浅谈cswin-transformers

> 论文链接:https://arxiv.org/abs/2107.00652
> 论文代码:https://github.com/microsoft/CSWin-Transformer

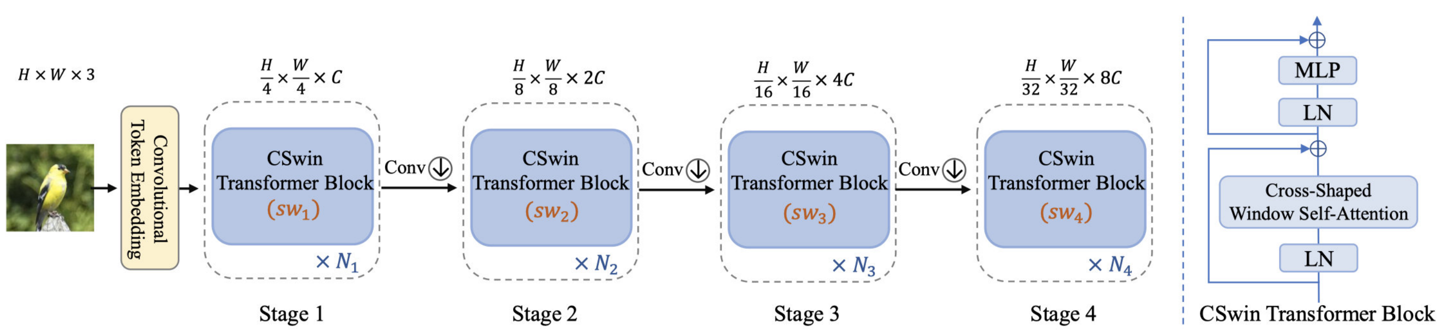
1. 出发点

- 基于global attention的transformer效果虽然好但是计算量太大了。
- 基于local attention的transformer的会限制每个token的感受野的交互，减缓感受野的增长。

2. 怎么做

- 提出了Cross-Shaped Window self-attention机制，可以并行计算水平和竖直方向的self-attention，可以在更小的计算量条件下获得更好的效果。
- 提出了Locally-enhanced Positional Encoding(LePE), 可以更好的处理局部位置信息，并且支持任意形状的输入。

3. 模型结构



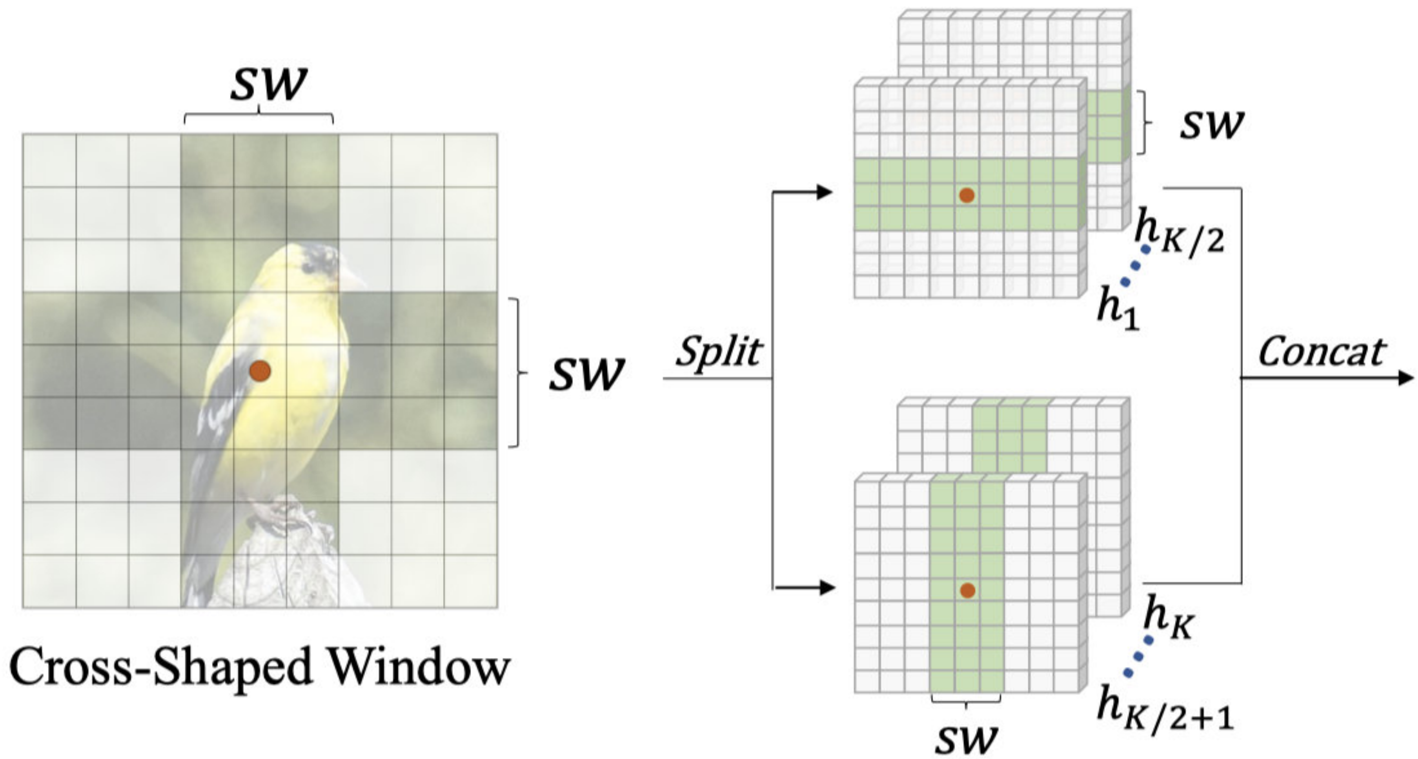
模型整体结构如上所示，由token embeeding layer和4个stageblock所堆叠而成，每个stage block后面都会接入一个conv层，用来对featuremap进行下采样。和典型的R50设计类似，每次下采样后，会增加dim的数量，一是为了提升感受野，二是为了增加特征性。下面详解每个部分的构成。

