

# Ai 缝合怪

## 1、SE Net 模块

论文《Squeeze-and-Excitation Networks》

### 1、作用

SENet 通过引入一个新的结构单元——“Squeeze-and-Excitation”（SE）块——来增强卷积神经网络的代表能力。是提高卷积神经网络（CNN）的表征能力，通过显式地建模卷积特征通道之间的依赖关系，从而在几乎不增加计算成本的情况下显著提升网络性能。SE 模块由两个主要操作组成：压缩（Squeeze）和激励（Excitation）

### 2、机制

#### 1、压缩操作：

SE 模块首先通过全局平均池化操作对输入特征图的空间维度（高度  $H$  和宽度  $W$ ）进行聚合，为每个通道生成一个通道描述符。这一步有效地将全局空间信息压缩成一个通道向量，捕获了通道特征响应的全局分布。这一全局信息对于接下来的重新校准过程至关重要。

#### 2、激励操作：

在压缩步骤之后，应用一个激励机制，该机制本质上是由两个全连接（FC）层和一个非线性激活函数（通常是 sigmoid）组成的自门控机制。第一个 FC 层降低了通道描述符的维度，应用 ReLU 非线性激活，随后第二个 FC 层将其投影回原始通道维度。这个过程建模了通道间的非线性交互，并产生了一组通道权重。

#### 3、特征重新校准：

激励操作的输出用于重新校准原始输入特征图。输入特征图的每个通道都由激励输出中对应的标量进行缩放。这一步骤有选择地强调信息丰富的特征，同时抑制不太有用的特征，使模型能够专注于任务中最相关的特征。

### 3、独特优势

#### 1、通道间依赖的显式建模：

SE Net 的核心贡献是通过 SE 块显式建模通道间的依赖关系，有效地提升了网络对不同通道特征重要性的适应性和敏感性。这种方法允许网络学会动态地调整各个通道的特征响应，以增强有用的特征并抑制不那么重要的特征。

## 2、轻量级且高效：

尽管 SE 块为网络引入了额外的计算，但其设计非常高效，额外的参数量和计算量相对较小。这意味着 SENet 可以在几乎不影响模型大小和推理速度的情况下，显著提升模型性能。

## 3、模块化和灵活性：

SE 块可以视为一个模块，轻松插入到现有 CNN 架构中的任何位置，包括 ResNet、Inception 和 VGG 等流行模型。这种模块化设计提供了极大的灵活性，使得 SENet 可以广泛应用于各种架构和任务中，无需对原始网络架构进行大幅度修改。

## 4、跨任务和跨数据集的泛化能力：

SENet 在多个基准数据集上展现出了优异的性能，包括图像分类、目标检测和语义分割等多个视觉任务。这表明 SE 块不仅能提升特定任务的性能，还具有良好的泛化能力，能够跨任务和跨数据集提升模型的效果。

## 5、增强的特征表征能力：

通过调整通道特征的重要性，SENet 能够更有效地利用模型的特征表征能力。这种增强的表征能力使得模型能够在更细粒度上理解图像内容，从而提高决策的准确性和鲁棒性。

## 4、代码：

```
import numpy as np

import torch

from torch import nn

from torch.nn import init

class SEAttention(nn.Module):

    # 初始化 SE 模块，channel 为通道数，reduction 为降维比率

    def __init__(self, channel=512, reduction=16):

        super().__init__()
```

```

self.avg_pool = nn.AdaptiveAvgPool2d(1) # 自适应平均池化层，将特征图的空间维度压缩为 1x1

self.fc = nn.Sequential( # 定义两个全连接层作为激励操作，通过降维和升维调整通道重要性

    nn.Linear(channel, channel // reduction, bias=False), # 降维，减少参数数量和计算量

    nn.ReLU(inplace=True), # ReLU 激活函数，引入非线性

    nn.Linear(channel // reduction, channel, bias=False), # 升维，恢复到原始通道数

    nn.Sigmoid() # Sigmoid 激活函数，输出每个通道的重要性系数

)

```

# 权重初始化方法

```

def init_weights(self):

    for m in self.modules(): # 遍历模块中的所有子模块

        if isinstance(m, nn.Conv2d): # 对于卷积层

            init.kaiming_normal_(m.weight, mode='fan_out') # 使用 Kaiming 初始化方法初始化权重

            if m.bias is not None:

                init.constant_(m.bias, 0) # 如果有偏置项，则初始化为 0

        elif isinstance(m, nn.BatchNorm2d): # 对于批归一化层

            init.constant_(m.weight, 1) # 权重初始化为 1

            init.constant_(m.bias, 0) # 偏置初始化为 0

        elif isinstance(m, nn.Linear): # 对于全连接层

            init.normal_(m.weight, std=0.001) # 权重使用正态分布初始化

            if m.bias is not None:

                init.constant_(m.bias, 0) # 偏置初始化为 0

```

```
# 前向传播方法
```

```
def forward(self, x):
```

```
    b, c, _, _ = x.size() # 获取输入 x 的批量大小 b 和通道数 c
```

```
    y = self.avg_pool(x).view(b, c) # 通过自适应平均池化层后，调整形状以匹配全连接层的输入
```

```
    y = self.fc(y).view(b, c, 1, 1) # 通过全连接层计算通道重要性，调整形状以匹配原始特征图的形状
```

```
    return x * y.expand_as(x) # 将通道重要性系数应用到原始特征图上，进行特征重新校准
```

```
# 示例使用
```

```
if __name__ == '__main__':
```

```
    input = torch.randn(50, 512, 7, 7) # 随机生成一个输入特征图
```

```
    se = SEAttention(channel=512, reduction=8) # 实例化 SE 模块，设置降维比率为 8
```

```
    output = se(input) # 将输入特征图通过 SE 模块进行处理
```

```
    print(output.shape) # 打印处理后的特征图形状，验证 SE 模块的作用
```

## 2、CBAM 模块

论文《CBAM: Convolutional Block Attention Module》

### 1、作用

是为了提升前馈卷积神经网络性能而提出的一种简单而有效的注意力模块。CBAM 通过顺序地推断两个维度上的注意力图（通道和空间），然后将这些注意力图乘以输入特征图进行自适应特征精炼。

## 2、机制

### 1、通道注意力模块（Channel Attention Module）：

通过利用特征之间的通道关系来生成通道注意力图。每个通道的特征图被视为一个特征探测器，通道注意力关注于给定输入图像中“什么”是有意义的。为了有效地计算通道注意力，CBAM 首先对输入特征图的空间维度进行压缩，同时使用平均池化和最大池化操作来捕获不同的空间上下文描述符，这些被送入共享的多层感知机（MLP）以产生通道注意力图。

### 2、空间注意力模块（Spatial Attention Module）：

利用特征之间的空间关系来生成空间注意力图。与通道注意力不同，空间注意力关注于“在哪里”是一个有信息的一部分，这与通道注意力是互补的。为了计算空间注意力，CBAM 首先沿着通道轴应用平均池化和最大池化操作，然后将它们连接起来生成一个高效的特征描述符。在该描述符上应用一个卷积层来生成空间注意力图。

## 3、独特优势

### 1、双重注意力机制：

CBAM 首次将通道注意力（Channel Attention）和空间注意力（Spatial Attention）顺序结合起来，对输入特征进行两阶段的精炼。这种设计让模型先关注于“哪些通道是重要的”，然后再关注于“空间上哪些位置是重要的”，从而更加全面地捕获特征中的关键信息。

### 2、自适应特征重标定：

通过通道注意力和空间注意力的逐步应用，CBAM 能够自适应地调整输入特征图中每个通道和空间位置的重要性。这种自适应重标定机制允许模型根据任务需求和内容上下文动态地关注于最有用的特征，从而提高模型的表征能力和决策准确性。

### 3、灵活性和通用性：

CBAM 模块设计简洁，可轻松集成到各种现有的 CNN 架构中，如 ResNet、Inception 等，而不需要对原始架构进行大的修改。这种灵活性和通用性使 CBAM 成为一种有效的插件，可以广泛应用于各种视觉识别任务，包括图像分类、目标检测和语义分割等。

### 4、计算效率高：

尽管 CBAM 为模型引入了额外的计算，但其设计考虑了计算效率，如通过全局平均池化和最大池化来简化通道注意力的计算，通过简单的卷积操作来实现空间注意力。这些设计使得 CBAM 能够在带来性能提升的同时，保持较低的额外计算成本。

### 5、逐步精炼策略：

CBAM 中通道和空间注意力的顺序应用，形成了一种逐步精炼输入特征的策略。这种从通道到空间的逐步细化过程，有助于模型更有效地利用注意力机制，逐渐提取并强调更加有意义的特征，而不是一次性地处理所有信息。

## 4、代码

```
import torch

from torch import nn

# 通道注意力模块

class ChannelAttention(nn.Module):

    def __init__(self, in_planes, ratio=16):

        super(ChannelAttention, self).__init__()

        self.avg_pool = nn.AdaptiveAvgPool2d(1) # 自适应平均池化

        self.max_pool = nn.AdaptiveMaxPool2d(1) # 自适应最大池化

        # 两个卷积层用于从池化后的特征中学习注意力权重

        self.fc1 = nn.Conv2d(in_planes, in_planes // ratio, 1, bias=False) # 第一个卷积层，降维

        self.relu1 = nn.ReLU() # ReLU 激活函数

        self.fc2 = nn.Conv2d(in_planes // ratio, in_planes, 1, bias=False) # 第二个卷积层，升维

        self.sigmoid = nn.Sigmoid() # Sigmoid 函数生成最终的注意力权重

    def forward(self, x):

        avg_out = self.fc2(self.relu1(self.fc1(self.avg_pool(x)))) # 对平均池化的特征进行处理

        max_out = self.fc2(self.relu1(self.fc1(self.max_pool(x)))) # 对最大池化的特征进行处理

        out = avg_out + max_out # 将两种池化的特征加权和作为输出
```

```
        return self.sigmoid(out) # 使用 sigmoid 激活函数计算注意力权重
```

# 空间注意力模块

```
class SpatialAttention(nn.Module):
```

```
    def __init__(self, kernel_size=7):
```

```
        super(SpatialAttention, self).__init__()
```

```
        assert kernel_size in (3, 7), 'kernel size must be 3 or 7' # 核心大小只能是 3 或 7
```

```
        padding = 3 if kernel_size == 7 else 1 # 根据核心大小设置填充
```

```
        # 卷积层用于从连接的平均池化和最大池化特征图中学习空间注意力权重
```

```
        self.conv1 = nn.Conv2d(2, 1, kernel_size, padding=padding, bias=False)
```

```
        self.sigmoid = nn.Sigmoid() # Sigmoid 函数生成最终的注意力权重
```

```
    def forward(self, x):
```

```
        avg_out = torch.mean(x, dim=1, keepdim=True) # 对输入特征图执行平均池化
```

```
        max_out, _ = torch.max(x, dim=1, keepdim=True) # 对输入特征图执行最大池化
```

```
        x = torch.cat([avg_out, max_out], dim=1) # 将两种池化的特征图连接起来
```

```
        x = self.conv1(x) # 通过卷积层处理连接后的特征图
```

```
        return self.sigmoid(x) # 使用 sigmoid 激活函数计算注意力权重
```

# CBAM 模块

```
class CBAM(nn.Module):
```

```
    def __init__(self, in_planes, ratio=16, kernel_size=7):
```

```
        super(CBAM, self).__init__()
```

```
        self.ca = ChannelAttention(in_planes, ratio) # 通道注意力实例
```

```
self.sa = SpatialAttention(kernel_size) # 空间注意力实例

def forward(self, x):

    out = x * self.ca(x) # 使用通道注意力加权输入特征图

    result = out * self.sa(out) # 使用空间注意力进一步加权特征图

    return result # 返回最终的特征图

# 示例使用

if __name__ == '__main__':

    block = CBAM(64) # 创建一个 CBAM 模块，输入通道为 64

    input = torch.rand(1, 64, 64, 64) # 随机生成一个输入特征图

    output = block(input) # 通过 CBAM 模块处理输入特征图

    print(input.size(), output.size()) # 打印输入和输出的
```

## 3、ECA 模块

论文《ECA-Net: Efficient Channel Attention for Deep Convolutional Neural Networks》

### 1、作用

ECA 模块旨在通过引入一种高效的通道注意力机制来增强深度卷积神经网络的特征表示能力。它着重于捕获通道间的动态依赖关系，从而使网络能够更加精确地重视对当前任务更重要的特征，提升模型在各种视觉任务上的性能。



## 2、机制

ECA 模块的核心机制是通过一个简单而高效的一维卷积来自适应地捕捉通道之间的依赖性，而无需降维和升维的过程。这种设计避免了传统注意力机制中复杂的多层感知机（MLP）结构，减少了模型复杂度和计算负担。ECA 通过计算一个自适应的核大小，直接在通道特征上应用一维卷积，从而学习到每个通道相对于其他通道的重要性。

## 3、独特优势

### 1、计算高效：

ECA 模块通过避免使用复杂的 MLP 结构，大幅降低了额外的计算成本和模型参数。这种高效的设计使得 ECA 能够在不增加显著计算负担的情况下，为模型带来性能提升。

### 2、无需降维升维：

与传统的注意力机制相比，ECA 模块无需进行降维和升维的操作，这样不仅保留了原始通道特征的信息完整性，还进一步减少了模型复杂度。

### 3、自适应核大小：

ECA 模块根据通道数自适应地调整一维卷积的核大小，使其能够灵活地捕捉不同范围内的通道依赖性，这种自适应机制使得 ECA 在不同规模的网络 and 不同深度的层次中都能有效工作。

### 4、易于集成：

由于其轻量级和高效的特性，ECA 模块可以轻松地嵌入到任何现有的 CNN 架构中，无需对原始网络架构进行大的修改，为提升网络性能提供了一种简单而有效的方式。

## 4、代码

```
import torch

from torch import nn

from torch.nn import init

# 定义 ECA 注意力模块的类
```

```

class ECAAttention(nn.Module):

    def __init__(self, kernel_size=3):

        super().__init__()

        self.gap = nn.AdaptiveAvgPool2d(1) # 定义全局平均池化层，将空间维度压缩为 1x1

        # 定义一个 1D 卷积，用于处理通道间的关系，核大小可调，padding 保证输出通道数不变

        self.conv = nn.Conv1d(1, 1, kernel_size=kernel_size, padding=(kernel_size - 1) // 2)

        self.sigmoid = nn.Sigmoid() # Sigmoid 函数，用于激活最终的注意力权重

# 权重初始化方法

def init_weights(self):

    for m in self.modules():

        if isinstance(m, nn.Conv2d):

            init.kaiming_normal_(m.weight, mode='fan_out') # 对 Conv2d 层使用 Kaiming 初始化

            if m.bias is not None:

                init.constant_(m.bias, 0) # 如果有偏置项，则初始化为 0

        elif isinstance(m, nn.BatchNorm2d):

            init.constant_(m.weight, 1) # 批归一化层权重初始化为 1

            init.constant_(m.bias, 0) # 批归一化层偏置初始化为 0

        elif isinstance(m, nn.Linear):

            init.normal_(m.weight, std=0.001) # 全连接层权重使用正态分布初始化

            if m.bias is not None:

