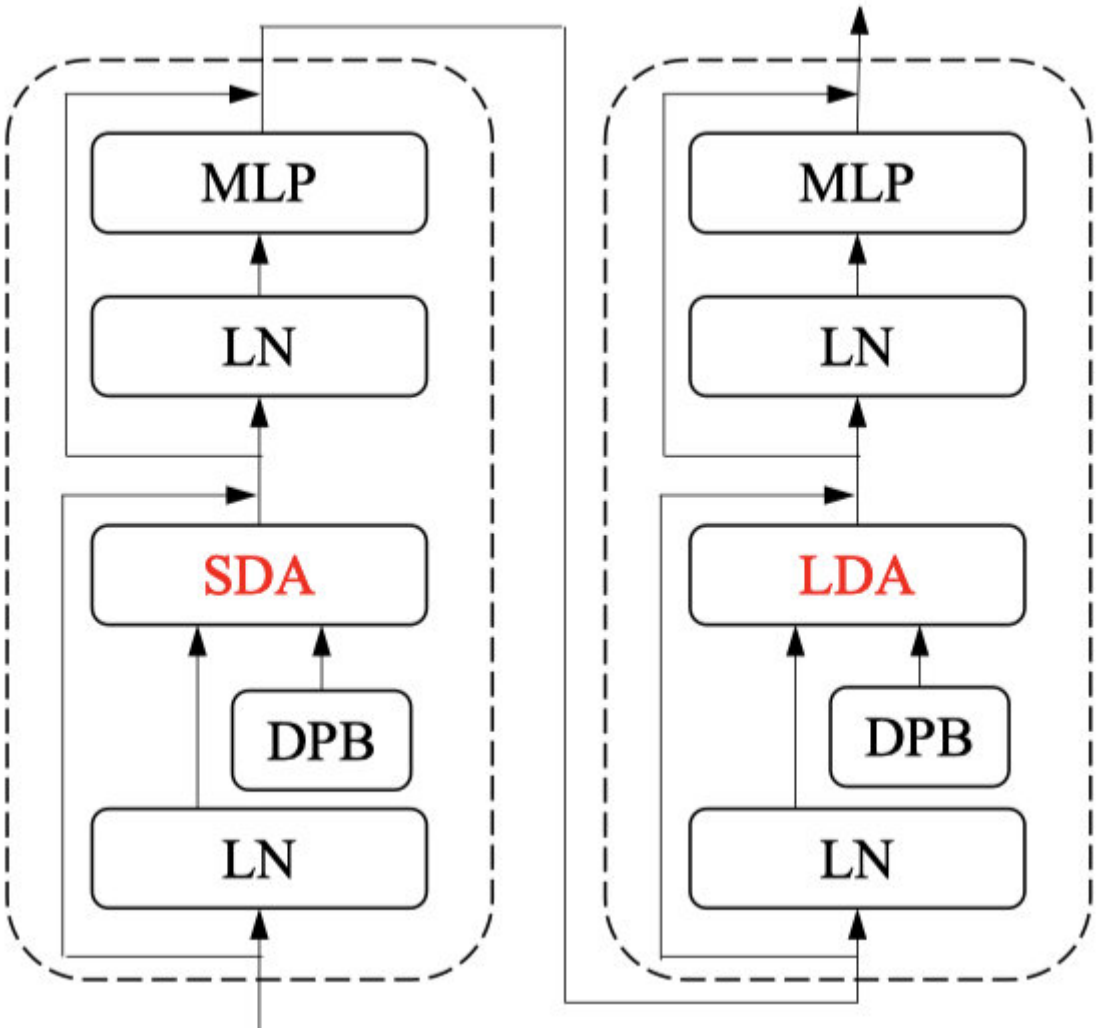
- 初始化几个不同kernel，不同padding，相同stride的conv
- 对输入进行卷积操作后得到的feature，做concat

这样, 以输入为224x224为例, 我们通过patch embeeding, 得到了一个56x56的featuremap，输入到第一个stage，输出继续做一个patchmerging，得到了一个28x28的featuremap，输入到第二个stage，输出继续做一个patchmerging，得到了一个14x14的featuremap, 输入到第三个stage, 输出再次做一个patchmerging，得到一个7x7的featuremap，在输入到最后一个stage，最后的输出做分类即可，基本上都是这么一个套路了，大同小异。那么stage里面是怎么做的，看下一节。

3.2 Stage block

对于标准的transformerblock来说，假设输入为$B \times N \times L$, 经过transformer后，我们的输出还是$B \times N \times L$，输入和输出是没有变化的，唯一的尺度变换都在patch embeeding和patch merging。那么我们在改动transformer block的时候，也是要遵守这一原则，对应的，如果想有resolution上的变化，那么就要借助于reshape或者view等操作，好了，不说废话，看这篇文章的crossformer block是怎样的。