3.1. Convolutional Token Embeeding

顾名思义，用convolution来做embeeding，为了减少计算量，本文直接采用了7x7的卷积核，stride为4的卷积来直接对输入进行embeeding，假设输入为$H \times W \times 3$，那么输出为$\frac{H}{4} \times \frac{W}{4} \times C$。

3.2. Cross-Shaped Window Self-Attention

尽管有很强的长距离上下文建模能力，但原始的global self-attention的计算复杂度与特征图大小平方(H==W的情况)成正比的。因此，对于以高分辨率特征图为输入的视觉任务，如物体检测和分割，计算成本会非常大。为了缓解这个问题，现有的工作Swin等建议使用local windows self-attention，通过shift窗口来扩大感受野。然而，每个Transformer块内的token依旧是有限的注意区域，需要堆叠更多的block来实现全局感受野。为了更有效地扩大注意力区域和实现全局性的自我注意，有了Cross-shaped Window Self-attention，下面细讲是怎么做的以及代码实现。



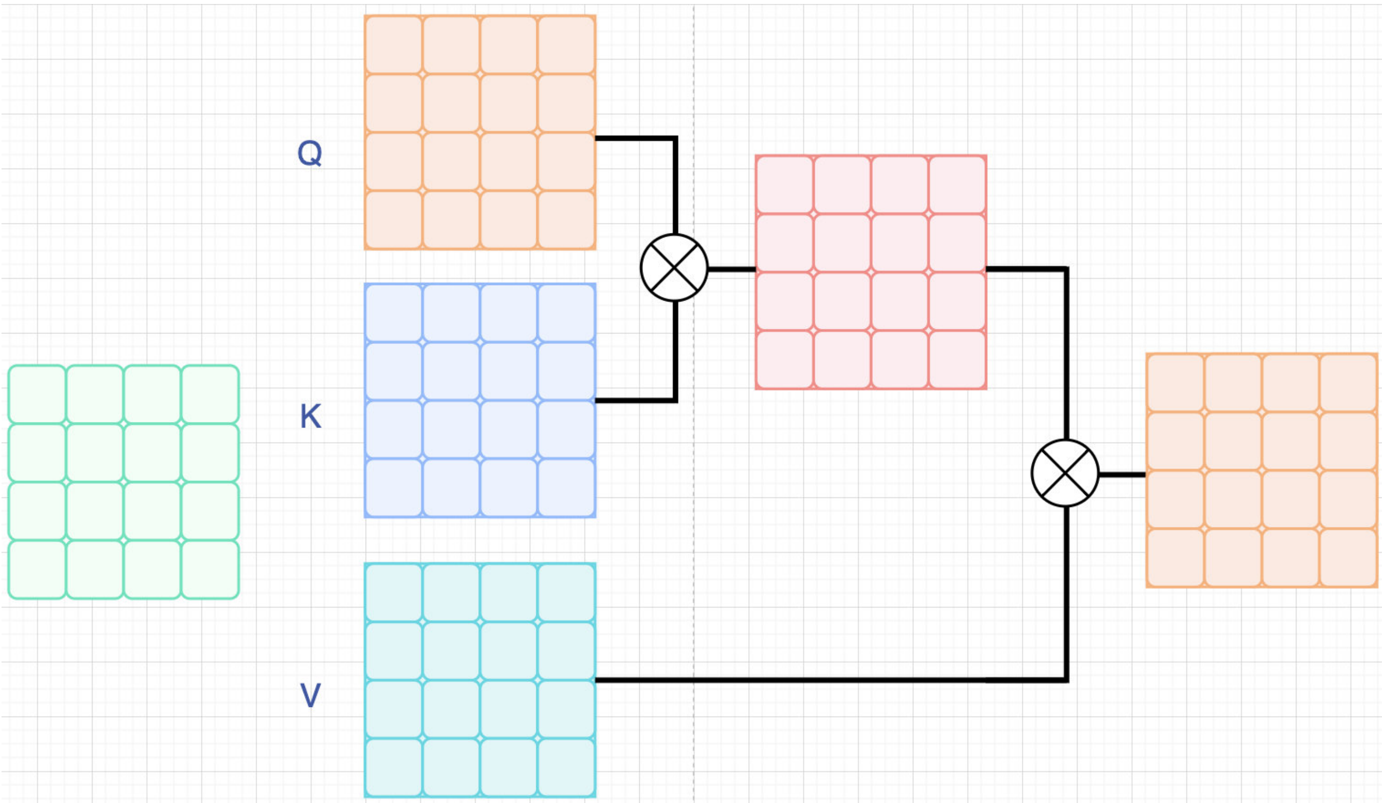
(a) Cross-Shaped Window Self-Attention

看图说话，很简单，假设原始的featuremap为$H \times W \times C$，设置windows的大小为$S_W \times S_H$，如果我们希望做行attention，设置S_W为W，设置S_H为s，那么就可以获得一个$s \times W$的局部窗口，同理，如果我们希望做列attention，设置S_H为H，设置S_W为s,可以获得一个$H \times s$的窗口。同时，对应的dim一分为2，一部分用于计算行attention，另一部分用于计算列attention，最后在concat起来，实现并行处理。由于transformers在计算attention的时候是采用mutilhead的，为了保持计算量，本文对head一分为2，一部分用于行attention，一部分用于列attention。以行attention为例，公式如下:

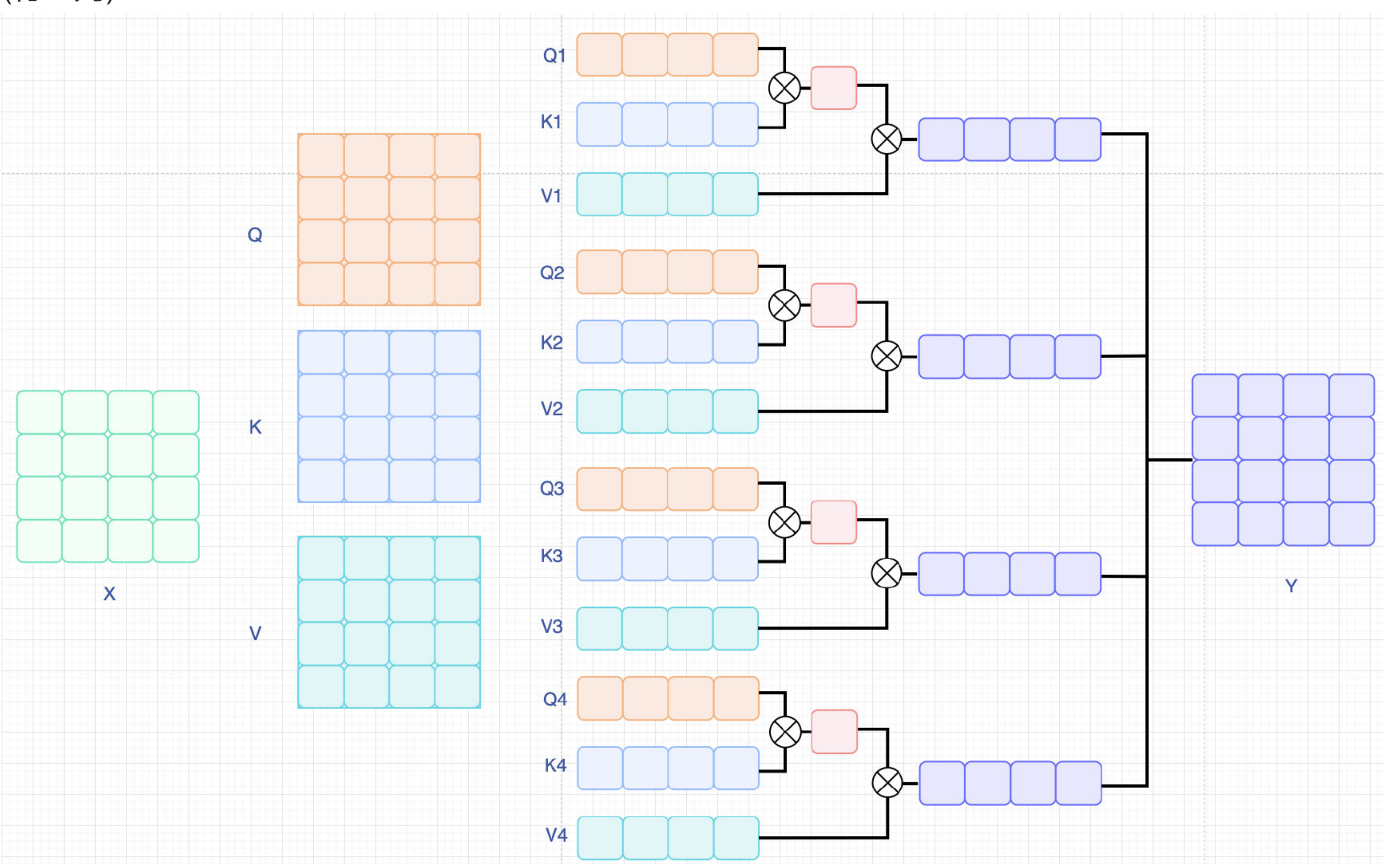
$$
\begin{gathered}
X \in R^{(H \times W) \times C} \\
X = [X^1, X^2, ..., X^M], where\ X^i \in R^{(sw \times W) \times C}\ and\ M = H/sw \\
Y_k^i = Attention(X^i W_k^Q, X^i W_k^K, X^i W_k^V), where\ i = 1, ..., M \\
Hattention_k(X) = [Y_k^1, Y_k^2, ..., Y_k^M] \\
CSWinattn(X) = Concat(head1, ..., headK)W^O \\
head_k = \begin{cases} Hattention_k(X) & \text{k=1,...,K/2} \\ Vattention_k(X) & \text{k=K/2+1,...,K} \end{cases}
\end{gathered}
$$