```

```

init.constant_(m.bias, 0) # 全连接层偏置初始化为 0

# 前向传播方法

def forward(self, x):

    y = self.gap(x) # 对输入 x 应用全局平均池化，得到 bs,c,1,1 维度的输出

    y = y.squeeze(-1).permute(0, 2, 1) # 移除最后一个维度并转置，为 1D 卷积准备，变为 bs,1,c

    y = self.conv(y) # 对转置后的 y 应用 1D 卷积，得到 bs,1,c 维度的输出

    y = self.sigmoid(y) # 应用 Sigmoid 函数激活，得到最终的注意力权重

    y = y.permute(0, 2, 1).unsqueeze(-1) # 再次转置并增加一个维度，以匹配原始输入 x 的维度

    return x * y.expand_as(x) # 将注意力权重应用到原始输入 x 上，通过广播机制扩展维度并执行逐元素乘法

# 示例使用

if __name__ == '__main__':

    block = ECAAttention(kernel_size=3) # 实例化 ECA 注意力模块，指定核大小为 3

    input = torch.rand(1, 64, 64, 64) # 生成一个随机输入

    output = block(input) # 将输入通过 ECA 模块处理

    print(input.size(), output.size()) # 打印输入和输出的尺寸，验证 ECA 模块的作用

```

## 4、CoordAttention 模块

论文《Coordinate Attention for Efficient Mobile Network Design》

## 1、作用

**Coordinate Attention** 提出了一种新的注意力机制，用于在移动网络中嵌入位置信息到通道注意力中。这种方法不仅关注“哪些通道是重要的”，而且关注“在哪里”关注，通过更精细地控制空间选择性注意力图的生成，进一步提升模型性能。

## 2、机制

### 1、坐标信息嵌入：

与传统的通道注意力通过 2D 全局池化将特征张量转换为单一特征向量不同，**Coordinate Attention** 将通道注意力分解为两个 1D 特征编码过程，分别沿两个空间方向聚合特征。这种方法能够捕捉沿一个空间方向的长程依赖性，同时保留沿另一个空间方向的精确位置信息。

### 2、坐标注意力生成：

将沿垂直和水平方向聚合的特征图编码成一对方向感知和位置敏感的注意力图，这两个注意力图被互补地应用到输入特征图上，增强了对兴趣对象的表示。

## 3、独特优势

### 1、方向感知和位置敏感：

**Coordinate Attention** 通过生成方向感知和位置敏感的注意力图，使模型能够更准确地定位和识别兴趣对象。这种注意力图能够精确地高亮兴趣区域，提升了模型对空间结构的理解能力。

### 2、灵活性和轻量级：

**Coordinate Attention** 的设计简洁而高效，可以轻松嵌入到经典的移动网络结构中，如 MobileNetV2、MobileNeXt 和 EfficientNet，几乎不增加计算开销，适用于计算资源受限的环境。

### 3、跨任务性能提升：

**Coordinate Attention** 不仅在 ImageNet 分类任务上有效，更在下游任务如对象检测和语义分割上展现出更好的性能。这证明了其对于捕捉关键信息的能力，尤其在需要密集预测的任务中表现出色。

## 4、代码

```
import torch

import torch.nn as nn

import torch.nn.functional as F

# 定义 h_sigmoid 激活函数，这是一种硬 Sigmoid 函数

class h_sigmoid(nn.Module):

    def __init__(self, inplace=True):

        super(h_sigmoid, self).__init__()

        self.relu = nn.ReLU6(inplace=inplace) # 使用 ReLU6 实现

    def forward(self, x):

        return self.relu(x + 3) / 6 # 公式为 ReLU6(x+3)/6，模拟 Sigmoid 激活函数

# 定义 h_swish 激活函数，这是基于 h_sigmoid 的 Swish 函数变体

class h_swish(nn.Module):

    def __init__(self, inplace=True):

        super(h_swish, self).__init__()

        self.sigmoid = h_sigmoid(inplace=inplace) # 使用上面定义的 h_sigmoid

    def forward(self, x):

        return x * self.sigmoid(x) # 公式为 x * h_sigmoid(x)
```

# 定义 Coordinate Attention 模块

```
class CoordAtt(nn.Module):
```

```
    def __init__(self, inp, oup, reduction=32):
```

```
        super(CoordAtt, self).__init__()
```

# 定义水平和垂直方向的自适应平均池化

```
        self.pool_h = nn.AdaptiveAvgPool2d((None, 1)) # 水平方向
```

```
        self.pool_w = nn.AdaptiveAvgPool2d((1, None)) # 垂直方向
```

```
        mip = max(8, inp // reduction) # 计算中间层的通道数
```

# 1x1 卷积用于降维

```
        self.conv1 = nn.Conv2d(inp, mip, kernel_size=1, stride=1, padding=0)
```

```
        self.bn1 = nn.BatchNorm2d(mip) # 批归一化
```

```
        self.act = h_swish() # 激活函数
```

# 两个 1x1 卷积，分别对应水平和垂直方向

```
        self.conv_h = nn.Conv2d(mip, oup, kernel_size=1, stride=1, padding=0)
```

```
        self.conv_w = nn.Conv2d(mip, oup, kernel_size=1, stride=1, padding=0)
```

```
    def forward(self, x):
```

```
        identity = x # 保存输入作为残差连接
```

```

n, c, h, w = x.size() # 获取输入的尺寸

x_h = self.pool_h(x) # 水平方向池化

x_w = self.pool_w(x).permute(0, 1, 3, 2) # 垂直方向池化并交换维度以适应拼接

y = torch.cat([x_h, x_w], dim=2) # 拼接水平和垂直方向的特征

y = self.conv1(y) # 通过 1x1 卷积降维

y = self.bn1(y) # 批归一化

y = self.act(y) # 激活函数

```

```

x_h, x_w = torch.split(y, [h, w], dim=2) # 将特征拆分回水平和垂直方向

x_w = x_w.permute(0, 1, 3, 2) # 恢复 x_w 的原始维度

```

```

a_h = self.conv_h(x_h).sigmoid() # 通过 1x1 卷积并应用 Sigmoid 获取水平方向的注意力权重

a_w = self.conv_w(x_w).sigmoid() # 通过 1x1 卷积并应用 Sigmoid 获取垂直方向的注意力权重

```

```

out = identity * a_w * a_h # 应用注意力权重到输入特征，并与残差连接相乘

```

```

return out # 返回输出

```

# 示例使用

```

if __name__ == '__main__':

```

```

    block = CoordAtt(64, 64) # 实例化 Coordinate Attention 模块

```

```
input = torch.rand(1, 64, 64, 64) # 创建一个随机输入
```

```
output = block(input) # 通过模块处理输入
```

```
print(output.shape()) # 打印输入和输出的尺寸
```

## 5、SimAM 模块

论文《SimAM: A Simple, Parameter-Free Attention Module for Convolutional Neural Networks》

### 1、作用

SimAM (Simple Attention Module) 提出了一个概念简单但非常有效的注意力模块，用于卷积神经网络。与现有的通道维度和空间维度注意力模块不同，SimAM 能够为特征图中的每个神经元推断出 3D 注意力权重，而无需在原始网络中添加参数。

### 2、机制

#### 1、能量函数优化：

SimAM 基于著名的神经科学理论，通过优化一个能量函数来找出每个神经元的重要性。这个过程不添加任何新参数到原始网络中。

#### 2、快速闭合形式解决方案：

对于能量函数，SimAM 推导出了一个快速的闭合形式解决方案，并展示了这个解决方案可以在不到十行代码中实现。这种方法避免了结构调整的繁琐工作，使模块的设计更为简洁高效。

### 3、独特优势

#### 1、无参数设计：

SimAM 的一个显著优势是它不增加任何额外的参数。这使得 SimAM 可以轻松地集成到任何现有的 CNN 架构中，几乎不增加计算成本。



## 2、直接生成 3D 权重：

与大多数现有的注意力模块不同，SimAM 能够直接为每个神经元生成真正的 3D 权重，而不是仅仅在通道或空间维度上。这种全面的注意力机制能够更精确地捕捉到重要的特征信息。

## 3、基于神经科学的设计：

SimAM 的设计灵感来自于人类大脑中的注意力机制，尤其是空间抑制现象，使其在捕获视觉任务中的关键信息方面更为高效和自然。

## 4、代码

```
import torch

import torch.nn as nn

from thop import profile # 引入 thop 库来计算模型的 FLOPs 和参数数量

# 定义 SimAM 模块

class Simam_module(torch.nn.Module):

    def __init__(self, e_lambda=1e-4):

        super(Simam_module, self).__init__()

        self.act = nn.Sigmoid() # 使用 Sigmoid 激活函数

        self.e_lambda = e_lambda # 定义平滑项 e_lambda，防止分母为 0

    def forward(self, x):

        b, c, h, w = x.size() # 获取输入 x 的尺寸

        n = w * h - 1 # 计算特征图的元素数量减一，用于下面的归一化

        # 计算输入特征 x 与其均值之差的平方
```

```

x_minus_mu_square = (x - x.mean(dim=[2, 3], keepdim=True)).pow(2)

# 计算注意力权重 y, 这里实现了 SimAM 的核心计算公式

y = x_minus_mu_square / (4 * (x_minus_mu_square.sum(dim=[2, 3], keepdim=True) / n + self.e_lambda)) +
0.5

# 返回经过注意力加权的输入特征

return x * self.act(y)

# 示例使用

if __name__ == '__main__':

    model = Simam_module().cuda() # 实例化 SimAM 模块并移到 GPU 上

    x = torch.randn(1, 3, 64, 64).cuda() # 创建一个随机输入并移到 GPU 上

    y = model(x) # 将输入传递给模型

    print(y.size()) # 打印输出尺寸

    # 使用 thop 库计算模型的 FLOPs 和参数数量

    flops, params = profile(model, (x,))

    print(flops / 1e9) # 打印以 Giga FLOPs 为单位的浮点操作数

    print(params) # 打印模型参数数量

```

## 6、ACmix 模块

论文《On the Integration of Self-Attention and Convolution》

### 1、作用

ACmix 设计为一个结合了卷积和自注意力机制优势的混合模块，旨在通过融合两种机制的优点来增强模型的表示能力和性能。

## 2、机制

### 1、混合机制：

ACmix 通过结合自注意力机制的全局感知能力和卷积的局部特征提取能力，实现了一个高效的特征融合策略。这种策略通过在单个框架中同时利用这两种机制的优势，来提升模型对特征的处理能力。

### 2、卷积与自注意力的关联：

首先，ACmix 通过  $1\times 1$  卷积对输入特征图进行投影，产生一组丰富的中间特征。然后，这些中间特征被重用并根据不同的范式进行聚合，即自注意力和卷积方式。这种设计使 ACmix 既能享受到自注意力模块的灵活性，也能利用卷积的局部感受野特性。

### 3、改进的位移与求和操作：

ACmix 中卷积路径的中间特征遵循位移和求和操作，类似于传统卷积模块。为了提高实际推理效率，ACmix 采用了深度可分离卷积（depthwise convolution）来代替低效的张量位移操作。

## 3、独特优势

### 1、计算效率：

ACmix 通过优化计算路径和减少重复计算，提高了整体模块的计算效率，使其在不显著增加计算负担的前提下提升模型性能。

### 2、性能提升：

通过有效结合卷积和自注意力的优点，ACmix 在多个视觉任务上显示出优于单一机制（仅卷积或仅自注意力）的性能，展示了其广泛的应用潜力。

## 4、代码

```
import torch

import torch.nn as nn
```

```
import torch.nn.functional as F
```

```
# 定义一个函数生成位置编码，返回位置信息张量
```

```
def position(H, W, is_cuda=True):
```

```
    # 生成宽度和高度的位置信息，范围在-1 到 1 之间
```

```
    if is_cuda:
```

```
        loc_w = torch.linspace(-1.0, 1.0, W).cuda().unsqueeze(0).repeat(H, 1)
```

```
        loc_h = torch.linspace(-1.0, 1.0, H).cuda().unsqueeze(1).repeat(1, W)
```

```
    else:
```

```
        loc_w = torch.linspace(-1.0, 1.0, W).unsqueeze(0).repeat(H, 1)
```

```
        loc_h = torch.linspace(-1.0, 1.0, H).unsqueeze(1).repeat(1, W)
```

```
    loc = torch.cat([loc_w.unsqueeze(0), loc_h.unsqueeze(0)], 0).unsqueeze(0)
```

```
    return loc
```

```
# 定义一个函数实现步长操作，用于降采样
```

```
def stride(x, stride):
```

```
    b, c, h, w = x.shape
```

```
    return x[:, :, ::stride, ::stride]
```

```
# 初始化函数，将张量的值填充为 0.5
```

```
def init_rate_half(tensor):
```

```
    if tensor is not None:
```

```
        tensor.data.fill_(0.5)
```

# 初始化函数，将张量的值填充为 0

```
def init_rate_0(tensor):
```

```
    if tensor is not None:
```

```
        tensor.data.fill_(0.)
```

# 定义 ACmix 模块的类

```
class ACmix(nn.Module):
```

```
    def __init__(self, in_planes, out_planes, kernel_att=7, head=4, kernel_conv=3, stride=1, dilation=1):
```

```
        super(ACmix, self).__init__()
```

# 初始化模块参数

```
self.in_planes = in_planes
```

```
self.out_planes = out_planes
```

```
self.head = head
```

```
self.kernel_att = kernel_att
```

```
self.kernel_conv = kernel_conv
```

```
self.stride = stride
```

```
self.dilation = dilation
```

```
self.rate1 = torch.nn.Parameter(torch.Tensor(1)) # 注意力分支权重
```

```
self.rate2 = torch.nn.Parameter(torch.Tensor(1)) # 卷积分支权重
```

```
self.head_dim = self.out_planes // self.head # 每个头的维度
```

# 定义用于特征变换的卷积层

```
self.conv1 = nn.Conv2d(in_planes, out_planes, kernel_size=1)
```

```
self.conv2 = nn.Conv2d(in_planes, out_planes, kernel_size=1)
```

```
self.conv3 = nn.Conv2d(in_planes, out_planes, kernel_size=1)
```

```
self.conv_p = nn.Conv2d(2, self.head_dim, kernel_size=1) # 位置编码的卷积层
```

# 定义自注意力所需的 padding 和展开操作

```
self.padding_att = (self.dilation * (self.kernel_att - 1) + 1) // 2
```

```
self.pad_att = torch.nn.ReflectionPad2d(self.padding_att)
```

```
self.unfold = nn.Unfold(kernel_size=self.kernel_att, padding=0, stride=self.stride)
```

```
self.softmax = torch.nn.Softmax(dim=1)
```