CrossFormer Block



CrossFormer block由两个transformer的block堆叠而成，两个transformer block的self-attention都是基于windows来做的，不同之处在于一个考虑的是局部内的信息，一个则是考虑的是全局的信息。这个思想并没有什么突出的地方，目前来说transformer做局部和全局的串联，已经屡见不鲜。

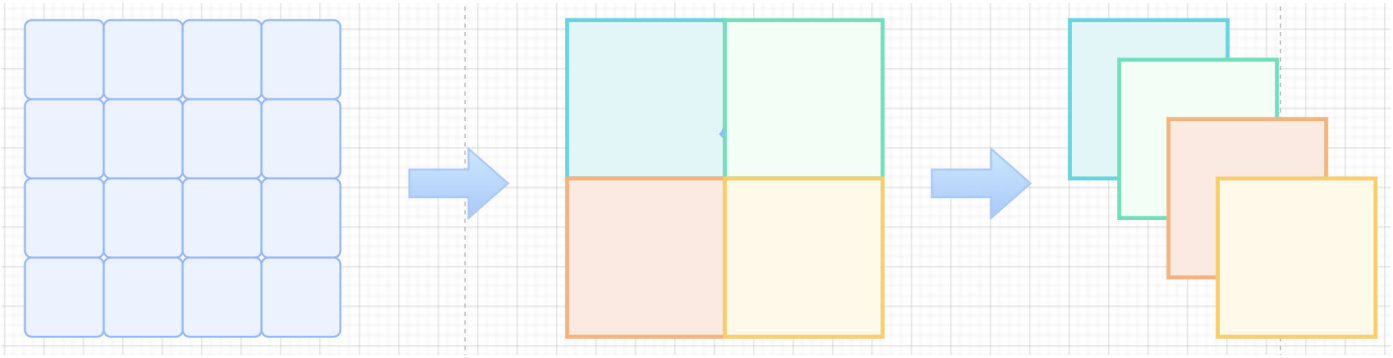
Q&A

Question: 问题来了，怎样实现呢，既要保证基于windows做self-attention，又想要全局的信息?
Answer: 使用一个固定的步长step，比如2或者3，对行和列分别按步长采样，这样可以得到多个全局的信息，同时基于一个$\frac{H}{step} \times \frac{W}{step}$大小的windows。这样最大可能的利用到了featuremap的全局性，同时

节省了计算的复杂度，假设输入为$S \times S$，step为I，那么windows的窗口大小为$G \times G, G = \frac{S}{I}$，原始的复杂度为$O(S^4)$, 那么基于窗口的attention的复杂度为$O(G^4) = O(G^2(\frac{S}{I})^2) == O(G^2S^2), G < S$。