其中，窗口大小为(sw, H)，相比于标准的self-attention，区别在于H，或者W是部分的而不是全部的，如下图所示。

- 标准的self-attention

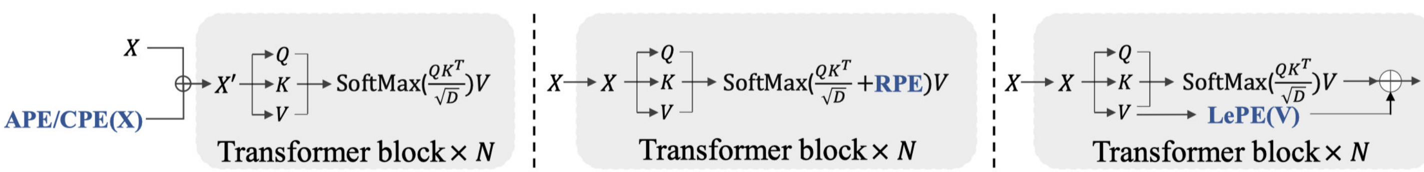


- (行or列)self-attention



- **自己的思考**
  其实乍一看很像ACNet和RepVGG，只不过他们是全都要，本文的话只要行和列的计算。在Transformers的attention中，Q实际上起指导的作用，K则是用来做token之间的交互，那么对于一个$X \in (N \times L)$的矩阵，会得到一个$(N \times N)$的attention map，意义就是再Q的指导下得到的关于K的attention。很多的时候我们会发现这个attention map 高亮的部分往往都是集中于对角线区域以及周围的部分区域，也就是自己attention自己和对自己有用的token。那么我们是不是就可以拆解这两部分，构造两个attention，一个用于自己attn自己，一个用于attn对自己有价值的位置。那么先拆解为$X \in (1 \times L)$表示的是第一个token，得到(1×1)的atten结果，那么意义就是当前的token于其他的token之间的相似度。反过来，$X \in (L \times 1)$表示的每个token，同样得到(1×1)的atten结果，但是意义为每个token指导第一个token的embeeding的变化。两者结合，就是找对自己有用的token。
- **Q&A**
  Question: 本文的另一个核心思想是增大感受野，那么怎么才能增大感受野呢？
  Answer: 首先明确一点，cross-shaped windows self-attention，并不是基于一个H和W相等的window来做attention的，实际的窗口大小是随着featuremap和滑动步长的改变而变化的。我们知道R50是通过1/32的下采样来获得很大的感受野，cross-shaped也是如此，通过降采样图像大小，同时增加窗口滑动步长, 最终从local-attention 变为 global-attention, 实现扩张感受野。(这里说感受野不准，应该表示为长距离依赖)

3.3. Locally-Enhanced Positional Encoding(LePE)



上图所示，左边为VIT模型的PE，使用的绝对位置编码或者是条件位置编码，只在embeeding的时候与token一起进入transformer，中间的是Swin，CrossFormer等模型的PE，使用相对位置编码偏差，不再和输入的embeeding一起进入transformer，通过引入token图的权重，来和attention一起计算，灵活度更好相对APE效果更好。最后就是本文所提出的LePE，相比于RPE，本文的方法更加直接，直接作用在value上，公式如下:

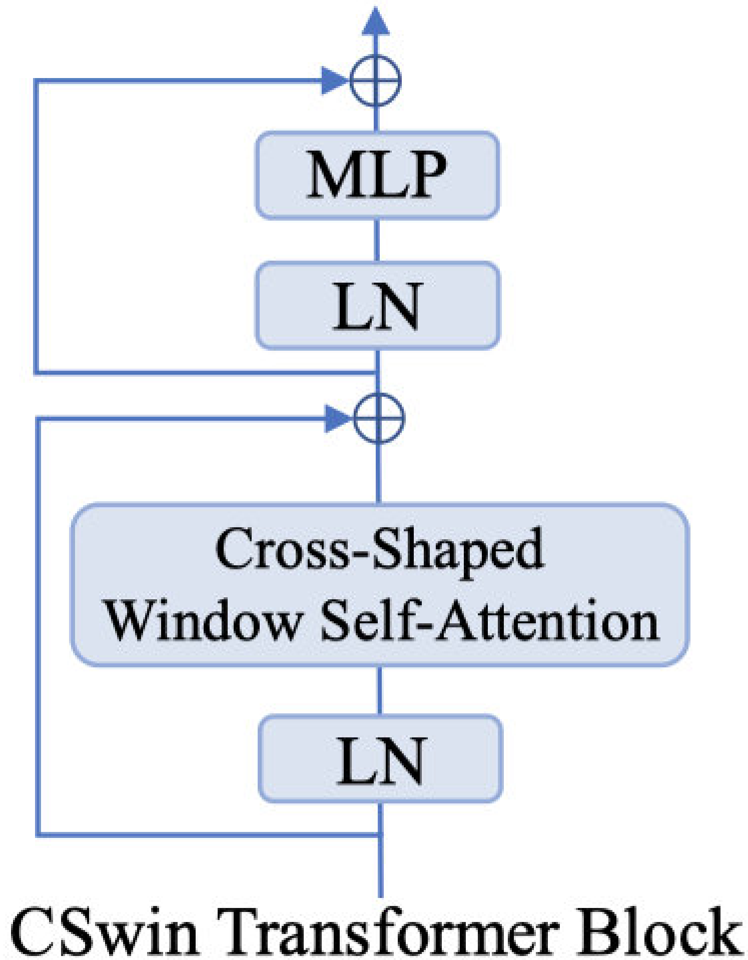
$$Attention(Q, K, V) = SoftMax(QK^T/\sqrt{d})V + EV$$