# 定义用于生成动态卷积核的全连接层和深度可分离卷积层

```
self.fc = nn.Conv2d(3 * self.head, self.kernel_conv * self.kernel_conv, kernel_size=1, bias=False)
```

```
self.dep_conv = nn.Conv2d(self.kernel_conv * self.kernel_conv * self.head_dim, out_planes,  
  
                           kernel_size=self.kernel_conv, bias=True, groups=self.head_dim, padding=1,  
  
                           stride=stride)
```

```
self.reset_parameters() # 参数初始化
```

```
def reset_parameters(self):
```

```
    init_rate_half(self.rate1)
```

```

init_rate_half(self.rate2)

kernel = torch.zeros(self.kernel_conv * self.kernel_conv, self.kernel_conv, self.kernel_conv)

for i in range(self.kernel_conv * self.kernel_conv):

    kernel[i, i // self.kernel_conv, i % self.kernel_conv] = 1.

kernel = kernel.squeeze(0).repeat(self.out_planes, 1, 1, 1)

self.dep_conv.weight = nn.Parameter(data=kernel, requires_grad=True)

self.dep_conv.bias = init_rate_0(self.dep_conv.bias)


def forward(self, x):

    q, k, v = self.conv1(x), self.conv2(x), self.conv3(x)

    scaling = float(self.head_dim) ** -0.5

    b, c, h, w = q.shape

    h_out, w_out = h // self.stride, w // self.stride

    pe = self.conv_p(position(h, w, x.is_cuda))

    q_att = q.view(b * self.head, self.head_dim, h, w) * scaling

    k_att = k.view(b * self.head, self.head_dim, h, w)

    v_att = v.view(b * self.head, self.head_dim, h, w)

    if self.stride > 1:

        q_att = stride(q_att, self.stride)

```

```

        q_pe = stride(pe, self.stride)

    else:

        q_pe = pe

    unfold_k = self.unfold(self.pad_att(k_att)).view(b * self.head, self.head_dim,

                                                    self.kernel_att * self.kernel_att,

                                                    w_out) # b*head, head_dim, k_att^2, h_out, w_out

    unfold_rpe = self.unfold(self.pad_att(pe)).view(1, self.head_dim, self.kernel_att * self.kernel_att,

                                                    h_out,

                                                    w_out) # 1, head_dim, k_att^2, h_out, w_out

    att = (q_att.unsqueeze(2) * (unfold_k + q_pe.unsqueeze(2) - unfold_rpe)).sum(

        1) # (b*head, head_dim, 1, h_out, w_out) * (b*head, head_dim, k_att^2, h_out, w_out) -> (b*head,

        k_att^2, h_out, w_out)

    att = self.softmax(att)

    out_att = self.unfold(self.pad_att(v_att)).view(b * self.head, self.head_dim, self.kernel_att *

                                                    self.kernel_att,

                                                    h_out, w_out)

    out_att = (att.unsqueeze(1) * out_att).sum(2).view(b, self.out_planes, h_out, w_out)

    f_all = self.fc(torch.cat(

        [q.view(b, self.head, self.head_dim, h * w), k.view(b, self.head, self.head_dim, h * w),

        v.view(b, self.head, self.head_dim, h * w)], 1))

```



```
f_conv = f_all.permute(0, 2, 1, 3).reshape(x.shape[0], -1, x.shape[-2], x.shape[-1])
```

```
out_conv = self.dep_conv(f_conv)
```

```
return self.rate1 * out_att + self.rate2 * out_conv
```

```
# 输入 N C H W, 输出 N C H W
```

```
if __name__ == '__main__':
```

```
    block = ACmix(in_planes=64, out_planes=64)
```

```
    input = torch.rand(1, 64, 64, 64)
```

```
    output = block(input)
```

```
    print(input.size(), output.size())
```

## 7、Axial\_attention 模块

论文《AXIAL ATTENTION IN MULTIDIMENSIONAL TRANSFORMERS》

### 1、作用

Axial Attention 提出了一种用于图像和其他作为高维张量组织的数据的自注意力基的自回归模型。传统的自回归模型要么因高维数据而导致计算资源需求过大，要么为了减少资源需求而在分布表达性或实现简便性方面做出妥协。Axial Transformers 设计旨在在保持数据上联合分布的完整表达性和易于使用标准深度学习框架实现的同时，要求合理的内存和计算资源，并在标准生成建模基准上实现最先进的结果。

## 2、机制

### 1、轴向注意力：

与对张量元素的序列应用标准自注意力不同，Axial Transformer 沿着张量的单个轴应用注意力，称为“轴向注意力”，而不是展平张量。这种操作在计算和内存使用上比标准自注意力节省显著，因为它自然地与张量的多个维度对齐。

### 2、半并行结构：

Axial Transformer 的层结构允许在解码时并行计算绝大多数上下文，而无需引入任何独立性假设，这对于即使是非常大的 Axial Transformer 也是广泛适用的。

## 3、独特优势

### 1、计算效率：

Axial Transformer 通过轴向注意力操作在资源使用上实现了显著节省，对于具有  $N = N1/d \times \dots \times N1/d$  形状的  $d$  维张量，相比标准自注意力，轴向注意力在资源上节省了  $O(N(d-1)/d)$  因子。

### 2、完全表达性：

尽管 Axial Transformer 沿单个轴应用注意力，但其结构设计确保了模型可以表达数据的全局依赖性，不丢失对前一个像素的依赖性。

### 3、简单易实现：

Axial Transformer 不需要为 GPU 或 TPU 编写特定的子程序，它可以使用深度学习框架中广泛可用的高效操作（主要是密集的 MatMul 操作）简单实现。

## 4、代码

```
import torch

from torch import nn

from operator import itemgetter

from torch.autograd.function import Function

from torch.utils.checkpoint import get_device_states, set_device_states
```

```
class Deterministic(nn.Module):

    def __init__(self, net):

        super().__init__()

        self.net = net

        self.cpu_state = None

        self.cuda_in_fwd = None

        self.gpu_devices = None

        self.gpu_states = None

    def record_rng(self, *args):

        self.cpu_state = torch.get_rng_state()

        if torch.cuda._initialized:

            self.cuda_in_fwd = True

            self.gpu_devices, self.gpu_states = get_device_states(*args)

    def forward(self, *args, record_rng=False, set_rng=False, **kwargs):

        if record_rng:

            self.record_rng(*args)

        if not set_rng:
```

```
return self.net(*args, **kwargs)
```

```
rng_devices = []
```

```
if self.cuda_in_fwd:
```

```
    rng_devices = self.gpu_devices
```

```
with torch.random.fork_rng(devices=rng_devices, enabled=True):
```

```
    torch.set_rng_state(self.cpu_state)
```

```
    if self.cuda_in_fwd:
```

```
        set_device_states(self.gpu_devices, self.gpu_states)
```

```
    return self.net(*args, **kwargs)
```

```
class ReversibleBlock(nn.Module):
```

```
    def __init__(self, f, g):
```

```
        super().__init__()
```

```
        self.f = Deterministic(f)
```

```
        self.g = Deterministic(g)
```

```
    def forward(self, x, f_args={}, g_args={}):
```

```
        x1, x2 = torch.chunk(x, 2, dim=1)
```

```
y1, y2 = None, None
```

```
with torch.no_grad():
```

```
    y1 = x1 + self.f(x2, record_rng=self.training, **f_args)
```

```
    y2 = x2 + self.g(y1, record_rng=self.training, **g_args)
```

```
return torch.cat([y1, y2], dim=1)
```

```
def backward_pass(self, y, dy, f_args={}, g_args={}):
```

```
    y1, y2 = torch.chunk(y, 2, dim=1)
```

```
    del y
```

```
    dy1, dy2 = torch.chunk(dy, 2, dim=1)
```

```
    del dy
```

```
with torch.enable_grad():
```

```
    y1.requires_grad = True
```

```
    gy1 = self.g(y1, set_rng=True, **g_args)
```

```
    torch.autograd.backward(gy1, dy2)
```

```
with torch.no_grad():
```

```
    x2 = y2 - gy1
```

```
del y2, gy1

dx1 = dy1 + y1.grad

del dy1

y1.grad = None

with torch.enable_grad():

    x2.requires_grad = True

    fx2 = self.f(x2, set_rng=True, **f_args)

    torch.autograd.backward(fx2, dx1, retain_graph=True)

with torch.no_grad():

    x1 = y1 - fx2

    del y1, fx2

    dx2 = dy2 + x2.grad

    del dy2

    x2.grad = None

    x = torch.cat([x1, x2.detach()], dim=1)

    dx = torch.cat([dx1, dx2], dim=1)
```

```
return x, dx
```

```
class IrreversibleBlock(nn.Module):
```

```
    def __init__(self, f, g):
```

```
        super().__init__()
```

```
        self.f = f
```

```
        self.g = g
```

```
    def forward(self, x, f_args, g_args):
```

```
        x1, x2 = torch.chunk(x, 2, dim=1)
```

```
        y1 = x1 + self.f(x2, **f_args)
```

```
        y2 = x2 + self.g(y1, **g_args)
```

```
        return torch.cat([y1, y2], dim=1)
```

```
class _ReversibleFunction(Function):
```

```
    @staticmethod
```

```
    def forward(ctx, x, blocks, kwargs):
```

```
        ctx.kwargs = kwargs
```

```
        for block in blocks:
```

```
            x = block(x, **kwargs)
```

```
ctx.y = x.detach()
```

```
ctx.blocks = blocks
```

```
return x
```

```
@staticmethod
```

```
def backward(ctx, dy):
```

```
    y = ctx.y
```

```
    kwargs = ctx.kwargs
```

```
    for block in ctx.blocks[::-1]:
```

```
        y, dy = block.backward_pass(y, dy, **kwargs)
```

```
    return dy, None, None
```

```
class ReversibleSequence(nn.Module):
```

```
    def __init__(self, blocks, ):
```

```
        super().__init__()
```

```
        self.blocks = nn.ModuleList([ReversibleBlock(f, g) for (f, g) in blocks])
```

```
    def forward(self, x, arg_route=(True, True), **kwargs):
```

```
        f_args, g_args = map(lambda route: kwargs if route else {}, arg_route)
```

```
        block_kwargs = {'f_args': f_args, 'g_args': g_args}
```

```
        x = torch.cat((x, x), dim=1)
```



```
x = _ReversibleFunction.apply(x, self.blocks, block_kwargs)
```

```
return torch.stack(x.chunk(2, dim=1)).mean(dim=0)
```

```
def exists(val):
```

```
    return val is not None
```

```
def map_el_ind(arr, ind):
```

```
    return list(map(itemgetter(ind), arr))
```

```
def sort_and_return_indices(arr):
```

```
    indices = [ind for ind in range(len(arr))]
```

```
    arr = zip(arr, indices)
```

```
    arr = sorted(arr)
```

```
    return map_el_ind(arr, 0), map_el_ind(arr, 1)
```

```

def calculate_permutations(num_dimensions, emb_dim):

    total_dimensions = num_dimensions + 2

    emb_dim = emb_dim if emb_dim > 0 else (emb_dim + total_dimensions)

    axial_dims = [ind for ind in range(1, total_dimensions) if ind != emb_dim]

    permutations = []

    for axial_dim in axial_dims:

        last_two_dims = [axial_dim, emb_dim]

        dims_rest = set(range(0, total_dimensions)) - set(last_two_dims)

        permutation = [*dims_rest, *last_two_dims]

        permutations.append(permutation)

    return permutations

```

```

class ChanLayerNorm(nn.Module):

    def __init__(self, dim, eps=1e-5):

        super().__init__()

```

```
self.eps = eps
```

```
self.g = nn.Parameter(torch.ones(1, dim, 1, 1))
```

```
self.b = nn.Parameter(torch.zeros(1, dim, 1, 1))
```

```
def forward(self, x):
```

```
    std = torch.var(x, dim=1, unbiased=False, keepdim=True).sqrt()
```

```
    mean = torch.mean(x, dim=1, keepdim=True)
```

```
    return (x - mean) / (std + self.eps) * self.g + self.b
```

```
class PreNorm(nn.Module):
```

```
    def __init__(self, dim, fn):
```

```
        super().__init__()
```

```
        self.fn = fn
```

```
        self.norm = nn.LayerNorm(dim)
```

```
    def forward(self, x):
```

```
        x = self.norm(x)
```

```
        return self.fn(x)
```

```
class Sequential(nn.Module):
```

```
def __init__(self, blocks):
```

```
    super().__init__()
```

```
    self.blocks = blocks
```

```
def forward(self, x):
```

```
    for f, g in self.blocks:
```

```
        x = x + f(x)
```

```
        x = x + g(x)
```

```
    return x
```

```
class PermuteToFrom(nn.Module):
```

```
def __init__(self, permutation, fn):
```

```
    super().__init__()
```

```
    self.fn = fn
```

```
    _, inv_permutation = sort_and_return_indices(permutation)
```

```
    self.permutation = permutation
```

```
    self.inv_permutation = inv_permutation
```

```
def forward(self, x, **kwargs):
```

```
    axial = x.permute(*self.permutation).contiguous()
```

```
shape = axial.shape
```

```
*_, t, d = shape
```

```
axial = axial.reshape(-1, t, d)
```

```
axial = self.fn(axial, **kwargs)
```

```
axial = axial.reshape(*shape)
```

```
axial = axial.permute(*self.inv_permutation).contiguous()
```

```
return axial
```

```
class AxialPositionalEmbedding(nn.Module):
```

```
    def __init__(self, dim, shape, emb_dim_index=1):
```

```
        super().__init__()
```

```
        parameters = []
```

```
        total_dimensions = len(shape) + 2
```

```
ax_dim_indexes = [i for i in range(1, total_dimensions) if i != emb_dim_index]
```

```
self.num_axials = len(shape)
```

```
for i, (axial_dim, axial_dim_index) in enumerate(zip(shape, ax_dim_indexes)):
```

```
    shape = [1] * total_dimensions
```

```
    shape[emb_dim_index] = dim
```

```
    shape[axial_dim_index] = axial_dim
```

```
    parameter = nn.Parameter(torch.randn(*shape))
```

```
    setattr(self, f'param_{i}', parameter)
```

```
def forward(self, x):
```

```
    for i in range(self.num_axials):
```

```
        x = x + getattr(self, f'param_{i}')
```

```
    return x
```

```
class SelfAttention(nn.Module):
```

```
    def __init__(self, dim, heads, dim_heads=None):
```

```
        super().__init__()
```

```
        self.dim_heads = (dim // heads) if dim_heads is None else dim_heads
```

```

dim_hidden = self.dim_heads * heads

self.heads = heads

self.to_q = nn.Linear(dim, dim_hidden, bias=False)

self.to_kv = nn.Linear(dim, 2 * dim_hidden, bias=False)

self.to_out = nn.Linear(dim_hidden, dim)

def forward(self, x, kv=None):

    kv = x if kv is None else kv

    q, k, v = (self.to_q(x), *self.to_kv(kv).chunk(2, dim=-1))

    b, t, d, h, e = *q.shape, self.heads, self.dim_heads

    merge_heads = lambda x: x.reshape(b, -1, h, e).transpose(1, 2).reshape(b * h, -1, e)

    q, k, v = map(merge_heads, (q, k, v))

    dots = torch.einsum('bie,bje->bij', q, k) * (e ** -0.5)

    dots = dots.softmax(dim=-1)

    out = torch.einsum('bij,bje->bie', dots, v)

    out = out.reshape(b, h, -1, e).transpose(1, 2).reshape(b, -1, d)

    out = self.to_out(out)