CrossFormerblock中的基石: windows self-attention

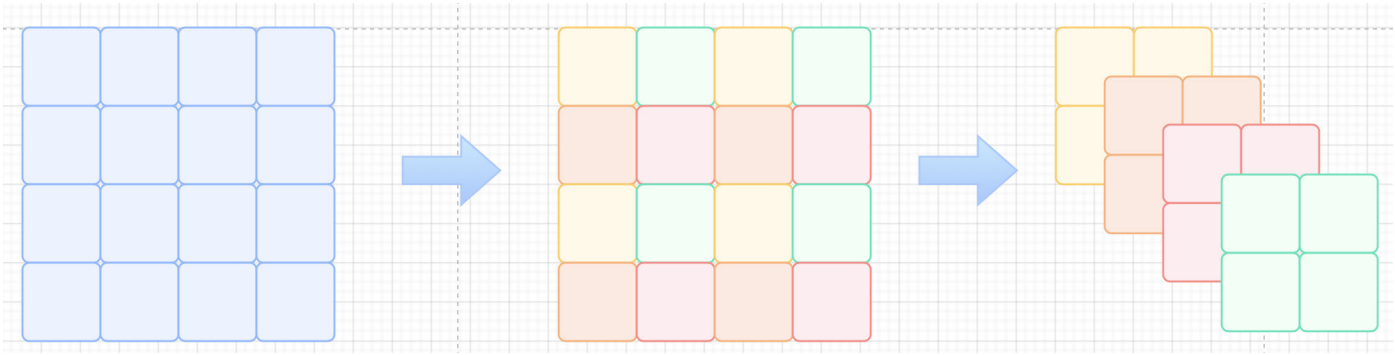
- **Short Distance Attention(SDA)**



对于一个$4 \times 4 \times C$的feautremap，如果我们想要实现self-attention, 需要先转换为$(4 \times 4) \times C$的向量，那么这里就是所谓的long-range的attention，也就是全局的。但是对于MHA来说，部分head还是更多的focus到short-range，结合swin和twins的结论可以验证，局部attention不仅可以达到很好的效果同时还会节省计算。那么怎么获得局部的attention，很简单，如上图所示，只需要把原始的$4 \times 4 \times C$做reshape操作, 既可以得到$4 \times (2 \times 2) \times C$，那么我们只需要对4个$2 \times 2$做attention即可，最后在reshape回原始形状，代码如下:

```
x = x.reshape(B, H // G, G, W // G, G, C).permute(0, 1, 3, 2, 4, 5)
x = attention(x)
....
```

- **Long Distance Attention(LDA)**



从上面的SDA, 我们得到了局部attention，但是也说了，部分head是局部友好的，也就是说，对于self-attention来说，long-range始终是必不可少的，所以还是需要引入long distance attention。如上图所示，颜色一致的部分表示的是归属于同一个sub-windows的，对于原始的$4 \times 4 \times C$，使用step为2进行采样，得到了4个$2 \times 2 \times C$, 可以抽象成两种计算方法，一种是空洞卷积，一种则是

1x1的卷积，stride为step，对于图像来说，相邻的位置，像素所表达的信息接近，所以两种得到的都是全局的一个感受野，所以对应我们的attention，也会得到一个近乎全局的attention，代码如下:

```
x = x.reshape(B, G, H // G, G, W // G, C).permute(0, 2, 4, 1, 3, 5)
x = attention(x)
...
```

直接看这个代码可能不太好理解，我们用 `einops` 简单改写一下，代码如下:
输入:

```
x[0,:,:,0]
tensor([[ 1,  2,  3,  4],
        [ 2,  4,  6,  8],
        [ 3,  6,  9, 12],
        [ 4,  8, 12, 16]])
x.shape
torch.Size([1, 4, 4, 1])
```

SDA:

```
a1 = rearrange(x, ' b (h g1) (w g2) c -> b h w g1 g2 c ', g1=2, g2=2)
a1[0,:,:,:,:,0]
tensor([[[[ 1,  2],
          [ 2,  4]],

         [[ 3,  4],
          [ 6,  8]]],


        [[[ 3,  6],
          [ 4,  8]],

         [[ 9, 12],
          [12, 16]]]])
```

对于SDA的情况，实际上就是循环HW，扣2x2的区域下来，那么因为有行遍历优先，或者列遍历优先，实际上得到的结果是顺序的。
LDA:

```
a2 = rearrange(x, ' b (g1 h) (g2 w) c -> b h w g1 g2 c ', g1=2, g2=2)
a2[0,:,:,:,:,0]
tensor([[[[ 1,  3],
          [ 3,  9]],

         [[ 2,  4],
          [ 6, 12]]],


        [[[ 2,  6],
          [ 4, 12]],

         [[ 4,  8],
          [ 8, 16]]]])
```