这里，E表示的是Value的位置权重，有$e_{ij}^V \in E$。
但是直接去计算E, 还是有一定程度的计算量，假设对于输入，对其影响大的元素只在他的附近，所以改写公式为:

$$Attention(Q, K, V) = SoftMax(QK^T/\sqrt{d})V + DWConv(V)$$

这样，LePE可以友好地应用于将任意输入分辨率作为输入的下游任务。

3.4. CSWin Transformer Block



CSwin的block很简单，有两个prenorm堆叠而成，一个是做LayerNorm和Cross-shaped window self-attention并接一个shortcut，另一个则是做LayerNorm和MLP，相比于Swin和Twins来说，block的计算量大大的降低了(swin,twins则是有两个attention+两个MLP堆叠一个block)。公式如下:

$$\hat{X}^l = CSWinAttention(LN(X^{l-1})) + X^{l-1}$$
$$X^l = MLP(LN(\hat{X}^l)) + \hat{X}^l$$

3.5. code review

```
class LePEAttention(nn.Module):
    def __init__(self, dim, resolution, idx, split_size=7, dim_out=None, num_heads=8, attn_drop=
        super().__init__()
        self.dim = dim
        self.dim_out = dim_out or dim
        self.resolution = resolution
        self.split_size = split_size
        self.num_heads = num_heads
        head_dim = dim // num_heads
        # NOTE scale factor was wrong in my original version, can set manually to be compat with
        self.scale = qk_scale or head_dim ** -0.5
        if idx == -1:    # global attenton
            H_sp, W_sp = self.resolution, self.resolution
        elif idx == 0:   # row attention
            H_sp, W_sp = self.resolution, self.split_size
        elif idx == 1:   # column attention
            W_sp, H_sp = self.resolution, self.split_size
        else:
            print ("ERROR MODE", idx)
            exit(0)
        self.H_sp = H_sp
        self.W_sp = W_sp
        stride = 1
        self.get_v = nn.Conv2d(dim, dim, kernel_size=3, stride=1, padding=1,groups=dim)

        self.attn_drop = nn.Dropout(attn_drop)

    def im2cswin(self, x):
        B, N, C = x.shape
        H = W = int(np.sqrt(N))
        # (B, N, C) -> (B, C, N) -> (B, C, H, W)
        x = x.transpose(-2,-1).contiguous().view(B, C, H, W)
        x = img2windows(x, self.H_sp, self.W_sp)  # (B*(H//h_sp, W//w_sp), h_sp * w_sp, C)
        # (B*(H//h_sp, W//w_sp), h_sp * w_sp, C) -> (B*(H//h_sp, W//w_sp), h_sp*w_sp, h, C//h) -
        x = x.reshape(-1, self.H_sp* self.W_sp, self.num_heads, C // self.num_heads).permute(0,
        return x

    def get_lepe(self, x, func):
        B, N, C = x.shape
        H = W = int(np.sqrt(N))
        x = x.transpose(-2,-1).contiguous().view(B, C, H, W)

        H_sp, W_sp = self.H_sp, self.W_sp
        x = x.view(B, C, H // H_sp, H_sp, W // W_sp, W_sp)
        x = x.permute(0, 2, 4, 1, 3, 5).contiguous().reshape(-1, C, H_sp, W_sp) ### B', C, H', W

        lepe = func(x) ### B', C, H', W'   # dw conv

        # (B', C, H', W') -> (B, h, C//h, h_sp * w_sp) -> (B, h, h_sp*w_sp, C//h)
        lepe = lepe.reshape(-1, self.num_heads, C // self.num_heads, H_sp * W_sp).permute(0, 1,
```

```python
        x = x.reshape(-1, self.num_heads, C // self.num_heads, self.H_sp* self.W_sp).permute(0,
        return x, lepe

    def forward(self, qkv):
        """
        x: B L C
        """
        q,k,v = qkv[0], qkv[1], qkv[2]

        ### Img2Window
        H = W = self.resolution
        B, L, C = q.shape
        assert L == H * W, "flatten img_tokens has wrong size"

        q = self.im2cswin(q)
        k = self.im2cswin(k)
        v, lepe = self.get_lepe(v, self.get_v)

        q = q * self.scale
        attn = (q @ k.transpose(-2, -1))  # B head N C @ B head C N --> B head N N
        attn = nn.functional.softmax(attn, dim=-1, dtype=attn.dtype)
        attn = self.attn_drop(attn)

        x = (attn @ v) + lepe  # B head N N @ B head N C
        # (B, h, N, C//h) --> (B, N, C)
        x = x.transpose(1, 2).reshape(-1, self.H_sp* self.W_sp, C)

        ### Window2Img
        x = windows2img(x, self.H_sp, self.W_sp, H, W).view(B, -1, C)  # B (H' W') C

        return x
```

代码很简单，对于滑窗后的处理，都是把外循环并入到了batch的维度了，可以并行处理。因为是按照dim来进行分水平和竖直的， 所以对应的heads也进行相应的分发处理。

4. 实验

4.1. 模型设计

Models	#Channels	#Blocks in 4 stages	*sw* in 4 stages	#heads in 4 stages	#Param.	FLOPs
CSWin-T	64	[1, 2, 21, 1]	[1, 2, 7, 7]	[2, 4, 8, 16]	23M	4.3G
CSWin-S	64	[2, 4, 32, 2]	[1, 2, 7, 7]	[2, 4, 8, 16]	35M	6.9G
CSWin-B	96	[2, 4, 32, 2]	[1, 2, 7, 7]	[2, 4, 8, 16]	78M	15.0G
CSWin-L	144	[2, 4, 32, 2]	[1, 2, 7, 7]	[6,12,24,48]	173M	31.5G