```

```
return out
```

```
class AxialAttention(nn.Module):
```

```
    def __init__(self, dim, num_dimensions=2, heads=8, dim_heads=None, dim_index=-1, sum_axial_out=True):
```

```
        assert (dim % heads) == 0, 'hidden dimension must be divisible by number of heads'
```

```
        super().__init__()
```

```
        self.dim = dim
```

```
        self.total_dimensions = num_dimensions + 2
```

```
        self.dim_index = dim_index if dim_index > 0 else (dim_index + self.total_dimensions)
```

```
        attentions = []
```

```
        for permutation in calculate_permutations(num_dimensions, dim_index):
```

```
            attentions.append(PermuteToFrom(permutation, SelfAttention(dim, heads, dim_heads)))
```

```
        self.axial_attentions = nn.ModuleList(attentions)
```

```
        self.sum_axial_out = sum_axial_out
```

```
    def forward(self, x):
```

```
        assert len(x.shape) == self.total_dimensions, 'input tensor does not have the correct number of dimensions'
```

```
        assert x.shape[self.dim_index] == self.dim, 'input tensor does not have the correct input dimension'
```



```

if self.sum_axial_out:

    return sum(map(lambda axial_attn: axial_attn(x), self.axial_attentions))

out = x

for axial_attn in self.axial_attentions:

    out = axial_attn(out)

return out

```

```

class AxialImageTransformer(nn.Module):

    def __init__(self, dim, depth, heads=8, dim_heads=None, dim_index=1, reversible=True,
axial_pos_emb_shape=None):

        super().__init__()

        permutations = calculate_permutations(2, dim_index)

        get_ff = lambda: nn.Sequential(

            ChanLayerNorm(dim),

            nn.Conv2d(dim, dim * 4, 3, padding=1),

            nn.LeakyReLU(inplace=True),

            nn.Conv2d(dim * 4, dim, 3, padding=1)

```

```
)
```

```
self.pos_emb = AxialPositionalEmbedding(dim, axial_pos_emb_shape, dim_index) if exists(  
    axial_pos_emb_shape) else nn.Identity()
```

```
layers = nn.ModuleList([])
```

```
for _ in range(depth):
```

```
    attn_functions = nn.ModuleList(  
        [PermuteToFrom(permutation, PreNorm(dim, SelfAttention(dim, heads, dim_heads))) for  
permutation in
```

```
        permutations])
```

```
    conv_functions = nn.ModuleList([get_ff(), get_ff()])
```

```
    layers.append(attn_functions)
```

```
    layers.append(conv_functions)
```

```
execute_type = ReversibleSequence if reversible else Sequential
```

```
self.layers = execute_type(layers)
```

```
def forward(self, x):
```

```
    x = self.pos_emb(x)
```

```
    return self.layers(x)
```

```
if __name__ == '__main__':  
  
    block = AxialImageTransformer(  
  
        dim=64,  
  
        depth=12,  
  
        reversible=True  
  
    ).cuda()  
  
    input = torch.rand(1, 64, 64, 64).cuda()  
  
    output = block(input)  
  
    print(output.shape)
```

## 8、CoTAttention 模块

论文《Contextual Transformer Networks for Visual Recognition》

### 1、 作用

Contextual Transformer (CoT) block 设计为视觉识别的一种新颖的 Transformer 风格模块。该设计充分利用输入键之间的上下文信息指导动态注意力矩阵的学习，从而加强视觉表示的能力。CoT block 首先通过 3x3 卷积对输入键进行上下文编码，得到输入的静态上下文表示。然后，将编码后的键与输入查询合并，通过两个连续的 1x1 卷积学习动态多头注意力矩阵。学习到的注意力矩阵乘以输入值，实现输入的动态上下文表示。最终将静态和动态上下文表示的融合作为输出。

### 2、 机制

1、上下文编码：

通过 3x3 卷积在所有邻居键内部空间上下文化每个键表示，捕获键之间的静态上下文信息。

## 2、动态注意力学习：

基于查询和上下文化的键的连接，通过两个连续的 1x1 卷积产生注意力矩阵，这一过程自然地利用每个查询和所有键之间的相互关系进行自我注意力学习，并由静态上下文指导。

## 3、静态和动态上下文的融合：

将静态上下文和通过上下文化自注意力得到的动态上下文结合，作为 CoT block 的最终输出。

# 3、 独特优势

## 1、上下文感知：

CoT 通过在自注意力学习中探索输入键之间的富上下文信息，使模型能够更准确地捕获视觉内容的细微差异。

## 2、动静态上下文的统一：

CoT 设计巧妙地将上下文挖掘与自注意力学习统一到单一架构中，既利用键之间的静态关系又探索动态特征交互，提升了模型的表达能力。

## 3、灵活替换与优化：

CoT block 可以直接替换现有 ResNet 架构中的标准卷积，不增加参数和 FLOP 预算的情况下实现转换为 Transformer 风格的骨干网络（CoTNet），通过广泛的实验验证了其在多种应用（如图像识别、目标检测和实例分割）中的优越性。

# 4、代码

```
# 导入必要的 PyTorch 模块

import torch

from torch import nn

from torch.nn import functional as F


class CoTAttention(nn.Module):
```

# 初始化 CoT 注意力模块

```
def __init__(self, dim=512, kernel_size=3):
```

```
    super().__init__()
```

```
    self.dim = dim # 输入的通道数
```

```
    self.kernel_size = kernel_size # 卷积核大小
```

# 定义用于键(key)的卷积层，包括一个分组卷积，BatchNorm 和 ReLU 激活

```
self.key_embed = nn.Sequential(
```

```
    nn.Conv2d(dim, dim, kernel_size=kernel_size, padding=kernel_size//2, groups=4, bias=False),
```

```
    nn.BatchNorm2d(dim),
```

```
    nn.ReLU()
```

```
)
```

# 定义用于值(value)的卷积层，包括一个 1x1 卷积和 BatchNorm

```
self.value_embed = nn.Sequential(
```

```
    nn.Conv2d(dim, dim, 1, bias=False),
```

```
    nn.BatchNorm2d(dim)
```

```
)
```

# 缩小因子，用于降低注意力嵌入的维度

```
factor = 4
```

# 定义注意力嵌入层，由两个卷积层、一个 BatchNorm 层和 ReLU 激活组成

```

self.attention_embed = nn.Sequential(

    nn.Conv2d(2*dim, 2*dim//factor, 1, bias=False),

    nn.BatchNorm2d(2*dim//factor),

    nn.ReLU(),

    nn.Conv2d(2*dim//factor, kernel_size*kernel_size*dim, 1)

)

```

```

def forward(self, x):

```

```

    # 前向传播函数

```

```

    bs, c, h, w = x.shape # 输入特征的尺寸

```

```

    k1 = self.key_embed(x) # 生成键的静态表示

```

```

    v = self.value_embed(x).view(bs, c, -1) # 生成值的表示并调整形状

```

```

    y = torch.cat([k1, x], dim=1) # 将键的静态表示和原始输入连接

```

```

    att = self.attention_embed(y) # 生成动态注意力权重

```

```

    att = att.reshape(bs, c, self.kernel_size*self.kernel_size, h, w)

```

```

    att = att.mean(2, keepdim=False).view(bs, c, -1) # 计算注意力权重的均值并调整形状

```

```

    k2 = F.softmax(att, dim=-1) * v # 应用注意力权重到值上

```

```

    k2 = k2.view(bs, c, h, w) # 调整形状以匹配输出

```

```

    return k1 + k2 # 返回键的静态和动态表示的总和

```

```
# 实例化 CoTAttention 模块并测试

if __name__ == '__main__':

    block = CoTAttention(64) # 创建一个输入通道数为 64 的 CoTAttention 实例

    input = torch.rand(1, 64, 64, 64) # 创建一个随机输入

    output = block(input) # 通过 CoTAttention 模块处理输入

    print(output.shape) # 打印输入和输出的尺寸
```

## 9、TripletAttention 模块

论文《Rotate to Attend: Convolutional Triplet Attention Module》

### 1、作用

Triplet Attention 是一种新颖的注意力机制，它通过捕获跨维度交互，利用三支结构来计算注意力权重。对于输入张量，Triplet Attention 通过旋转操作建立维度间的依赖关系，随后通过残差变换对信道和空间信息进行编码，实现了几乎不增加计算成本的情况下，有效增强视觉表征的能力。

### 2、机制

#### 1、三支结构：

Triplet Attention 包含三个分支，每个分支负责捕获输入的空间维度 H 或 W 与信道维度 C 之间的交互特征。

#### 2、跨维度交互：

通过在每个分支中对输入张量进行排列（permute）操作，并通过 Z-pool 和  $k \times k$  的卷积层处理，以捕获跨维度的交互特征。

#### 3、注意力权重的生成：

利用 sigmoid 激活层生成注意力权重，并应用于排列后的输入张量，然后将其排列回原始输入形状。

### 3、 独特优势

#### 1、跨维度交互：

Triplet Attention 通过捕获输入张量的跨维度交互，提供了丰富的判别特征表征，较之前的注意力机制（如 SENet、CBAM 等）能够更有效地增强网络的性能。

#### 2、几乎无计算成本增加：

相比于传统的注意力机制，Triplet Attention 在提升网络性能的同时，几乎不增加额外的计算成本和参数数量，使得它可以轻松地集成到经典的骨干网络中。

#### 3、无需降维：

与其他注意力机制不同，Triplet Attention 不进行维度降低处理，这避免了因降维可能导致的信息丢失，保证了信道与权重间的直接对应关系。

总的来说，Triplet Attention 通过其独特的三支结构 and 跨维度交互机制，在提高模型性能的同时，保持了计算效率，显示了其在各种视觉任务中的应用潜力。

### 4、 代码

```
import torch

import torch.nn as nn

# 定义一个基本的卷积模块，包括卷积、批归一化和 ReLU 激活

class BasicConv(nn.Module):

    def __init__(self, in_planes, out_planes, kernel_size, stride=1, padding=0, dilation=1, groups=1, relu=True,
bn=True, bias=False):

        super(BasicConv, self).__init__()

        self.out_channels = out_planes

        # 定义卷积层
```



```
self.conv = nn.Conv2d(in_planes, out_planes, kernel_size=kernel_size, stride=stride, padding=padding,
dilation=dilation, groups=groups, bias=bias)
```

```
# 条件性地添加批归一化层
```

```
self.bn = nn.BatchNorm2d(out_planes, eps=1e-5, momentum=0.01, affine=True) if bn else None
```

```
# 条件性地添加 ReLU 激活函数
```

```
self.relu = nn.ReLU() if relu else None
```

```
def forward(self, x):
```

```
x = self.conv(x) # 应用卷积
```

```
if self.bn is not None:
```

```
x = self.bn(x) # 应用批归一化
```

```
if self.relu is not None:
```

```
x = self.relu(x) # 应用 ReLU
```

```
return x
```

```
# 定义 ZPool 模块，结合最大池化和平均池化结果
```

```
class ZPool(nn.Module):
```

```
def forward(self, x):
```

```
# 结合最大值和平均值
```

```
return torch.cat((torch.max(x, 1)[0].unsqueeze(1), torch.mean(x, 1).unsqueeze(1)), dim=1)
```

```
# 定义注意力门，用于根据输入特征生成注意力权重
```

```
class AttentionGate(nn.Module):
```

```
def __init__(self):
```

```
    super(AttentionGate, self).__init__()
```

```
    kernel_size = 7 # 设定卷积核大小
```

```
    self.compress = ZPool() # 使用 ZPool 模块
```

```
    self.conv = BasicConv(2, 1, kernel_size, stride=1, padding=(kernel_size - 1) // 2, relu=False) # 通过卷积调整通道数
```

```
def forward(self, x):
```

```
    x_compress = self.compress(x) # 应用 ZPool
```

```
    x_out = self.conv(x_compress) # 通过卷积生成注意力权重
```

```
    scale = torch.sigmoid_(x_out) # 应用 Sigmoid 激活
```

```
    return x * scale # 将注意力权重乘以原始特征
```

```
# 定义 TripletAttention 模块，结合了三种不同方向的注意力门
```

```
class TripletAttention(nn.Module):
```

```
    def __init__(self, no_spatial=False):
```

```
        super(TripletAttention, self).__init__()
```

```
        self.cw = AttentionGate() # 定义宽度方向的注意力门
```

```
        self.hc = AttentionGate() # 定义高度方向的注意力门
```

```
        self.no_spatial = no_spatial # 是否忽略空间注意力
```

```
        if not no_spatial:
```

```
            self.hw = AttentionGate() # 定义空间方向的注意力门
```

```
def forward(self, x):
```

```
    # 应用注意力门并结合结果
```

```
    x_perm1 = x.permute(0, 2, 1, 3).contiguous() # 转置以应用宽度方向的注意力
```

```
    x_out1 = self.cw(x_perm1)
```

```
    x_out11 = x_out1.permute(0, 2, 1, 3).contiguous() # 还原转置
```

```
    x_perm2 = x.permute(0, 3, 2, 1).contiguous() # 转置以应用高度方向的注意力
```

```
    x_out2 = self.hc(x_perm2)
```

```
    x_out21 = x_out2.permute(0, 3, 2, 1).contiguous() # 还原转置
```

```
    if not self.no_spatial:
```

```
        x_out = self.hw(x) # 应用空间注意力
```

```
        x_out = 1 / 3 * (x_out + x_out11 + x_out21) # 结合三个方向的结果
```

```
    else:
```

```
        x_out = 1 / 2 * (x_out11 + x_out21) # 结合两个方向的结果（如果 no_spatial 为 True）
```

```
    return x_out
```

```
# 示例代码
```

```
if __name__ == '__main__':
```

```
    input = torch.randn(50, 512, 7, 7) # 生成随机输入
```

```
    triplet = TripletAttention() # 实例化 TripletAttention
```

```
    output = triplet(input) # 应用 TripletAttention
```

```
    print(output.shape) # 打印输出形状
```

# 10、S2Attention 模块

论文《S2-MLPV2: IMPROVED SPATIAL-SHIFT MLP ARCHITECTURE FOR VISION》

## 1、作用

S2-MLPv2 是一个改进的空间位移多层感知器（MLP）视觉骨架网络，旨在通过利用通道维度的扩展和分割以及采用分割注意力（split-attention）操作来增强图像识别准确性。与传统的 S2-MLP 相比，S2-MLPv2 在不同的部分执行不同的空间位移操作，然后利用分割注意力操作来融合这些部分。此外，该方法采用了较小尺度的图像块和金字塔结构，进一步提升图像识别精度。

## 2、机制

### 1、特征图扩展和分割：

首先沿着通道维度扩展特征图，然后将扩展后的特征图分割成多个部分。

### 2、空间位移操作：

对每个分割的部分执行不同的空间位移操作，以增强特征表征。

### 3、分割注意力操作：

使用分割注意力操作融合经过空间位移处理的各个部分，生成融合后的特征图。

### 4、金字塔结构：

采用较小尺度的图像块和层次化的金字塔结构，以捕获更精细的视觉细节，提高模型的识别精度。

## 3、独特优势

### 1、增强的特征表征能力：

通过对特征图进行扩展、分割和不同方向的空间位移操作，S2-MLPv2 能够捕获更加丰富的特征信息，提升模型的特征能力。

### 2、分割注意力机制：

利用分割注意力操作有效地融合了不同空间位移处理的特征，进一步增强了特征的代表力。

### 3、金字塔结构的应用：

通过采用较小尺度的图像块和层次化的金字塔结构，S2-MLPv2 模型能够更好地捕捉图像中的细粒度细节，从而在图像识别任务上达到更高的准确率。

### 4、高效的性能：

即使在没有自注意力机制和额外训练数据的情况下，S2-MLPv2 也能在 ImageNet-1K 基准上达到 83.6%的顶级 1 准确率，表现优于其他 MLP 模型，同时参数数量更少，表明其在实际部署中具有竞争力。