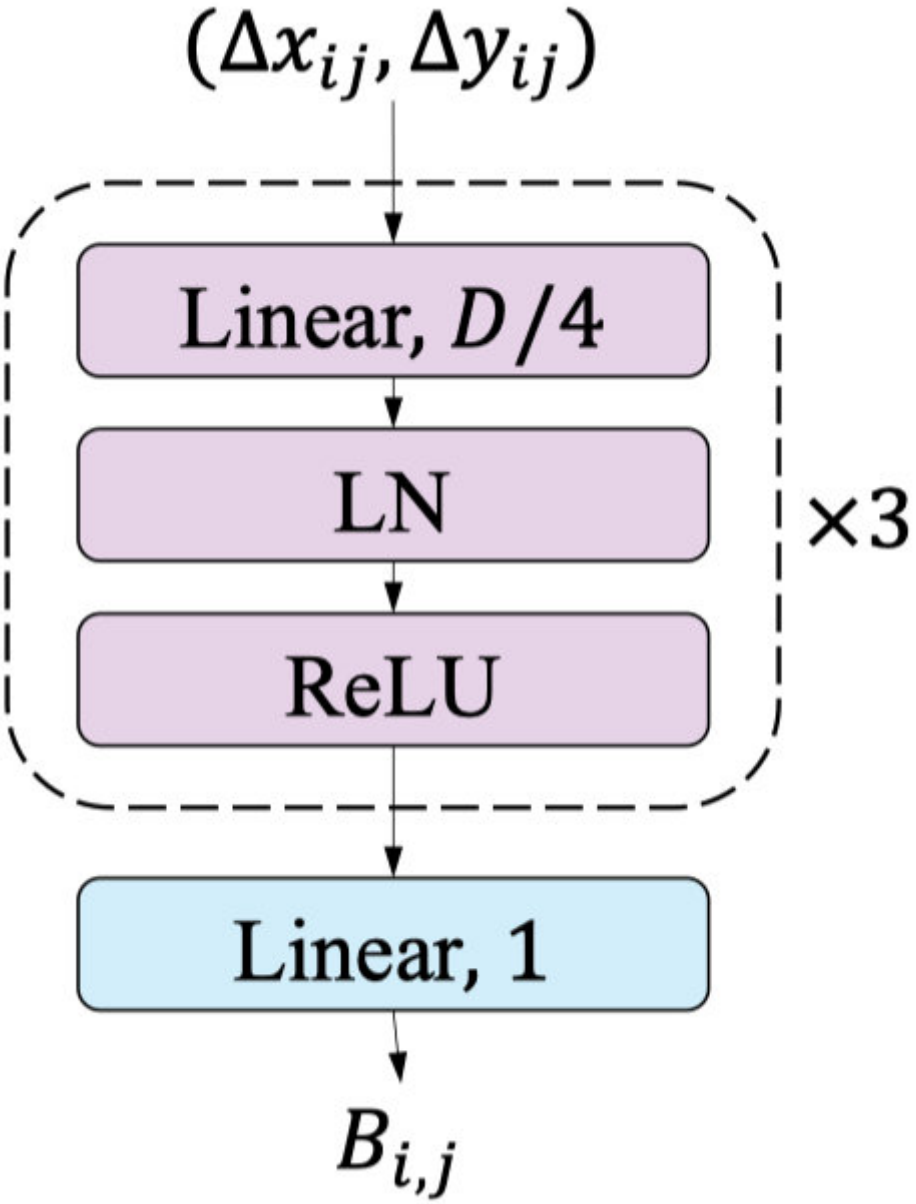
那么对于LDA的情况，我们希望的是外循环是有间隔的，所以把step放到HW的外面，这样循环的时候则是按间隔来进行sample，以达到全局的效果。

CrossFormerblock中的位置编码: Dynamic Position Bias(DPB)

- **Relative position bias (RPB)**
  随着位置编码技术的不断发展，相对位置编码偏差逐渐的应用到了transformers中，很多的vision transformers均采用RPB来替换原始的APE，好处是可以直接插入到我们的attention中，不需要很繁琐的公式计算，并且可学习性高，鲁棒性强，公式如下:

$$Attention = Softmax(QK^T/\sqrt{d} + B)V$$

  Q&A
  Question:但是这里有个问题，对于$Q, K, V \in \mathbb{R}^{G^2 \times D}$来说，会有一个偏差$B \in R^{G^2 \times G^2}$, B所表达的则是matrix上的i和j的相对位置的embeeding，很显然，如果图像的尺寸变化，那么可能会超出B所表达的范围，会导致PE没有作用，那么要怎么改进呢?
  Answer:很简单，插值或者切片不就好了，但是切片会导致pe完整性差，损失信息，插值是通过原始的位置信息来模拟出来信息，实际上还是原始的信息，没有信息收益。那本文想到的一个方法就是可以通过学习得到位置信息。
- **Dynamic Position Bias (DPB)**



举个栗子，如果我们的窗口大小为7×7, 那么我们希望的相对位置范围为$x \in [-6, 6]$假设我们不考虑截断距离，如果我们的窗口突然放大到了9×9，那么我们实际的相对位置所表达的信息只是中间的一部分窗口，失去了对外层数据位置的访问。DPB的思想则是，我们不希望通过用实际的相对位置来做embeeidng，而是希望通过隐空间先对位置偏差进行学习，如上图所示。

DPB，由3个线性层+LayerNorm+ReLU组成的block堆叠而成，最后接一个输出为1的线性层做bias的表征，输入是$(N, 2)$，由于self-attention是由多个head组成的，所以输出为$(N, 1 \times heads)$，代码如下：

1. 先得到一个相对位置偏差的矩阵，假设group_size的大小为7×7，那么bias的维度为$(169, 2)$

```
self.pos = DynamicPosBias(self.dim // 4, self.num_heads, residual=False)

# generate mother-set
position_bias_h = torch.arange(1 - self.group_size[0], self.group_size[0])
position_bias_w = torch.arange(1 - self.group_size[1], self.group_size[1])
biases = torch.stack(torch.meshgrid([position_bias_h, position_bias_w]))  # 2, 2Wh-1, 2
biases = biases.flatten(1).transpose(0, 1).float()
self.register_buffer("biases", biases)

biases:
tensor([[-6., -6.],
[-6., -5.],
[-6., -4.],
[-6., -3.],
[-6., -2.],
[-6., -1.],
...
[ 6.,  4.],
[ 6.,  5.],
[ 6.,  6.]])
```