还是按照FLOPs的分布，来设计了四种模型,CSWin-T,CSWin-S,CSWin-B,CSwin-L，这里的FLOPs都是在224x224条件下计算的。

4.2. imagenet结果

ImageNet-1K 224^2 trained models			
Method	#Param.	FLOPs	Top-1
Reg-4G [42]	21M	4.0G	80.0
Eff-B4* [52]	19M	4.2G	**82.9**
DeiT-S [54]	22M	4.6G	79.8
PVT-S [59]	25M	3.8G	79.8
T2T-14 [67]	22M	5.2G	81.5
ViL-S [70]	25M	4.9G	82.0
TNT-S [21]	24M	5.2G	81.3
CViT-15 [4]	27M	5.6G	81.0
Visf-S [8]	40M	4.9G	82.3
LViT-S [37]	22M	4.6G	80.8
CoaTL-S [65]	20M	4.0G	81.9
CPVT-S [13]	23M	4.6G	81.5
Swin-T [39]	29M	4.5G	81.3
CvT-13 [61]	20M	4.5G	81.6
CSWin-T	23M	4.3G	82.7
ImageNet-1K 384^2 finetuned models			
CvT-13 [61]	20M	16.3G	83.0
T2T-14 [67]	22M	17.1G	83.3
CViT$_c$-15 [4]	28M	21.4G	83.5
CSWin-T	23M	14.0G	**84.3**

(a) Tiny Model

ImageNet-1K 224^2 trained models			
Method	#Param.	FLOPs	Top-1
Reg-8G [42]	39M	8.0G	81.7
Eff-B5* [52]	30M	9.9G	**83.6**
PVT-M [59]	44M	6.7G	81.2
PVT-L [59]	61M	9.8G	81.7
T2T-19 [67]	39M	8.9G	81.9
T2T$_t$-19 [67]	39M	9.8G	82.2
ViL-M [70]	40M	8.7G	83.3
MViT-B [20]	37M	7.8G	83.0
CViT-18 [4]	43M	9.0G	82.5
CViT$_c$-18 [4]	44M	9.5G	82.8
Twins-B [12]	56M	8.3G	83.2
Swin-S [39]	50M	8.7G	83.0
CvT-21 [61]	32M	7.1G	82.5
CSWin-S	35M	6.9G	**83.6**
ImageNet-1K 384^2 finetuned models			
CvT-21 [61]	32M	24.9G	83.3
CViT$_c$-18 [4]	45M	32.4G	83.9
CSWin-S	35M	22.0G	**85.0**

(b) Small Model

ImageNet-1K 224^2 trained models			
Method	#Param.	FLOPs	Top-1
Reg-16G [42]	84M	16.0G	82.9
Eff-B6* [52]	43M	19.0G	84.0
DeiT-B [54]	87M	17.5G	81.8
PiT-B [25]	74M	12.5G	82.0
T2T-24 [67]	64M	14.1G	82.3
T2T$_t$-24 [67]	64M	15.0G	82.6
CPVT-B [13]	88M	17.6G	82.3
TNT-B [21]	66M	14.1G	82.8
ViL-B [70]	56M	13.4G	83.2
Twins-L [12]	99M	14.8G	83.7
Swin-B [39]	88M	15.4G	83.3
CSWin-B	78M	15.0G	**84.2**
ImageNet-1K 384^2 finetuned models			
ViT-B/16 [18]	86M	49.3G	77.9
DeiT-B [54]	86M	55.4G	83.1
Swin-B [39]	88M	47.0G	84.2
CSWin-B	78M	47.0G	**85.4**

(c) Base Model

224表示的是模型再224x224的输入下，使用imagenet1k的数据来训练得到的结果，384表示的是在384x384上进行微调后的结果，可以看到CSWin取得了比较SOTA的结果。

Method	#Param.	Input Size	FLOPs	Top-1	Method	#Param.	Input Size	FLOPs	Top-1
R-101x3 [35]	388M	384^2	204.6G	84.4	R-152x4 [35]	937M	480^2	840.5G	85.4
ViT-B/16 [18]	86M	384^2	55.4G	84.0	ViT-L/16 [35]	307M	384^2	190.7G	85.2
ViL-B [70]	56M	384^2	43.7G	86.2	—	—	—	—	—
Swin-B [39]	88M	224^2 384^2	15.4G 47.1G	85.2 86.4	Swin-L [39]	197M	224^2 384^2	34.5G 103.9G	86.3 87.3
CSWin-B(ours)	78M	224^2 384^2	15.0G 47.0G	85.9 **87.0**	CSWin-L(ours)	173M	224^2 384^2	31.5G 96.8G	86.5 **87.5**

Table 3: ImageNet-1K fine-tuning results by pre-training on ImageNet-21K datasets.