## 4、代码

```
import numpy as np

import torch

from torch import nn

from torch.nn import init


def spatial_shift1(x):

    # 实现第一种空间位移，位移图像的四分之一块

    b, w, h, c = x.size()

    # 以下四行代码分别向左、向右、向上、向下移动图像的四分之一块

    x[:, 1:, :, :c // 4] = x[:, :w - 1, :, :c // 4]

    x[:, :w - 1, :, c // 4:c // 2] = x[:, 1:, :, c // 4:c // 2]

    x[:, :, 1:, c // 2:c * 3 // 4] = x[:, :, :h - 1, c // 2:c * 3 // 4]

    x[:, :, :h - 1, 3 * c // 4:] = x[:, :, 1:, 3 * c // 4:]
```

```
return x
```

```
def spatial_shift2(x):
```

```
# 实现第二种空间位移，逻辑与 spatial_shift1 相似，但位移方向不同
```

```
b, w, h, c = x.size()
```

```
# 对图像的四分之一块进行空间位移
```

```
x[:, :, 1:, :c // 4] = x[:, :, :h - 1, :c // 4]
```

```
x[:, :, :h - 1, c // 4:c // 2] = x[:, :, 1:, c // 4:c // 2]
```

```
x[:, 1:, :, c // 2:c * 3 // 4] = x[:, :w - 1, :, c // 2:c * 3 // 4]
```

```
x[:, :w - 1, :, 3 * c // 4:] = x[:, 1:, :, 3 * c // 4:]
```

```
return x
```

```
class SplitAttention(nn.Module):
```

```
# 定义分割注意力模块，使用 MLP 层进行特征转换和注意力权重计算
```

```
def __init__(self, channel=512, k=3):
```

```
    super().__init__()
```

```
    self.channel = channel
```

```
    self.k = k # 分割的块数
```

```
# 定义 MLP 层和激活函数
```

```
self.mlp1 = nn.Linear(channel, channel, bias=False)
```

```
self.gelu = nn.GELU()
```

```
self.mlp2 = nn.Linear(channel, channel * k, bias=False)
```

```
self.softmax = nn.Softmax(1)
```

```
def forward(self, x_all):
```

```
    # 计算分割注意力，并应用于输入特征
```

```
    b, k, h, w, c = x_all.shape
```

```
    x_all = x_all.reshape(b, k, -1, c) # 重塑维度
```

```
    a = torch.sum(torch.sum(x_all, 1), 1) # 聚合特征
```

```
    hat_a = self.mlp2(self.gelu(self.mlp1(a))) # 通过 MLP 计算注意力权重
```

```
    hat_a = hat_a.reshape(b, self.k, c) # 调整形状
```

```
    bar_a = self.softmax(hat_a) # 应用 softmax 获取注意力分布
```

```
    attention = bar_a.unsqueeze(-2) # 增加维度
```

```
    out = attention * x_all # 将注意力权重应用于特征
```

```
    out = torch.sum(out, 1).reshape(b, h, w, c) # 聚合并调整形状
```

```
    return out
```

```
class S2Attention(nn.Module):
```

```
# S2 注意力模块，整合空间位移和分割注意力
```

```
def __init__(self, channels=512):
```

```
    super().__init__()
```

```
    # 定义 MLP 层
```

```
    self.mlp1 = nn.Linear(channels, channels * 3)
```

```
    self.mlp2 = nn.Linear(channels, channels)
```

```
    self.split_attention = SplitAttention()
```

```
def forward(self, x):
```

```
    b, c, w, h = x.size()
```

```
    x = x.permute(0, 2, 3, 1) # 调整维度顺序
```

```
    x = self.mlp1(x) # 通过 MLP 层扩展特征
```

```
    x1 = spatial_shift1(x[:, :, :, :c]) # 应用第一种空间位移
```

```
    x2 = spatial_shift2(x[:, :, :, c:c * 2]) # 应用第二种空间位移
```

```
    x3 = x[:, :, :, c * 2:] # 保留原始特征的一部分
```

```
    x_all = torch.stack([x1, x2, x3], 1) # 堆叠特征
```

```
    a = self.split_attention(x_all) # 应用分割注意力
```



```
x = self.mlp2(a) # 通过另一个 MLP 层缩减特征维度
```

```
x = x.permute(0, 3, 1, 2) # 调整维度顺序回原始
```

```
return x
```

```
# 示例代码
```

```
if __name__ == '__main__':
```

```
    input = torch.randn(50, 512, 7, 7) # 创建输入张量
```

```
    s2att = S2Attention(channels=512) # 实例化 S2 注意力模块
```

```
    output = s2att(input) # 通过 S2 注意力模块处理输入
```

```
    print(output.shape) # 打印输出张量的形状
```

# 11、DilateFormer 模块

论文《DilateFormer: Multi-Scale Dilated Transformer for Visual Recognition》

## 1、作用

DilateFormer 旨在通过采用多尺度膨胀注意力机制，提升视觉识别任务中的特征提取能力。这种方法通过在不同的尺度上应用膨胀卷积，捕获图像的局部和全局依赖性。

## 2、机制

### 1、DilateAttention 模块：

该模块实现了膨胀注意力机制。它首先通过 `nn.Unfold` 提取滑动局部块，然后利用膨胀参数改变局部块的尺寸和间隔，实现对不同尺度特征的感知。通过对查询（q）、键（k）和值（v）的操作，计算注意力权重，并通过加权求和获取输出特征图。

### 2、MultiDilateLocalAttention 模块：

该模块将输入特征图在通道维度拆分为多个尺度（根据膨胀率的不同），然后对每个尺度分别应用 `DilateAttention`。这允许模型在不同的尺度上捕获信息，最后通过投影层合并各尺度的输出特征。

## 3、独特优势

### 1、多尺度感知能力：

通过结合不同膨胀率的膨胀注意力机制，`DilateFormer` 能够同时捕获图像的局部细节和全局上下文信息，增强模型的特征提取能力。

### 2、局部和全局信息的有效整合：

通过在不同的膨胀尺度上应用注意力机制，能够有效整合来自不同感受野大小的特征信息，提升了特征的表达力。

### 3、灵活性和适应性：

该模型能够根据任务需求调整膨胀率的设置，灵活适应不同的视觉识别任务，展现出良好的泛化能力。

## 4、代码

```
import numpy as np

import torch

from torch import flatten, nn

from torch.nn import init

from torch.nn.modules.activation import ReLU

from torch.nn.modules.batchnorm import BatchNorm2d

from torch.nn import functional as F
```

"DilateFormer: Multi-Scale Dilated Transformer for Visual Recognition"

```
class DilateAttention(nn.Module):
```

"Implementation of Dilate-attention"

```
def __init__(self, head_dim, qk_scale=None, attn_drop=0, kernel_size=3, dilation=1):
```

```
    super().__init__()
```

```
    self.head_dim = head_dim
```

```
    self.scale = qk_scale or head_dim ** -0.5
```

```
    self.kernel_size=kernel_size
```

# nn.unfold(): Extracts sliding local blocks from a batched input tensor. local block 是根据卷积核、膨胀率确定下来的，以此达到局部、稀疏的效果

```
    self.unfold = nn.Unfold(kernel_size=kernel_size, dilation=dilation,
padding=dilation*(kernel_size-1)//2, stride=1)
```

```
    self.attn_drop = nn.Dropout(attn_drop)
```

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

```
    #(B, scale_d, H, W)    scale_d == d;    C = scale_num * scale_d
```

```
    B,d,H,W = q.shape
```

# 首先对 q 进行变换: (B,d,H,W) -->reshape--> (B,h,hd,1,HW) -->permute--> (B,h,HW,1,hd) d=h\*hd;  
(hd:head\_dim), h:注意力头的个数, 这里的注意力头的个数和 MultiDilateLocalAttention 中的设置是不一样的, 在 MultiDilateLocalAttention 中, 也就是 76 行, 因为 N 个尺度平分了 C 个通道, 所以在这里注意力头的个数  $h=(C/N)/\text{head\_dim}$

```
q = q.reshape([B, d//self.head_dim, self.head_dim, 1, H*W]).permute(0, 1, 4, 3, 2) # B,h,N,1,d
```

# 通过滑动窗口选取 keys patch: (B,d,H,W) --> (B,d\*k\*k,HW) 作者通过 zero padding 填充, 使输入前后的 patch 数量都为 HW 个; k\*k 是滑窗的尺寸, 经过滑动窗口移动完之后, 得到 HW 个 patch, 每个 patch 的大小是 k\*k //unfold 的计算方式见官网: <https://pytorch.org/docs/stable/generated/torch.nn.Unfold.html#torch.nn.Unfold>, 解释见博客: [https://blog.csdn.net/qq\\_37937847/article/details/115663343](https://blog.csdn.net/qq_37937847/article/details/115663343)

```
k = self.unfold(k)
```

# 对 k 进行变换, 便于与 q 进行乘法: (B,d\*k\*k,HW) --> (B,h,hd,k\*k,HW) --> (B,h,HW,hd,k\*k) d=h\*hd;  
HW\*(k\*k): HW 个 patch, 每个 patch 的大小是 k\*k 个像素, 每个像素的通道是 hd

```
k = k.reshape([B, d//self.head_dim, self.head_dim, self.kernel_size*self.kernel_size, H*W]).permute(0, 1, 4, 2, 3)
```

# 每个 query patch 对选择的 k\*k 个 keys patch 做注意力: (B,h,HW,1,hd) @ (B,h,HW,hd,k\*k) = (B,h,HW,1,k\*k)

```
attn = (q @ k) * self.scale
```

```
attn = attn.softmax(dim=-1)
```

```
attn = self.attn_drop(attn)
```

# 同样, 通过滑动窗口选取 values patch: (B,d,H,W) --> (B,d\*k\*k,HW) --> (B,h,hd,k\*k,HW) --> (B,h,HW,k\*k,hd)

```
v = self.unfold(v).reshape([B, d//self.head_dim, self.head_dim, self.kernel_size*self.kernel_size, H*W]).permute(0, 1, 4, 3, 2)
```

# 通过 attn 权重对 values 加权求和: (B,h,HW,1,k\*k) @ (B,h,HW,k\*k,hd) = (B,h,HW,1,hd) --> transpose --> (B,HW,h,1,hd) --> reshape --> (B,H,W,h\*hd) == (B,H,W,d)

```
x = (attn @ v).transpose(1, 2).reshape(B, H, W, d)
```

```
return x
```

```
class MultiDilateLocalAttention(nn.Module):
```

"Implementation of Dilate-attention"

```
def __init__(self, dim, num_heads=8, qkv_bias=False, qk_scale=None,

              attn_drop=0., proj_drop=0., kernel_size=3, dilation=[1, 3]):

    super().__init__()

    self.dim = dim

    self.num_heads = num_heads

    head_dim = dim // num_heads

    self.dilation = dilation

    self.kernel_size = kernel_size

    self.scale = qk_scale or head_dim ** -0.5

    self.num_dilation = len(dilation)

    # 膨胀率的个数要能够被注意力头的个数整除才行

    assert num_heads % self.num_dilation == 0, f"num_heads{num_heads} must be the times of
num_dilation{self.num_dilation}!!"

    self.qkv = nn.Conv2d(dim, dim * 3, 1, bias=qkv_bias)

    self.dilate_attention = nn.ModuleList(

        [DilateAttention(head_dim, qk_scale, attn_drop, kernel_size, dilation[i])

         for i in range(self.num_dilation)])

    self.proj = nn.Linear(dim, dim)

    self.proj_drop = nn.Dropout(proj_drop)

def forward(self, x):

    B, H, W, C = x.shape

    x = x.permute(0, 3, 1, 2) # (B,H,W,C)-->(B,C,H,W)
```

```

# 通过卷积层,将通道 C 映射到 3C,并将其在通道上平分为 qkv: (B,C,H,W) --> (B,3C,H,W) -->
(B,3,scale_num,scale_d,H,W) --> (scale_num,3,B,scale_d,H,W) // num_dilation==scale_num; C=scale_num *
scale_d 有多少个膨胀率就说明有多少个尺度,所以这里定义: num_dilation=scale_num

qkv = self.qkv(x).reshape(B, 3, self.num_dilation, C//self.num_dilation, H, W).permute(2, 1, 0, 3, 4,
5)

# (B,C,H,W) --> (B,scale_num,scale_d,H,W) --> (scale_num,B,H,W,scale_d)

x = x.reshape(B, self.num_dilation, C//self.num_dilation, H, W).permute(1, 0, 3, 4, 2 )

# 循环计算每一个尺度,在这里 dilation=[1, 3],有两种取值,意味着有两个尺度 【假设有 8 个注意力头,两个尺度, 那每
一个尺度在计算注意力的时候都分成 4 个头, 大家可以详细的看看注释, 加油! 】

for i in range(self.num_dilation):

    # 执行 dilate_attention: q=(B,scale_d,H,W) k=(B,scale_d,H,W) v=(B,scale_d,H,W) 输出:
    x[i]=(B,H,W,scale_d)

    x[i] = self.dilate_attention[i](qkv[i][0], qkv[i][1], qkv[i][2])

# (scale_num,B,H,W,scale_d) --> (B,H,W,scale_num,scale_d) --> (B,H,W,scale_num*scale_d)=(B,H,W,C)

x = x.permute(1, 2, 3, 0, 4).reshape(B, H, W, C)

x = self.proj(x)

x = self.proj_drop(x)

return x

if __name__ == '__main__':

    # (B,H,W,C)

    input=torch.randn(1,16,16,512)

    Model = MultiDilateLocalAttention(dim=512)

    output=Model(input)

    print(output.shape)

```

# 12、DANet 模块

论文《Dual Attention Network for Scene Segmentation》

## 1、作用

双重注意力网络（DANet）旨在通过引入自注意力机制来捕获丰富的上下文依赖性，从而提升场景分割任务中的特征提取能力。通过在空间和通道维度上捕捉全局依赖关系，DANet 能够适应性地整合局部特征及其全局依赖，显著提高像素级识别的精确度。

## 2、机制

### 1、位置注意力模块：

通过自注意力机制，该模块能够捕获特征图中任意两个位置之间的空间依赖关系。对于给定位置的特征，通过加权求和的方式来聚合所有位置的特征，权重由相应位置之间的特征相似性决定。这使得即使在空间维度上距离较远的相似特征也能相互促进，增强特征的代表能力。

### 2、通道注意力模块：

类似地，该模块通过自注意力机制捕获不同通道图之间的依赖关系，每个通道图通过加权求和所有通道图的方式更新。这种机制使得模型能够强调相互依赖的通道图，进一步增强了特定语义的特征表示。

## 3、独特优势

### 1、全局依赖性的捕获：

DANet 通过位置和通道注意力模块有效捕获全局上下文信息，无论是空间上还是通道上的依赖关系，都能被模型准确识别和利用。

### 2、精确的像素级识别：

通过整合局部特征及其全局依赖，DANet 在场景分割任务中实现了更精确的像素级识别，尤其是在处理细节和对象边界方面表现出色。

### 3、跨多个数据集的优异表现：

DANet 在 Cityscapes、PASCAL VOC 2012、PASCAL Context 和 COCO Stuff 等多个挑战性的场景分割数据集上均取得了先进的性能，证明了其强大的泛化能力和有效性。