2. 构建索引矩阵, 得到了一个49×49的一个索引，从右上角为0开始，向左和向下递增。

```
coords_h = torch.arange(self.group_size[0])
coords_w = torch.arange(self.group_size[1])
coords = torch.stack(torch.meshgrid([coords_h, coords_w]))  # 2, Wh, Ww
coords_flatten = torch.flatten(coords, 1)  # 2, Wh*Ww
relative_coords = coords_flatten[:, :, None] - coords_flatten[:, None, :]  # 2, Wh*Ww,
relative_coords = relative_coords.permute(1, 2, 0).contiguous()  # Wh*Ww, Wh*Ww, 2
relative_coords[:, :, 0] += self.group_size[0] - 1  # shift to start from 0
relative_coords[:, :, 1] += self.group_size[1] - 1
relative_coords[:, :, 0] *= 2 * self.group_size[1] - 1
relative_position_index = relative_coords.sum(-1)  # Wh*Ww, Wh*Ww
self.register_buffer("relative_position_index", relative_position_index)

relateive_position_index:
tensor([[ 84,  83,  82,  ...,   2,   1,   0],
[ 85,  84,  83,  ...,   3,   2,   1],
[ 86,  85,  84,  ...,   4,   3,   2],
...,
[166, 165, 164,  ...,  84,  83,  82],
[167, 166, 165,  ...,  85,  84,  83],
[168, 167, 166,  ...,  86,  85,  84]])
```

3. 初始化DBP模块

```
pos = DynamicPosBias(64 // 4, 8, residual=False)
```

4. 通过DBP生成bias的embeeding，通过索引矩阵进行取值，最后与attn相加

```
pos = self.pos(self.biases) # 2Wh-1 * 2Ww-1, heads
# select position bias
relative_position_bias = pos[self.relative_position_index.view(-1)].view(
    self.group_size[0] * self.group_size[1], self.group_size[0] * self.group_size[1], -
relative_position_bias = relative_position_bias.permute(2, 0, 1).contiguous()  # nH, Wh
attn = attn + relative_position_bias.unsqueeze(0)
```

- **Rethinking**: 对于PE来说，目前的形成方法都是通过embeeding来构建bias矩阵，对于VIT来说，直接使用绝对位置的embeeding，通过学习来更新，对于swins来说，直接使用embeeding而不是相对bias的值，相当于$embeeding_{bias} == DBP(bias)$，其实本质上没有太大的差异, 从消融实验结果上来看，DBP和RBP的性能一样。唯一的作用，就是embeeding是后验而不是先验，对于变换的尺寸来说，可能更加友好，只不过这个paper里面没有给出结论，还需要更多的实验来验证。

Method	#Params/FLOPs	Throughput	Acc.
APE	30.9342M/4.9061G	686 imgs/sec	82.1%
RPB	30.6159M/4.9062G	684 imgs/sec	**82.5%**
DPB	30.6573M/4.9098G	672 imgs/sec	**82.5%**
DPB-residual	30.6573M/4.9098G	672 imgs/sec	82.4%

综上，我们每个stageblock里面，都是由SDA+DBP&LDA+DBP堆叠而成，与swin类似，奇数layer走SDAblock，偶数layer走LDAblock，从结构上来看，先局部attention，再全局attention，有一点点由点到面的既视感。