使用imagenet21k做pretrain后在imagenet1k上微调的结果，可以发现用更多的数据训练出来的模型做pretrain对于所有模型都有提升，cswin无论是224和384尺度训练都取得了SOTA。

4.3. 检测和分割结果

Backbone	#Params (M)	FLOPs (G)	Cascade Mask R-CNN 3x +MS					
			AP^b	AP^b_{50}	AP^b_{75}	AP^m	AP^m_{50}	AP^m_{75}
Res50 [23]	82	739	46.3	64.3	50.5	40.1	61.7	43.4
Swin-T [39]	86	745	50.5	69.3	54.9	43.7	66.6	47.1
CSWin-T	80	757	**52.5**	**71.5**	**57.1**	**45.3**	**68.8**	**48.9**
X101-32 [64]	101	819	48.1	66.5	52.4	41.6	63.9	45.2
Swin-S [39]	107	838	51.8	70.4	56.3	44.7	67.9	48.5
CSWin-S	92	820	**53.7**	**72.2**	**58.4**	**46.4**	**69.6**	**50.6**
X101-64 [64]	140	972	48.3	66.4	52.3	41.7	64.0	45.1
Swin-B [39]	145	982	51.9	70.9	56.5	45.0	68.4	48.7
CSWin-B	135	1004	**53.9**	**72.6**	**58.5**	**46.4**	**70.0**	**50.4**

Table 5: Object detection and instance segmentation performance on the COCO val2017 with Cascade Mask R-CNN.

Backbone	#Params (M)	FLOPs (G)	Mask R-CNN 1x schedule						Mask R-CNN 3x + MS schedule					
			AP^b	AP^b_{50}	AP^b_{75}	AP^m	AP^m_{50}	AP^m_{75}	AP^b	AP^b_{50}	AP^b_{75}	AP^m	AP^m_{50}	AP^m_{75}
Res50 [23]	44	260	38.0	58.6	41.4	34.4	55.1	36.7	41.0	61.7	44.9	37.1	58.4	40.1
PVT-S [59]	44	245	40.4	62.9	43.8	37.8	60.1	40.3	43.0	65.3	46.9	39.9	62.5	42.8
ViL-S [70]	45	218	44.9	67.1	49.3	41.0	64.2	44.1	47.1	68.7	51.5	42.7	65.9	46.2
TwinsP-S [12]	44	245	42.9	65.8	47.1	40.0	62.7	42.9	46.8	69.3	51.8	42.6	66.3	46.0
Twins-S [12]	44	228	43.4	66.0	47.3	40.3	63.2	43.4	46.8	69.2	51.2	42.6	66.3	45.8
Swin-T [39]	48	264	42.2	64.6	46.2	39.1	61.6	42.0	46.0	68.2	50.2	41.6	65.1	44.8
CSWin-T	42	279	**46.7**	**68.6**	**51.3**	**42.2**	**65.6**	**45.4**	**49.0**	**70.7**	**53.7**	**43.6**	**67.9**	**46.6**
Res101 [23]	63	336	40.4	61.1	44.2	36.4	57.7	38.8	42.8	63.2	47.1	38.5	60.1	41.3
X101-32 [64]	63	340	41.9	62.5	45.9	37.5	59.4	40.2	44.0	64.4	48.0	39.2	61.4	41.9
PVT-M [59]	64	302	42.0	64.4	45.6	39.0	61.6	42.1	44.2	66.0	48.2	40.5	63.1	43.5
ViL-M [70]	60	261	43.4	—	—	39.7	—	—	44.6	66.3	48.5	40.7	63.8	43.7
TwinsP-B [12]	64	302	44.6	66.7	48.9	40.9	63.8	44.2	47.9	70.1	52.5	43.2	67.2	46.3
Twins-B [12]	76	340	45.2	67.6	49.3	41.5	64.5	44.8	48.0	69.5	52.7	43.0	66.8	46.6
Swin-S [39]	69	354	44.8	66.6	48.9	40.9	63.4	44.2	48.5	70.2	53.5	43.3	67.3	46.6
CSWin-S	54	342	**47.9**	**70.1**	**52.6**	**43.2**	**67.1**	**46.2**	**50.0**	**71.3**	**54.7**	**44.5**	**68.4**	**47.7**
X101-64 [64]	101	493	42.8	63.8	47.3	38.4	60.6	41.3	44.4	64.9	48.8	39.7	61.9	42.6
PVT-L[59]	81	364	42.9	65.0	46.6	39.5	61.9	42.5	44.5	66.0	48.3	40.7	63.4	43.7
ViL-B [70]	76	365	45.1	—	—	41.0	—	—	45.7	67.2	49.9	41.3	64.4	44.5
TwinsP-L [12]	81	364	45.4	—	—	41.5	—	—	—	—	—	—	—	—
Twins-L [12]	111	474	45.9	—	—	41.6	—	—	—	—	—	—	—	—
Swin-B [39]	107	496	46.9	—	—	42.3	—	—	48.5	69.8	53.2	43.4	66.8	46.9
CSWin-B	97	526	**48.7**	**70.4**	**53.9**	**43.9**	**67.8**	**47.3**	**50.8**	**72.1**	**55.8**	**44.9**	**69.1**	**48.3**

Table 4: Object detection and instance segmentation performance on the COCO val2017 with the Mask R-CNN framework. The FLOPs (G) are measured at resolution 800×1280, and the models are pre-trained on the ImageNet-1K dataset. ResNet/ResNeXt results results are copied from [59].