## 4、代码

```
import numpy as np

import torch

from torch import nn

from torch.nn import init

# 双重注意力网络（DANet）用于场景分割任务的实现

class ScaledDotProductAttention(nn.Module):

    '''
    实现缩放点积注意力机制。
    '''

    def __init__(self, d_model, d_k, d_v, h, dropout=.1):

        '''
        参数:

        :param d_model: 模型的输出维度

        :param d_k: 查询和键的维度

        :param d_v: 值的维度

        :param h: 头的数量

        '''

        super(ScaledDotProductAttention, self).__init__()
```



```
self.fc_q = nn.Linear(d_model, h * d_k)
```

```
self.fc_k = nn.Linear(d_model, h * d_k)
```

```
self.fc_v = nn.Linear(d_model, h * d_v)
```

```
self.fc_o = nn.Linear(h * d_v, d_model)
```

```
self.dropout=nn.Dropout(dropout)
```

```
self.d_model = d_model
```

```
self.d_k = d_k
```

```
self.d_v = d_v
```

```
self.h = h
```

```
self.init_weights()
```

```
def init_weights(self):
```

```
    for m in self.modules():
```

```
        if isinstance(m, nn.Conv2d):
```

```
            init.kaiming_normal_(m.weight, mode='fan_out')
```

```
            if m.bias is not None:
```

```
                init.constant_(m.bias, 0)
```

```
            elif isinstance(m, nn.BatchNorm2d):
```

```
                init.constant_(m.weight, 1)
```

```
init.constant_(m.bias, 0)
```

```
elif isinstance(m, nn.Linear):
```

```
init.normal_(m.weight, std=0.001)
```

```
if m.bias is not None:
```

```
init.constant_(m.bias, 0)
```

```
def forward(self, queries, keys, values, attention_mask=None, attention_weights=None):
```

```
'''
```

前向传播函数

参数:

:param queries: 查询 (b\_s, nq, d\_model)

:param keys: 键 (b\_s, nk, d\_model)

:param values: 值 (b\_s, nk, d\_model)

:param attention\_mask: 注意力遮罩 (b\_s, h, nq, nk), 掩码为 True 的地方将被忽略

:param attention\_weights: 注意力权重 (b\_s, h, nq, nk)

```
'''
```

```
b_s, nq = queries.shape[:2]
```

```
nk = keys.shape[1]
```

```
q = self.fc_q(queries).view(b_s, nq, self.h, self.d_k).permute(0, 2, 1, 3) # (b_s, h, nq, d_k)
```

```
k = self.fc_k(keys).view(b_s, nk, self.h, self.d_k).permute(0, 2, 3, 1) # (b_s, h, d_k, nk)
```

```
v = self.fc_v(values).view(b_s, nk, self.h, self.d_v).permute(0, 2, 1, 3) # (b_s, h, nk, d_v)
```

```
att = torch.matmul(q, k) / np.sqrt(self.d_k) # (b_s, h, nq, nk)
```

```
if attention_weights is not None:
```

```
    att = att * attention_weights
```

```
if attention_mask is not None:
```

```
    att = att.masked_fill(attention_mask, -np.inf)
```

```
att = torch.softmax(att, -1)
```

```
att=self.dropout(att)
```

```
out = torch.matmul(att, v).permute(0, 2, 1, 3).contiguous().view(b_s, nq, self.h * self.d_v) # (b_s, nq, h*d_v)
```

```
out = self.fc_o(out) # (b_s, nq, d_model)
```

```
return out
```

```
class SimplifiedScaledDotProductAttention(nn.Module):
```

```
    '''
```

```
    Scaled dot-product attention
```

```
    '''
```

```
    def __init__(self, d_model, h, dropout=.1):
```

```
        '''
```

```
        :param d_model: Output dimensionality of the model
```

```
:param d_k: Dimensionality of queries and keys
```

```
:param d_v: Dimensionality of values
```

```
:param h: Number of heads
```

```
'''
```

```
super(SimplifiedScaledDotProductAttention, self).__init__()
```

```
self.d_model = d_model
```

```
self.d_k = d_model//h
```

```
self.d_v = d_model//h
```

```
self.h = h
```

```
self.fc_o = nn.Linear(h * self.d_v, d_model)
```

```
self.dropout=nn.Dropout(dropout)
```

```
self.init_weights()
```

```
def init_weights(self):
```

```
    for m in self.modules():
```

```
        if isinstance(m, nn.Conv2d):
```

```
init.kaiming_normal_(m.weight, mode='fan_out')
```

```
if m.bias is not None:
```

```
    init.constant_(m.bias, 0)
```

```
elif isinstance(m, nn.BatchNorm2d):
```

```
    init.constant_(m.weight, 1)
```

```
    init.constant_(m.bias, 0)
```

```
elif isinstance(m, nn.Linear):
```

```
    init.normal_(m.weight, std=0.001)
```

```
    if m.bias is not None:
```

```
        init.constant_(m.bias, 0)
```

```
def forward(self, queries, keys, values, attention_mask=None, attention_weights=None):
```

```
    '''
```

```
    Computes
```

```
    :param queries: Queries (b_s, nq, d_model)
```

```
    :param keys: Keys (b_s, nk, d_model)
```

```
    :param values: Values (b_s, nk, d_model)
```

```
    :param attention_mask: Mask over attention values (b_s, h, nq, nk). True indicates masking.
```

```
    :param attention_weights: Multiplicative weights for attention values (b_s, h, nq, nk).
```

```
    :return:
```

```
    '''
```

```
    b_s, nq = queries.shape[:2]
```

```
nk = keys.shape[1]
```

```
q = queries.view(b_s, nq, self.h, self.d_k).permute(0, 2, 1, 3) # (b_s, h, nq, d_k)
```

```
k = keys.view(b_s, nk, self.h, self.d_k).permute(0, 2, 3, 1) # (b_s, h, d_k, nk)
```

```
v = values.view(b_s, nk, self.h, self.d_v).permute(0, 2, 1, 3) # (b_s, h, nk, d_v)
```

```
att = torch.matmul(q, k) / np.sqrt(self.d_k) # (b_s, h, nq, nk)
```

```
if attention_weights is not None:
```

```
    att = att * attention_weights
```

```
if attention_mask is not None:
```

```
    att = att.masked_fill(attention_mask, -np.inf)
```

```
att = torch.softmax(att, -1)
```

```
att=self.dropout(att)
```

```
out = torch.matmul(att, v).permute(0, 2, 1, 3).contiguous().view(b_s, nq, self.h * self.d_v) # (b_s, nq, h*d_v)
```

```
out = self.fc_o(out) # (b_s, nq, d_model)
```

```
return out
```

```
class PositionAttentionModule(nn.Module):
```

```
    def __init__(self,d_model=512,kernel_size=3,H=7,W=7):
```

```
super().__init__()
```

```
self.cnn=nn.Conv2d(d_model,d_model,kernel_size=kernel_size,padding=(kernel_size-1)//2)
```

```
self.pa=ScaledDotProductAttention(d_model,d_k=d_model,d_v=d_model,h=1)
```

```
def forward(self,x):
```

```
    # (B, C, H, W)
```

```
    B, C, H, W=x.shape
```

```
    y=self.cnn(x) # (B, C, H, W) --> (B, C, H, W)
```

```
    y=y.view(B,C,-1).permute(0,2,1) # (B, C, H, W) --> (B,C,N)-->(B,N,C)    N=H*W
```

```
    y=self.pa(y,y,y) #(B,N,C)
```

```
    return y
```

```
class ChannelAttentionModule(nn.Module):
```

```
    def __init__(self,d_model=512,kernel_size=3,H=7,W=7):
```

```
        super().__init__()
```

```
        self.cnn=nn.Conv2d(d_model,d_model,kernel_size=kernel_size,padding=(kernel_size-1)//2)
```

```
        self.pa=SimplifiedScaledDotProductAttention(H*W,h=1)
```

```
    def forward(self,x):
```

```
        # (B, C, H, W)
```

```
B,C,H,W=x.shape
```

```
y=self.cnn(x) # (B, C, H, W) --> (B, C, H, W)
```

```
y=y.view(B,C,-1) # (B, C, H, W)-->(B, C, N) N=H*W
```

```
y=self.pa(y,y,y) # (B, C, N)
```

```
return y
```

```
class DAModule(nn.Module):
```

```
def __init__(self,d_model=512,kernel_size=3,H=7,W=7):
```

```
    super().__init__()
```

```
    # 位置注意力和通道注意力的区别就是：通道注意力没有通过卷积操作生成 qkv
```

```
    self.position_attention_module=PositionAttentionModule(d_model=512,kernel_size=3,H=7,W=7)
```

```
    self.channel_attention_module=ChannelAttentionModule(d_model=512,kernel_size=3,H=7,W=7)
```

```
def forward(self,input):
```

```
    # (B, C, H, W)
```

```
    B,C,H,W=input.shape
```

```
    p_out=self.position_attention_module(input) # 执行位置注意力: (B, C, H, W)-->(B,N,C)
```

```
    c_out=self.channel_attention_module(input) # 执行通道注意力:(B, C, H, W)--> (B, C, N)
```



```
p_out=p_out.permute(0,2,1).view(B,C,H,W) #(B,N,C)-->(B,C,N)-->(B,C,H,W)
```

```
c_out=c_out.view(B,C,H,W) # (B,C,N)-->(B,C,H,W)
```

```
return p_out+c_out
```

# 两个注意力机制就不细讲了哦，基本一模一样，只不过通道注意力没有通过卷积生成新的 **qkv**，作者说会破坏原有通道之间的相关性。

```
if __name__ == '__main__':
```

```
    # (B, C, H, W)
```

```
    input=torch.randn(1,512,7,7)
```

```
    Model=DAModule(d_model=512,kernel_size=3,H=7,W=7)
```

```
    output = Model(input)
```

```
    print(output.shape)
```

## 13、EMA 模块

论文《Efficient Multi-Scale Attention Module with Cross-Spatial Learning》

### 1、作用

论文提出了一种新颖的高效多尺度注意力（EMA）模块，专注于在保留每个通道信息的同时降低计算成本。EMA 模块通过将部分通道重塑到批量维度并将通道维度分组为多个子特征，使得空间语义特征在每个特征组内得到良好分布。该模块的设计旨在提高图像分类和对象检测任务中的特征提取能力，通过编码全局信息来重新校准每个并行分支中的通道权重，并通过跨维度交互进一步聚合两个并行分支的输出特征，捕获像素级的成对关系。

### 2、机制

1、EMA 模块：

通过分组结构修改了坐标注意力（CA）的顺序处理方法，提出了一个不进行维度降低的高效多尺度注意力模块。EMA 模块通过在不同空间维度上进行特征分组和多尺度结构处理，有效地建立了短程和长程依赖关系，以实现更好的性能。

## 2、跨空间学习方法：

EMA 模块通过跨空间学习方法，将两个并行子网络的输出特征图融合，这种方法使用矩阵点积操作来捕获像素级的成对关系，并突出全局上下文，以丰富特征聚合。

## 3、独特优势

### 1、高效的多尺度感知能力：

EMA 模块通过结合 1x1 和 3x3 卷积核的并行子网络，有效捕获不同尺度的空间信息，同时保留精确的空间结构信息。

### 2、跨空间特征融合：

通过跨空间学习方法，EMA 能够有效整合来自不同空间位置的特征信息，提高了特征表示的丰富性和准确性。

### 3、参数效率和计算效率：

与现有的注意力机制相比，EMA 在提高性能的同时，还实现了更低的参数数量和计算复杂度，特别是在进行图像分类和对象检测任务时。

## 4、代码

```
import torch

from torch import nn

# 定义 EMA 模块

class EMA(nn.Module):

    def __init__(self, channels, factor=8):

        super(EMA, self).__init__()
```

```
# 设置分组数量，用于特征分组
```

```
self.groups = factor
```

```
# 确保分组后的通道数大于 0
```

```
assert channels // self.groups > 0
```

```
# softmax 激活函数，用于归一化
```

```
self.softmax = nn.Softmax(-1)
```

```
# 全局平均池化，生成通道描述符
```

```
self.agp = nn.AdaptiveAvgPool2d((1, 1))
```

```
# 水平方向的平均池化，用于编码水平方向的全局信息
```

```
self.pool_h = nn.AdaptiveAvgPool2d((None, 1))
```

```
# 垂直方向的平均池化，用于编码垂直方向的全局信息
```

```
self.pool_w = nn.AdaptiveAvgPool2d((1, None))
```

```
# GroupNorm 归一化，减少内部协变量偏移
```

```
self.gn = nn.GroupNorm(channels // self.groups, channels // self.groups)
```

```
# 1x1 卷积，用于学习跨通道的特征
```

```
self.conv1x1 = nn.Conv2d(channels // self.groups, channels // self.groups, kernel_size=1, stride=1,  
padding=0)
```

```
# 3x3 卷积，用于捕捉更丰富的空间信息
```

```
self.conv3x3 = nn.Conv2d(channels // self.groups, channels // self.groups, kernel_size=3, stride=1,  
padding=1)
```

```
def forward(self, x):
```

```
    b, c, h, w = x.size()
```

```
# 对输入特征图进行分组处理
```

```
group_x = x.reshape(b * self.groups, -1, h, w) # b*g,c//g,h,w
```

```
# 应用水平和垂直方向的全局平均池化
```

```
x_h = self.pool_h(group_x)
```

```
x_w = self.pool_w(group_x).permute(0, 1, 3, 2)
```

```
# 通过 1x1 卷积和 sigmoid 激活函数，获得注意力权重
```

```
hw = self.conv1x1(torch.cat([x_h, x_w], dim=2))
```

```
x_h, x_w = torch.split(hw, [h, w], dim=2)
```

```
# 应用 GroupNorm 和注意力权重调整特征图
```

```
x1 = self.gn(group_x * x_h.sigmoid() * x_w.permute(0, 1, 3, 2).sigmoid())
```

```
x2 = self.conv3x3(group_x)
```

```
# 将特征图通过全局平均池化和 softmax 进行处理，得到权重
```

```
x11 = self.softmax(self.agp(x1).reshape(b * self.groups, -1, 1).permute(0, 2, 1))
```

```
x12 = x2.reshape(b * self.groups, c // self.groups, -1) # b*g, c//g, hw
```

```
x21 = self.softmax(self.agp(x2).reshape(b * self.groups, -1, 1).permute(0, 2, 1))
```

```
x22 = x1.reshape(b * self.groups, c // self.groups, -1) # b*g, c//g, hw
```

```
# 通过矩阵乘法和 sigmoid 激活获得最终的注意力权重，调整特征图
```

```
weights = (torch.matmul(x11, x12) + torch.matmul(x21, x22)).reshape(b * self.groups, 1, h, w)
```

```
# 将调整后的特征图重塑回原始尺寸
```

```
return (group_x * weights.sigmoid()).reshape(b, c, h, w)
```

```
# 测试 EMA 模块
```

```
if __name__ == '__main__':  
  
    block = EMA(64).cuda() # 实例化 EMA 模块，并移至 CUDA 设备  
  
    input = torch.rand(1, 64, 64, 64).cuda() # 创建随机输入数据  
  
    output = block(input) # 前向传播  
  
    print(output.shape) # 打印输入和输出的尺寸
```