	Output Size	Layer Name	CrossFormer-T	CrossFormer-S	CrossFormer-B	CrossFormer-L
Stage-1	56×56 $(S_1 = 56)$	Cross Embed.	Kernel size: 4×4, 8×8, 16×16, 32×32, Stride=4			
		SDA/LDA MLP	$\begin{bmatrix} D_1 = 64 \\ H_1 = 2 \\ G_1 = 7 \\ I_1 = 8 \end{bmatrix} \times 1$	$\begin{bmatrix} D_1 = 96 \\ H_1 = 3 \\ G_1 = 7 \\ I_1 = 8 \end{bmatrix} \times 2$	$\begin{bmatrix} D_1 = 96 \\ H_1 = 3 \\ G_1 = 7 \\ I_1 = 8, \end{bmatrix} \times 2$	$\begin{bmatrix} D_1 = 128 \\ H_1 = 4 \\ G_1 = 7 \\ I_1 = 8 \end{bmatrix} \times 2$
Stage-2	28×28 $(S_2 = 28)$	Cross Embed.	Kernel size: 2×2, 4×4, Stride=2			
		SDA/LDA MLP	$\begin{bmatrix} D_2 = 128 \\ H_2 = 4 \\ G_2 = 7 \\ I_2 = 4 \end{bmatrix} \times 1$	$\begin{bmatrix} D_2 = 192 \\ H_2 = 6 \\ G_2 = 7 \\ I_2 = 4 \end{bmatrix} \times 2$	$\begin{bmatrix} D_2 = 192 \\ H_2 = 6 \\ G_2 = 7 \\ I_2 = 4 \end{bmatrix} \times 2$	$\begin{bmatrix} D_2 = 256 \\ H_2 = 8 \\ G_2 = 7 \\ I_2 = 4 \end{bmatrix} \times 2$
Stage-3	14×14 $(S_3 = 14)$	Cross Embed.	Kernel size: 2×2, 4×4, Stride=2			
		SDA/LDA MLP	$\begin{bmatrix} D_3 = 256 \\ H_3 = 8 \\ G_3 = 7 \\ I_3 = 2 \end{bmatrix} \times 8$	$\begin{bmatrix} D_3 = 384 \\ H_3 = 12 \\ G_3 = 7 \\ I_3 = 2 \end{bmatrix} \times 6$	$\begin{bmatrix} D_3 = 384 \\ H_3 = 12 \\ G_3 = 7 \\ I_3 = 2 \end{bmatrix} \times 18$	$\begin{bmatrix} D_3 = 512 \\ H_3 = 16 \\ G_3 = 7 \\ I_3 = 2 \end{bmatrix} \times 18$
Stage-4	7×7 $(S_4 = 7)$	Cross Embed.	Kernel size: 2×2, 4×4, Stride=2			
		SDA/LDA MLP	$\begin{bmatrix} D_4 = 512 \\ H_4 = 16 \\ G_4 = 7 \\ I_4 = 1 \end{bmatrix} \times 6$	$\begin{bmatrix} D_4 = 768 \\ H_4 = 24 \\ G_4 = 7 \\ I_4 = 1 \end{bmatrix} \times 2$	$\begin{bmatrix} D_4 = 768 \\ H_4 = 24 \\ G_4 = 7 \\ I_4 = 1 \end{bmatrix} \times 2$	$\begin{bmatrix} D_4 = 1024 \\ H_4 = 32 \\ G_4 = 7 \\ I_4 = 1 \end{bmatrix} \times 2$
Head	1×1	Avg Pooling	Kernel size: 7×7			
		Linear	Classes: 1000			

与其他的paper大同小异了，设计了4种不同FLOPs的模型，Tiny, Small, Big和Large 用来和其他的模型在同等FLOPs下公平比较。D表示的是维度，H表示的是attention头的个数，G表示的是attention窗口的大小，I表示的是滑动窗口的间隔。

4. 实验结果

Table 2: Results on ImageNet validation set. The input size is 224×224 for most models, while is 384×384 for the model with a †. Results of other architectures are drawn from their papers.

Architectures	#Params	FLOPs	Acc.
PVT-S	24.5M	3.8G	79.8%
RegionViT-T	13.8M	2.4G	80.4%
Twins-SVT-S	24.0M	2.8G	81.3%
CrossFormer-T	27.8M	2.9G	**81.5%**
DeiT-S	22.1M	4.6G	79.8%
T2T-ViT	21.5M	5.2G	80.7%
CViT-S	26.7M	5.6G	81.0%
PVT-M	44.2M	6.7G	81.2%
TNT-S	23.8M	5.2G	81.3%
Swin-T	29.0M	4.5G	81.3%
NesT-T	17.0M	5.8G	81.5%
CvT-13	20.0M	4.5G	81.6%
ResT	30.2M	4.3G	81.6%
CAT-S	37.0M	5.9G	81.8%
ViL-S	24.6M	4.9G	81.8%
RegionViT-S	30.6M	5.3G	**82.5%**
CrossFormer-S	30.7M	4.9G	**82.5%**

Architectures	#Params	FLOPs	Acc.
BoTNet-S1-59	33.5M	7.3G	81.7%
PVT-L	61.4M	9.8G	81.7%
DeiT-B	86.0M	17.5G	81.8%
CvT-21	32.0M	7.1G	82.5%
CAT-B	52.0M	8.9G	82.8%
Swin-S	50.0M	8.7G	83.0%
RegionViT-M	41.2M	7.4G	83.1%
Twins-SVT-B	56.0M	8.3G	83.1%
NesT-S	38.0M	10.4G	83.3%
CrossFormer-B	52.0M	9.2G	**83.4%**
DeiT-B†	86.0M	55.4G	83.1%
ViL-B	55.7M	13.4G	83.2%
RegionViT-B	72.0M	13.3G	83.3%
Twins-SVT-L	99.2M	14.8G	83.3%
Swin-B	88.0M	15.4G	83.3%
NesT-B	68.0M	17.9G	83.8%
CrossFormer-L	92.0M	16.1G	**84.0%**