Backbone	Semantic FPN 80k			Upernet 160k		
	#Param.(M)	FLOPs(G)	mIoU(%)	#Param.(M)	FLOPs(G)	mIoU/MS mIoU(%)
Res50 [23]	28.5	183	36.7	—	—	—/—
PVT-S [59]	28.2	161	39.8	—	—	—/—
TwinsP-S [12]	28.4	162	44.3	54.6	919	46.2/47.5
Twins-S [12]	28.3	144	43.2	54.4	901	46.2/47.1
Swin-T [39]	31.9	182	41.5	59.9	945	44.5/45.8
CSWin-T (ours)	26.1	202	**48.2**	59.9	959	**49.3/50.4**
Res101 [23]	47.5	260	38.8	86.0	1029	—/44.9
PVT-M [59]	48.0	219	41.6	—	—	—/—
TwinsP-B [12]	48.1	220	44.9	74.3	977	47.1/48.4
Twins-B [12]	60.4	261	45.3	88.5	1020	47.7/48.9
Swin-S [39]	53.2	274	45.2	81.3	1038	47.6/49.5
CSWin-S (ours)	38.5	271	**49.2**	64.6	1027	**50.0/50.8**
X101-64 [64]	86.4	—	40.2	—	—	—/—
PVT-L [59]	65.1	283	42.1	—	—	—/—
TwinsP-L [12]	65.3	283	46.4	91.5	1041	48.6/49.8
Twins-L [12]	103.7	404	46.7	133.0	1164	48.8/50.2
Swin-B [39]	91.2	422	46.0	121.0	1188	48.1/49.7
CSWin-B (ours)	81.2	464	**49.9**	109.2	1222	**50.8/51.7**
Swin-B† [39]	—	—	—	121.0	1841	50.0/51.7
Swin-L† [39]	—	—	—	234.0	3230	52.1/53.5
CSWin-B† (ours)	—	—	—	109.2	1941	51.8/52.6
CSWin-L† (ours)	—	—	—	207.7	2745	**54.0/55.7**

Table 6: Performance comparison of different backbones on the ADE20K segmentation task. Two different frameworks semantic FPN and Upernet are used. FLOPs are calculated with resolution 512×2048. ResNet/ResNeXt results and Swin FPN results are copied from [59] and [12] respectively. † means the model is pretrained on ImageNet-21K and finetuned with 640×640 resolution.

下游任务上，均有着非常sota的表现。

4.4. 消融实验

- 模型结构+trick

	ImageNet			COCO				ADE20K		
	#Param.	FLOPs	Top1(%)	#Param.	FLOPs	AP^b	AP^m	#Param.	FLOPs	mIoU(%)
CSWin-T	23M	4.3G	**82.7**	42M	279G	**46.7**	**42.2**	26M	202G	**48.2**
Increasing sw → Fixed $sw = 1$	23M	4.1G	81.9	42M	258G	45.2	40.8	26M	179G	47.5
Parallel SA → Sequential SA	23M	4.3G	82.4	42M	279G	45.1	41.1	26M	202G	46.2
Deep-Narrow → Shallow-Wide Arch	30M	4.8G	82.2	50M	286G	45.8	41.8	34M	209G	46.6
Overlapped→Non-Overlapped CTE	21M	4.2G	82.6	41M	276G	45.4	41.3	25M	199G	47.0

Table 7: Ablation study of each component to better understand CSWin Transformer. “SA”, “Arch”, “CTE” denote “Self-Attention”, “Architecture”, and “Convolutional Token Embedding” respectively.

实验采用的模型是CSWin-T，imagenet上的结果为82.7%。

1. 滑动窗口的步长从每个stage增长改为每个stage固定为1，发现性能下降了0.8个点，说明感受野的大小会影响模型的结果
2. 并行attention改成序列化attention，性能降低了0.3%个点。
3. 模型的设计，从深窄变成矮胖结构，性能下降了0.5%个点，这一点实际上在CNN都已经有过证明了。
4. 卷积获取embeeding改为非重叠切片获取embeeding，性能下降了0.1%个点，说明overlap和非overlap对于token来说意义不大，因为最终也是可以看到全局的。

- attention&position embeeding

	ImageNet Top1(%)	COCO AP^b	COCO AP^m	ADE20K mIoU(%)
Sliding window [44]	81.4	—	—	—
Shifted window [39]	81.3	42.2	39.1	41.5
Spatially Sep [12]	81.5	42.7	39.5	42.9
Sequential Axial [26]	81.5	40.4	37.6	39.8
Cross-shaped window(ours)	**82.2**	**43.4**	**40.2**	**43.4**

(a) Comparison of different self-attention mechanisms.

	ImageNet Top1(%)	COCO AP^b	COCO AP^m	ADE20K mIoU(%)
No PE	82.5	44.8	41.1	47.0
APE [18]	82.6	45.1	41.1	45.7
CPE [13]	82.2	45.8	41.6	46.1
CPE* [13]	82.4	45.4	41.3	46.6
RPE [47]	**82.7**	45.5	41.3	46.6
LePE	**82.7**	**46.7**	**42.2**	**48.2**

(b) Comparison of different positional encoding mechanisms.

- 本文提出的Cross-shaped window self-attention机制，不仅在分类任务上超过之前的attention，同时检测和分割这样的dense任务上效果也非常不错，说明对于感受野的考虑是非常正确的。
- 虽然RPE和LePE在分类的任务上性能类似，但是对于形状变化多的dense任务上，LePE更深一筹。

5. 结论

在本文中，提出了CSWin Transformer。CSWin Transformer的核心设计是CSWin Self-Attention，它通过将多头分成平行组来执行水平和垂直条纹的自我注意。这种多头分组设计可以有效地扩大一个Transformer块内每个token的注意区域。同时，进一步将局部增强的位置编码引入CSWin Transformer，可以更有效的用于下游任务。大量的实验证明了CSWin Transformer的有效性和高效性。