# 14、Sea 模块

论文《SeaFormer: Squeeze-Enhanced Axial Transformer for Mobile Semantic Segmentation》

## 1、作用

SeaFormer 旨在为移动设备上的语义分割任务提供一种新颖的方法，通过设计一个 squeeze-enhanced Axial Transformer（SeaFormer），能够在保持低延迟的同时，实现高效的计算和内存使用。该模型特别强调在移动设备上对高分辨率图像进行逐像素的语义分割，通过精心设计的注意力机制，实现对全局上下文信息的有效捕捉和利用。

## 2、机制

### 1、Squeeze-Enhanced Axial Attention（SEA Attention）：

SeaFormer 的核心是 SEA Attention 模块，该模块结合了轴向注意力机制和细节增强技术。通过沿水平和垂直轴压缩输入特征图，减少了计算复杂度，同时通过深度可分离卷积增强了局部细节，保证了特征的丰富性和有效性。

### 2、轻量级分割头：

为了进一步减少计算成本和提高推理速度，SeaFormer 采用了简化的分割头，该分割头通过少量的卷积层实现特征图的最终语义分割，有效平衡了性能和效率。

## 3、独特优势

### 1、高效性能和低延迟：

通过在 SEA Attention 中引入 squeeze 操作和细节增强策略，SeaFormer 显著降低了模型的计算复杂度和内存需求，特别是在处理高分辨率图像时，能够在 ARM 基础移动设备上实现低延迟的实时语义分割。

## 2、移动友好的设计：

SeaFormer 专为移动设备优化，其模型结构和参数都经过精心设计，以适应移动设备的硬件限制，包括有限的计算能力和内存资源。

## 3、通用性和灵活性：

尽管 SeaFormer 主要针对移动语义分割任务设计，但其模型架构的通用性和灵活性也使其能够轻松扩展到其他视觉识别任务，如图像分类，展现了作为多功能移动友好骨干网络的潜力。

## 4、代码

```
import numpy as np

import torch

import torch.nn.functional as F

from torch import nn

from mmcv.cnn import build_norm_layer

#这里的 mmcv 安装包用以下的指令

" pip install mmcv==2.0.0 -f https://download.openmmlab.com/mmcv/dist/cu117/torch1.13/index.html  "

class Conv2d_BN(nn.Sequential):

    def __init__(self, a, b, ks=1, stride=1, pad=0, dilation=1,

                groups=1, bn_weight_init=1,

                norm_cfg=dict(type='BN', requires_grad=True)):

        super().__init__()

        self.inp_channel = a
```

```
self.out_channel = b

self.ks = ks

self.pad = pad

self.stride = stride

self.dilation = dilation

self.groups = groups


self.add_module('c', nn.Conv2d(

    a, b, ks, stride, pad, dilation, groups, bias=False))

bn = build_norm_layer(norm_cfg, b)[1]

nn.init.constant_(bn.weight, bn_weight_init)

nn.init.constant_(bn.bias, 0)

self.add_module('bn', bn)
```

```
class SqueezeAxialPositionalEmbedding(nn.Module):
```

```
    def __init__(self, dim, shape):
```

```
        super().__init__()
```

```
        self.pos_embed = nn.Parameter(torch.randn([1, dim, shape]), requires_grad=True)
```

```

def forward(self, x):

    # (B,C_qk,H)

    B, C, N = x.shape

    x = x + F.interpolate(self.pos_embed, size=(N), mode='linear', align_corners=False)

    return x

```

```

class Sea_Attention(torch.nn.Module):

    def __init__(self, dim, key_dim, num_heads,

                attn_ratio=2,

                activation=nn.ReLU,

                norm_cfg=dict(type='BN', requires_grad=True), ):

        super().__init__()

        self.num_heads = num_heads

        self.scale = key_dim ** -0.5

        self.key_dim = key_dim

        self.nh_kd = nh_kd = key_dim * num_heads # num_head key_dim

        self.d = int(attn_ratio * key_dim)

        self.dh = int(attn_ratio * key_dim) * num_heads

        self.attn_ratio = attn_ratio

```



```

self.to_q = Conv2d_BN(dim, nh_kd, 1, norm_cfg=norm_cfg)

self.to_k = Conv2d_BN(dim, nh_kd, 1, norm_cfg=norm_cfg)

self.to_v = Conv2d_BN(dim, self.dh, 1, norm_cfg=norm_cfg)


self.proj = torch.nn.Sequential(activation(), Conv2d_BN(

    self.dh, dim, bn_weight_init=0, norm_cfg=norm_cfg))

self.proj_encode_row = torch.nn.Sequential(activation(), Conv2d_BN(

    self.dh, self.dh, bn_weight_init=0, norm_cfg=norm_cfg))

self.pos_emb_rowq = SqueezeAxialPositionalEmbedding(nh_kd, 16)

self.pos_emb_rowk = SqueezeAxialPositionalEmbedding(nh_kd, 16)

self.proj_encode_column = torch.nn.Sequential(activation(), Conv2d_BN(

    self.dh, self.dh, bn_weight_init=0, norm_cfg=norm_cfg))

self.pos_emb_columnq = SqueezeAxialPositionalEmbedding(nh_kd, 16)

self.pos_emb_columnk = SqueezeAxialPositionalEmbedding(nh_kd, 16)


self.dwconv = Conv2d_BN(2 * self.dh, 2 * self.dh, ks=3, stride=1, pad=1, dilation=1,

    groups=2 * self.dh, norm_cfg=norm_cfg)

self.act = activation()

self.pwconv = Conv2d_BN(2 * self.dh, dim, ks=1, norm_cfg=norm_cfg)

# self.sigmoid = torch.sigmoid()

```

```

def forward(self, x): # x (B,N,C)

```

```

B, C, H, W = x.shape

q = self.to_q(x) # 生成 query: (B,C,H,W)-->(B,C_qk,H,W)

k = self.to_k(x) # 生成 key: (B,C,H,W)-->(B,C_qk,H,W)

v = self.to_v(x) # 生成 value: (B,C,H,W)-->(B,C_v,H,W)

# Detail enhancement kernel

qkv = torch.cat([q, k, v], dim=1) # 将 qkv 拼接: (B,2*C_qk+C_v,H,W)

qkv = self.act(self.dwconv(qkv)) # 执行 3x3 卷积,建模局部空间依赖,从而增强局部细节感知:
(B,2*C_qk+C_v,H,W)-->(B,2*C_qk+C_v,H,W)

qkv = self.pwconv(qkv) # 执行 1x1 卷积,将通道数量从(2*C_qk+C_v)映射到 C,从而生成细节增强特征:
(B,2C_qk+C_v,H,W)-->(B,C,H,W)

# squeeze axial attention

## squeeze row, squeeze 操作将全局信息保留到单个轴上, 然后分别应用自注意力建模对应轴的长期依赖

qrow = self.pos_emb_row(q.mean(-1)).reshape(B, self.num_heads, -1, H).permute(0, 1, 3, 2) #通过平均
池化压缩水平方向,并为垂直方向的空间位置添加位置嵌入:
(B,C_qk,H,W)-->mean-->(B,C_qk,H)-->reshape-->(B,h,d,H)-->permute-->(B,h,H,d); C_qk=h*d, h:注意力头的个数; d:
每个注意力头的通道数

krow = self.pos_emb_row(k.mean(-1)).reshape(B, self.num_heads, -1, H) #通过平均池化压缩水平方向,并为
垂直方向的空间位置添加位置嵌入: (B,C_qk,H,W)-->mean-->(B,C_qk,H)-->reshape-->(B,h,d,H)

vrow = v.mean(-1).reshape(B, self.num_heads, -1, H).permute(0, 1, 3, 2) #通过平均池化压缩水平方向:
(B,C_v,H,W)-->mean-->(B,C_v,H)-->reshape-->(B,h,d_v,H)-->permute-->(B,h,H,d_v); C_v=h*d_v, h:注意力头的个
数; d_v:Value 矩阵中每个注意力头的通道数

attn_row = torch.matmul(qrow, krow) * self.scale # 计算水平方向压缩之后的自注意力机制: (B,h,H,d) @
(B,h,d,H) = (B,h,H,H)

```

```

attn_row = attn_row.softmax(dim=-1) # 执行 softmax 操作

xx_row = torch.matmul(attn_row, vrow) # 对 Value 进行加权求和: (B,h,H,H) @ (B,h,H,d_v) = (B,h,H,d_v)

xx_row = self.proj_encode_row(xx_row.permute(0, 1, 3, 2).reshape(B, self.dh, H, 1)) # 对注意力机制的
输出进行 reshape 操作,并进行卷积: (B,h,H,d_v)-->permute-->(B,h,d_v,H)-->reshape-->(B,C_v,H,1); C_v=h*d_v

## squeeze column

qcolumn = self.pos_emb_column(q.mean(-2)).reshape(B, self.num_heads, -1, W).permute(0, 1, 3, 2) # 通
过平均池化压缩垂直方向,并为水平方向的空间位置添加位置嵌入:
(B,C_qk,H,W)-->mean-->(B,C_qk,W)-->reshape-->(B,h,d,W)-->permute-->(B,h,W,d); C_qk=h*d, h:注意力头的个数; d:
每个注意力头的通道数

kcolumn = self.pos_emb_column(k.mean(-2)).reshape(B, self.num_heads, -1, W) # 通过平均池化压缩垂直方
向,并为水平方向的空间位置添加位置嵌入: (B,C_qk,H,W)-->mean-->(B,C_qk,W)-->reshape-->(B,h,d,W)

vcolumn = v.mean(-2).reshape(B, self.num_heads, -1, W).permute(0, 1, 3, 2) #通过平均池化压缩垂直方向:
(B,C_v,H,W)-->mean-->(B,C_v,W)-->reshape-->(B,h,d_v,W)-->permute-->(B,h,W,d_v)

attn_column = torch.matmul(qcolumn, kcolumn) * self.scale # 计算垂直方向压缩之后的自注意力机制: (B,h,W,d)
@ (B,h,d,W) = (B,h,W,W)

attn_column = attn_column.softmax(dim=-1) # 执行 softmax 操作

xx_column = torch.matmul(attn_column, vcolumn) # 对 Value 进行加权求和: (B,h,W,W) @ (B,h,W,d_v) =
(B,h,W,d_v)

xx_column = self.proj_encode_column(xx_column.permute(0, 1, 3, 2).reshape(B, self.dh, 1, W)) # 对注
意力机制的输出进行 reshape 操作,并进行卷积: (B,h,W,d_v)-->permute-->(B,h,d_v,W)-->reshape-->(B,C_v,1,W);
C_v=h*d_v

xx = xx_row.add(xx_column) # 将两个注意力机制的输出进行相加,这是一种 broadcast 操作: (B,C_v,H,1) +
(B,C_v,1,W) =(B,C_v,H,W)

xx = v.add(xx) # 添加残差连接

xx = self.proj(xx) # 应用 1x1Conv 得到 Squeeze Axial attention 的输出

```

```
xx = xx.sigmoid() * qkv # 为 Squeeze Axial attention 的输出应用门控机制获得权重，然后与 Detail enhancement  
kernel 的输出进行逐点乘法
```

```
return xx
```

```
if __name__ == '__main__':
```

```
# (B,C,H,W)
```

```
input=torch.randn(1,512,7,7)
```

```
Model = Sea_Attention(dim=512, key_dim=64, num_heads=8)
```

```
output=Model(input)
```

```
print(output.shape)
```

## 15、CoTAttention 模块

论文《Contextual Transformer Networks for Visual Recognition》

### 1、作用

Contextual Transformer (CoT) 网络旨在通过利用输入键之间的上下文信息来引导动态注意力矩阵的学习，从而增强视觉识别任务中的视觉表示能力。通过在不同的尺度上捕捉丰富的上下文依赖性，CoT 块能够更有效地捕获图像的局部和全局信息，提高模型对图像特征的理解能力。

### 2、机制

## 1、CoT 块：

首先通过 3x3 卷积上下文编码输入键，产生静态上下文表示。然后，将编码后的键与输入查询连接，通过两个连续的 1x1 卷积学习动态多头注意力矩阵。学到的注意力矩阵乘以输入值，以获得输入的动态上下文表示。最终，将静态和动态上下文表示的融合作为输出。

## 2、CoT 网络：

通过在 ResNet 架构中替换每个 3x3 卷积，使用 CoT 块构建了名为 Contextual Transformer Networks (CoT-Net) 的 Transformer 风格的骨干网络。CoT-Net 通过在不同层次上整合上下文信息和自注意力学习，形成了一个强大的视觉识别模型。

## 3、独特优势

### 1、上下文信息的充分利用：

CoT 网络通过在自注意力机制中整合键之间的上下文信息，提高了模型捕捉局部和全局依赖性的能力，从而增强了视觉表示的丰富性和准确性。

### 2、高效的计算性能：

相比于传统的自注意力机制，CoT 块在不增加参数和计算复杂度的情况下，有效地增强了模型的视觉表示能力。CoT 网络在多个视觉识别任务中展现了卓越的性能，同时保持了较低的计算成本。

### 3、灵活性与泛化能力：

CoT 网络可以轻松地集成到现有的 ResNet 架构中，并且在图像识别、对象检测和实例分割等多个视觉识别任务中表现出良好的泛化能力，证明了其作为一个强大骨干网络的潜力。

## 4、代码

```
# 导入必要的 PyTorch 模块

import torch

from torch import nn

from torch.nn import functional as F
```

```
class CoTAttention(nn.Module):
```

```
    # 初始化 CoT 注意力模块
```

```
    def __init__(self, dim=512, kernel_size=3):
```

```
        super().__init__()
```

```
        self.dim = dim # 输入的通道数
```

```
        self.kernel_size = kernel_size # 卷积核大小
```

```
        # 定义用于键(key)的卷积层，包括一个分组卷积，BatchNorm 和 ReLU 激活
```

```
        self.key_embed = nn.Sequential(
```

```
            nn.Conv2d(dim, dim, kernel_size=kernel_size, padding=kernel_size//2, groups=4, bias=False),
```

```
            nn.BatchNorm2d(dim),
```

```
            nn.ReLU()
```

```
        # 定义用于值(value)的卷积层，包括一个 1x1 卷积和 BatchNorm
```

```
        self.value_embed = nn.Sequential(
```

```
            nn.Conv2d(dim, dim, 1, bias=False),
```

```
            nn.BatchNorm2d(dim)
```

```
        # 缩小因子，用于降低注意力嵌入的维度
```

```
        factor = 4
```

```
# 定义注意力嵌入层，由两个卷积层、一个 BatchNorm 层和 ReLU 激活组成
```

```
self.attention_embed = nn.Sequential(
```

```
    nn.Conv2d(2*dim, 2*dim//factor, 1, bias=False),
```

```
    nn.BatchNorm2d(2*dim//factor),
```

```
    nn.ReLU(),
```

```
    nn.Conv2d(2*dim//factor, kernel_size*kernel_size*dim, 1)
```

```
)
```

```
def forward(self, x):
```

```
    # 前向传播函数
```

```
    bs, c, h, w = x.shape # 输入特征的尺寸
```

```
    k1 = self.key_embed(x) # 生成键的静态表示
```

```
    v = self.value_embed(x).view(bs, c, -1) # 生成值的表示并调整形状
```

```
    y = torch.cat([k1, x], dim=1) # 将键的静态表示和原始输入连接
```

```
    att = self.attention_embed(y) # 生成动态注意力权重
```

```
    att = att.reshape(bs, c, self.kernel_size*self.kernel_size, h, w)
```

```
    att = att.mean(2, keepdim=False).view(bs, c, -1) # 计算注意力权重的均值并调整形状
```

```
    k2 = F.softmax(att, dim=-1) * v # 应用注意力权重到值上
```

```
    k2 = k2.view(bs, c, h, w) # 调整形状以匹配输出
```

```
    return k1 + k2 # 返回键的静态和动态表示的总和
```

```
# 实例化 CoTAttention 模块并测试
```

```
if __name__ == '__main__':
```

```
    block = CoTAttention(64) # 创建一个输入通道数为 64 的 CoTAttention 实例
```

```
    input = torch.rand(1, 64, 64, 64) # 创建一个随机输入
```

```
    output = block(input) # 通过 CoTAttention 模块处理输入
```

```
    print(output.shape) # 打印输入和输出的尺寸
```