CrossFormer都是再224x224的图片大小下进行训练，使用的类似DeiT的训练策略，不过采用了更大的warmup(20个，DeiT是5), 学习率为1e-3, weightdecay为5e-2, 与DeiT不同的是，这里随着模型大小的改变，分别采用了0.1，0.2，0.3，0.5的drop path rate。可以看到，在同等数量级的FLOPs的情况下，CF在imagenet上都取得了SOTA的效果。

Table 3: Object detection and instance segmentation results on COCO *val* 2017. Results for Swin (Liu et al., 2021b) are drawn from Twins (Chu et al., 2021) as Swin does not report results on RetinaNet and Mask-RCNN. Results in blue fonts are the second-placed one. CrossFormers with ‡ use different group size from classification models, as described in the appendix (A.3).

Method	Backbone	#Params	FLOPs	AP^b	AP^b_{50}	AP^b_{75}	AP^b_S	AP^b_M	AP^b_L
RetinaNet 1× schedule	ResNet-50	37.7M	234.0G	36.3	55.3	38.6	19.3	40.0	48.8
	CAT-B	62.0M	337.0G	41.4	62.9	43.8	24.9	44.6	55.2
	Swin-T	38.5M	245.0G	41.5	62.1	44.2	25.1	44.9	55.5
	PVT-M	53.9M	–	41.9	63.1	44.3	25.0	44.9	57.6
	ViL-M	50.8M	338.9G	42.9	64.0	45.4	27.0	46.1	57.2
	RegionViT-B	83.4M	308.9G	43.3	65.2	46.4	29.2	46.4	57.0
	TransCNN-B	36.5M	–	43.4	64.2	46.5	27.0	47.4	56.7
	CrossFormer-S	40.8M	282.0G	**44.4**(+1.0)	65.8	47.4	28.2	48.4	59.4
	CrossFormer-S‡	40.8M	272.1G	**44.2**(+0.8)	65.7	47.2	28.0	48.0	59.1
	ResNet101	56.7M	315.0G	38.5	57.8	41.2	21.4	42.6	51.1
	PVT-L	71.1M	345.0G	42.6	63.7	45.4	25.8	46.0	58.4
	Twins-SVT-B	67.0M	322.0G	44.4	66.7	48.1	28.5	48.9	60.6
	RegionViT-B+	84.5M	328.2G	44.6	66.4	47.6	29.6	47.6	59.0
	Swin-B	98.4M	477.0G	44.7	65.9	49.2	–	–	–
	Twins-SVT-L	110.9M	455.0G	44.8	66.1	48.1	28.4	48.3	60.1
	CrossFormer-B	62.1M	389.0G	**46.2**(+1.4)	67.8	49.5	30.1	49.9	61.8
	CrossFormer-B‡	62.1M	379.1G	**46.1**(+1.3)	67.7	49.0	29.5	49.9	61.5

Method	Backbone	#Params	FLOPs	AP^b	AP^b_{50}	AP^b_{75}	AP^m	AP^m_{50}	AP^m_{75}
Mask-RCNN 1× schedule	PVT-M	63.9M	–	42.0	64.4	45.6	39.0	61.6	42
	Swin-T	47.8M	264.0G	42.2	64.6	46.2	39.1	61.6	42.0
	Twins-PCPVT-S	44.3M	245.0G	42.9	65.8	47.1	40.0	62.7	42.9
	TransCNN-B	46.4M	–	44.0	66.4	48.5	40.2	63.3	43.2
	ViL-M	60.1M	261.1G	43.3	65.9	47.0	39.7	62.8	42.0
	RegionViT-B	92.2M	287.9G	43.5	66.7	47.4	40.1	63.4	43.0
	RegionViT-B+	93.2M	307.1G	44.5	67.6	48.7	41.0	64.4	43.9
	CrossFormer-S	50.2M	301.0G	**45.4**(+0.9)	68.0	49.7	**41.4**(+0.4)	64.8	44.6
	CrossFormer-S‡	50.2M	291.1G	**45.0**(+0.5)	67.9	49.1	**41.2**(+0.2)	64.6	44.3
	CAT-B	71.0M	356.0G	41.8	65.4	45.2	38.7	62.3	41.4
	PVT-L	81.0M	364.0G	42.9	65.0	46.6	39.5	61.9	42.5
	Twins-SVT-B	76.3M	340.0G	45.1	67.0	49.4	41.1	64.1	44.4
	ViL-B	76.1M	365.1G	45.1	67.2	49.3	41.0	64.3	44.2
	Twins-SVT-L	119.7M	474.0G	45.2	67.5	49.4	41.2	64.5	44.5
	Swin-S	69.1M	354.0G	44.8	66.6	48.9	40.9	63.4	44.2
	Swin-B	107.2M	496.0G	45.5	–	–	41.3	–	–
	CrossFormer-B	71.5M	407.9G	**47.2**(+1.7)	69.9	51.8	**42.7**(+1.4)	66.6	46.2
	CrossFormer-B‡	71.5M	398.1G	**47.1**(+1.6)	69.9	52.0	**42.7**(+1.4)	66.5	46.1

可以看到CrossFromer在coco2017上基于RetinaNet架构，也可以达到SOTA的效果，高于Twins模型1.4个ap之多。实例分割则是基于Mask-Rcnn的架构，也是SOTA，超过Swin 1.7个ap。相比而言参数量和FLOPs都更少，性能更好。

Table 4: Semantic segmentation results on ADE20K validation set. "MS IOU" means testing with variable input size.

Semantic FPN (80K iterations)			
Backbone	#Params	FLOPs	IOU
PVT-M	48.0M	219.0G	41.6
Twins-SVT-B	60.4M	261.0G	45.0
Swin-S	53.2M	274.0G	45.2
CrossFormer-S	34.3M	220.7G	**46.0**(+0.8)
CrossFormer-S‡	34.3M	209.8G	**46.4**(+1.2)
PVT-L	65.1M	283.0G	42.1
CAT-B	55.0M	276.0G	43.6
CrossFormer-B	55.6M	331.0G	**47.7**(+4.1)
CrossFormer-B‡	55.6M	320.1G	**48.0**(+4.4)
Twins-SVT-L	103.7M	397.0G	45.8
CrossFormer-L	95.4M	497.0G	**48.7**(+2.9)
CrossFormer-L‡	95.4M	482.7G	**49.1**(+3.3)

UPerNet (160K iterations)				
Backbone	#Params	FLOPs	IOU	MS IOU
Swin-T	60.0M	945.0G	44.5	45.8
Shuffle-T	60.0M	949.0G	46.6	47.6
CrossFormer-S	62.3M	979.5G	**47.6**(+1.0)	**48.4**
CrossFormer-S‡	62.3M	968.5G	**47.4**(+0.8)	**48.2**
Swin-S	81.0M	1038.0G	47.6	49.5
Shuffle-S	81.0M	1044.0G	48.4	49.6
CrossFormer-B	83.6M	1089.7G	**49.7**(+1.3)	**50.6**
CrossFormer-B‡	83.6M	1078.8G	**49.2**(+0.8)	**50.1**
Swin-B	121.0M	1088.0G	48.1	49.7
Shuffle-B	121.0M	1096.0G	49.0	–
CrossFormer-L	125.5M	1257.8G	**50.4**(+1.4)	**51.4**
CrossFormer-L‡	125.5M	1243.5G	**50.5**(+1.5)	**51.4**

语义分割上，可以看到可以看到最多提升3.3%的MIOU，非常厉害了。

(a) Ablation study for cross-scale embedding (CEL) and long short distance attention (LSDA). The base model is CrossFormer-S (82.5%).

PVT-like	Swin-like	LSDA	CEL	Acc.
✓			✓	**81.3%**
	✓		✓	**81.9%**
		✓	✓	**82.5%**
		✓		**81.5%**

消融实验上，可以看到，当CEL和LSDA一起使用的时候，性能最高。不过这实验也很明显了，CrossFormer参考了PVT和swin的设计思想。使用了LSDA，相比于Swin提升了0.6%个点，设计比swin更加朴实，不错的提升。

5. 结论

本文提出了一个新的transformers架构称为CrossFormer。其核心设计包括一个跨尺度嵌入层（CEL）和长短距离注意（LSDA）模块。此外，我们提出了动态位置偏置（DPB），使相对位置偏置适用于任何输入尺寸。实验表明，CrossFormer在几个有代表性的视觉任务上取得了SOTA。特别是，CrossFormer在检测和分割方面有很大的改进，这表明跨尺度嵌入和LSDA对于密集预测的视觉任务特别重要。

> ps: 欢迎大家关注我的知乎：https://www.zhihu.com/people/flyegle