## 16、EMSA 模块

论文《ResT: An Efficient Transformer for Visual Recognition》

### 1、作用

ResT 是一种高效的多尺度视觉 Transformer，作为图像识别领域的通用骨架。与现有的 Transformer 方法相比，ResT 在处理不同分辨率的原始图像时具有多个优点：(1) 构建了一个内存高效的多头自注意力机制，通过简单的深度卷积来压缩内存，并在保持多头注意力的多样性能力的同时，跨注意力头维度进行交互；(2) 位置编码被设计为空间注意力，更加灵活，可以处理任意大小的输入图像，而无需插值或微调；(3) 与在每个阶段开始时直接对原始图像进行分块（tokenization）不同，设计了将分块嵌入作为一系列重叠卷积操作的堆栈，实现了更有效的特征提取。

### 2、机制

#### 1、多头自注意力压缩：

ResT 通过简单的深度卷积操作压缩内存，并在注意力头维度进行交互，减少了 MSA 在 Transformer 块中的计算和内存需求。

#### 2、空间注意力的位置编码：

ResT 采用位置编码作为空间注意力，使模型能够灵活地处理不同尺寸的输入图像。



### 3、重叠卷积的分块嵌入：

通过设计分块嵌入为重叠卷积操作的堆栈，ResT 在不同阶段有效地提取特征，创建了多尺度的特征金字塔。

## 3、独特优势

### 1、高效和灵活性：

ResT 通过引入压缩的多头自注意力机制和空间注意力的位置编码，在保持计算效率的同时，提供了处理不同分辨率图像的灵活性。

### 2、改进的特征提取能力：

通过重叠卷积的分块嵌入，ResT 能够更有效地捕获图像中的局部和全局信息，提高了模型对图像特征的理解能力。

### 3、通用性：

ResT 作为一个通用骨架，在图像分类和下游任务（如对象检测和实例分割）上展现了卓越的性能，证明了其作为强大骨架网络的潜力。

## 4、代码

```
import numpy as np

import torch

from torch import nn

from torch.nn import init


# EMSA 模块，为多尺度注意力机制的实现


class EMSA(nn.Module):


    def __init__(self, d_model, d_k, d_v, h, dropout=.1, H=7, W=7, ratio=3, apply_transform=True):
```

```

super(EMSA, self).__init__()

# 初始化参数和层

self.H = H# 输入特征图的高度

self.W = W# 输入特征图的宽度

self.fc_q = nn.Linear(d_model, h * d_k)# 查询向量的全连接层

self.fc_k = nn.Linear(d_model, h * d_k) # 键向量的全连接层

self.fc_v = nn.Linear(d_model, h * d_v)# 值向量的全连接层

self.fc_o = nn.Linear(h * d_v, d_model)# 输出的全连接层

self.dropout = nn.Dropout(dropout)# Dropout 层


self.ratio = ratio # 空间降采样比例

if (self.ratio > 1):

    # 如果空间降采样比例大于 1，则添加空间降采样层

    self.sr = nn.Sequential()

    self.sr_conv = nn.Conv2d(d_model, d_model, kernel_size=ratio + 1, stride=ratio, padding=ratio //

2,

                                groups=d_model)

    self.sr_ln = nn.LayerNorm(d_model)

self.apply_transform = apply_transform and h > 1

if (self.apply_transform):

    # 如果应用变换，则添加变换层，包括 1x1 卷积、Softmax 和实例归一化

    self.transform = nn.Sequential()

```

```

        self.transform.add_module('conv', nn.Conv2d(h, h, kernel_size=1, stride=1))

        self.transform.add_module('softmax', nn.Softmax(-1))

        self.transform.add_module('in', nn.InstanceNorm2d(h))


    self.d_model = d_model

    self.d_k = d_k

    self.d_v = d_v

    self.h = h

    self.init_weights()

def init_weights(self):

    for m in self.modules():

        if isinstance(m, nn.Conv2d):

            init.kaiming_normal_(m.weight, mode='fan_out')

            if m.bias is not None:

                init.constant_(m.bias, 0)

        elif isinstance(m, nn.BatchNorm2d):

            init.constant_(m.weight, 1)

            init.constant_(m.bias, 0)

        elif isinstance(m, nn.Linear):

            init.normal_(m.weight, std=0.001)

```

```

        if m.bias is not None:

            init.constant_(m.bias, 0)


def forward(self, queries, keys, values, attention_mask=None, attention_weights=None):

    b_s, nq, c = queries.shape

    nk = keys.shape[1]

    q = self.fc_q(queries).view(b_s, nq, self.h, self.d_k).permute(0, 2, 1, 3) # (b_s, h, nq, d_k)

    if (self.ratio > 1):

        x = queries.permute(0, 2, 1).view(b_s, c, self.H, self.W) # bs,c,H,W

        x = self.sr_conv(x) # bs,c,h,w

        x = x.contiguous().view(b_s, c, -1).permute(0, 2, 1) # bs,n',c

        x = self.sr_ln(x)

        k = self.fc_k(x).view(b_s, -1, self.h, self.d_k).permute(0, 2, 3, 1) # (b_s, h, d_k, n')

        v = self.fc_v(x).view(b_s, -1, self.h, self.d_v).permute(0, 2, 1, 3) # (b_s, h, n', d_v)

    else:

        k = self.fc_k(keys).view(b_s, nk, self.h, self.d_k).permute(0, 2, 3, 1) # (b_s, h, d_k, nk)

        v = self.fc_v(values).view(b_s, nk, self.h, self.d_v).permute(0, 2, 1, 3) # (b_s, h, nk, d_v)

    if (self.apply_transform):

```

```

        att = torch.matmul(q, k) / np.sqrt(self.d_k) # (b_s, h, nq, n')

        att = self.transform(att) # (b_s, h, nq, n')

    else:

        att = torch.matmul(q, k) / np.sqrt(self.d_k) # (b_s, h, nq, n')

        att = torch.softmax(att, -1) # (b_s, h, nq, n')

    if attention_weights is not None:

        att = att * attention_weights

    if attention_mask is not None:

        att = att.masked_fill(attention_mask, -np.inf)

    att = self.dropout(att)

    out = torch.matmul(att, v).permute(0, 2, 1, 3).contiguous().view(b_s, nq, self.h * self.d_v) # (b_s,
nq, h*d_v)

    out = self.fc_o(out) # (b_s, nq, d_model)

    return out

if __name__ == '__main__':

    block = EMSA(d_model=512, d_k=512, d_v=512, h=8, H=8, W=8, ratio=2, apply_transform=True).cuda()

    input = torch.rand(64, 64, 512).cuda()

    output = block(input, input, input)

    print(output.shape)

```

# 17、ExternalAttention 模块

论文《Beyond Self-attention: External Attention using Two Linear Layers for Visual Tasks》

## 1、作用

本文提出了一种新颖的注意力机制——外部注意力（External Attention），通过使用两个外部小型可学习的共享内存来实现。这种机制能够用两个连续的线性层和两个归一化层简单实现，并且可以方便地替换现有流行架构中的自注意力机制。外部注意力具有线性复杂度，并且隐式地考虑了所有数据样本之间的关联性，为图像分类、目标检测、语义分割、实例分割、图像生成以及点云分析等视觉任务提供了与自注意力机制相当或优于的性能，同时大幅降低了计算和内存成本。

## 2、机制

### 1、外部注意力机制：

与自注意力不同，外部注意力通过计算输入特征与两个外部学习内存之间的亲和力来更新特征，这两个外部内存存在整个数据集上共享，能够捕捉到跨数据集的全局上下文，提升注意力机制的泛化能力。

### 2、线性复杂度：

外部注意力的计算复杂度为线性，通过减少内存中的元素数量，实现了对大规模输入的直接应用，显著提高了效率。

### 3、多头外部注意力：

通过引入多头机制，外部注意力能够捕获输入的不同方面的关系，增强了模型的表示能力。这种机制对于各种视觉任务都非常有效。

## 3、独特优势

### 1、高效且具有正则化作用：

外部注意力通过使用较少的参数和线性的计算复杂度，实现了高效的特征更新，并且由于内存单元是跨数据集共享的，因此具有强大的正则化作用，提高了模型的泛化能力。

### 2、跨样本的关联性考虑：

不同于自注意力仅关注单个样本内部的特征关联，外部注意力能够捕捉不同样本之间的潜在关联，为更好的特征表示提供了新的途径。

### 3、易于集成：

由于其简单性，外部注意力可以轻松地集成到现有的基于自注意力的架构中，为各种视觉任务提供性能提升的同时，减少计算和存储开销。

## 4、代码

```
import numpy as np

import torch

from torch import nn

from torch.nn import init

# 定义外部注意力类，继承自 nn.Module

class ExternalAttention(nn.Module):

    def __init__(self, d_model, S=64):

        super().__init__()

        # 初始化两个线性变换层，用于生成注意力映射

        # mk: 将输入特征从 d_model 维映射到 S 维，即降维到共享内存空间的大小

        self.mk = nn.Linear(d_model, S, bias=False)

        # mv: 将降维后的特征从 S 维映射回原始的 d_model 维

        self.mv = nn.Linear(S, d_model, bias=False)

        # 使用 Softmax 函数进行归一化处理
```

```
self.softmax = nn.Softmax(dim=1)

# 调用权重初始化函数

self.init_weights()


def init_weights(self):

    # 自定义权重初始化方法

    for m in self.modules():

        if isinstance(m, nn.Conv2d):

            # 对卷积层的权重进行 Kaiming 正态分布初始化

            init.kaiming_normal_(m.weight, mode='fan_out')

            if m.bias is not None:

                # 如果有偏置项，则将其初始化为 0

                init.constant_(m.bias, 0)

        elif isinstance(m, nn.BatchNorm2d):

            # 对批归一化层的权重和偏置进行常数初始化

            init.constant_(m.weight, 1)

            init.constant_(m.bias, 0)

        elif isinstance(m, nn.Linear):

            # 对线性层的权重进行正态分布初始化，偏置项（如果存在）初始化为 0

            init.normal_(m.weight, std=0.001)

            if m.bias is not None:

                init.constant_(m.bias, 0)
```



```

def forward(self, queries):

    # 前向传播函数

    attn = self.mk(queries) # 使用 mk 层将输入特征降维到 S 维

    attn = self.softmax(attn) # 对降维后的特征进行 Softmax 归一化处理

    # 对归一化后的注意力分数进行标准化，使其和为 1

    attn = attn / torch.sum(attn, dim=2, keepdim=True)

    out = self.mv(attn) # 使用 mv 层将注意力特征映射回原始维度

    return out

# 示例代码，创建一个 ExternalAttention 实例，并对一个随机输入进行处理

if __name__ == '__main__':

    block = ExternalAttention(d_model=64, S=8).cuda() # 实例化模型并移至 CUDA 设备

    input = torch.rand(64, 64, 64).cuda() # 创建随机输入

    output = block(input) # 通过模型传递输入

    print(output.shape) # 打印输入和输出的尺寸

```

## 18、PAM 模块

论文《Pyraformer: Low-Complexity Pyramidal Attention for Long-Range Time Series Modeling and Forecasting》

### 1、作用

Pyraformer 通过利用时间序列的多分辨率表示，实现了对长期时间序列建模和预测的低复杂性解决方案。它引入了金字塔式注意力模块（PAM），该模块通过跨尺度的树状结构总结不同分辨率的特征，并通过内尺度的相邻连接模拟不同范围的时间依赖性。这种方法有效地捕获了长短期模式，对单步和多步长范围预测任务均表现出高精度，同时显著降低了时间和内存消耗。

## 2、机制

### 1、金字塔式注意力模块（PAM）：

在 PAM 中，通过在金字塔图中基于注意力传递消息，构建了跨尺度和内尺度连接。跨尺度连接建立多分辨率时间序列的表示，而内尺度边缘通过连接相邻节点来捕获每个分辨率上的时间依赖性。

### 2、粗尺度构建模块（CSCM）：

该模块通过在对应的子节点上执行卷积，从底层向上层初始化金字塔图中粗尺度节点，为后续的 PAM 提供信息交换的基础。

### 3、预测模块：

根据下游任务的不同，采用不同的网络结构输出最终预测结果。对于单步预测，将历史序列的最后节点在所有尺度上聚合后进行预测；对于多步预测，提出了两种预测模块，一种是批量映射所有未来时间步，另一种是利用解码器基于完全注意力层进行预测。

## 3、独特优势

### 1、低时间和空间复杂度：

Pyraformer 在序列长度  $L$  的情况下，实现了时间和空间复杂度的线性增长（ $O(L)$ ），同时保持了信号传播路径的最大长度为常数（ $O(1)$ ），这对于处理长时间序列尤其重要。

### 2、高预测精度：

在多个真实世界数据集上的实验结果表明，Pyraformer 在单步和长范围多步预测任务中都能达到最高的预测精度，且在时间和内存消耗上均优于现有的 Transformer 及其变体。

### 3、多分辨率时间依赖性建模：

通过金字塔图有效捕获长短期时间依赖性，并在不同分辨率上模拟时间序列的动态，提供了一种紧凑的表示方法来处理远距离位置之间的时间依赖性。

## 4、代码

```
import torch

import torch.nn as nn

import torch.nn.functional as F

import numpy as np

import math

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")


def get_mask(input_size, window_size, inner_size, device):

    """Get the attention mask of PAM-Naive"""

    # Get the size of all layers

    all_size = []

    all_size.append(input_size)

    for i in range(len(window_size)):

        layer_size = math.floor(all_size[i] / window_size[i])

        all_size.append(layer_size)

    # 12 + 6 + 3 + 1 = 22

    seq_length = sum(all_size)

    mask = torch.zeros(seq_length, seq_length, device=device)
```

```
# get intra-scale mask
```

```
inner_window = inner_size // 2
```

```
for layer_idx in range(len(all_size)):
```

```
    start = sum(all_size[:layer_idx])
```

```
    for i in range(start, start + all_size[layer_idx]):
```

```
        left_side = max(i - inner_window, start)
```

```
        right_side = min(i + inner_window + 1, start + all_size[layer_idx])
```

```
        # mask[i, left_side:right_side] = 1
```

```
        if layer_idx == 0:
```

```
            mask[i, 0:right_side] = 1
```

```
        else:
```

```
            mask[i, left_side:right_side] = 1
```

```
# get inter-scale mask
```

```
for layer_idx in range(1, len(all_size)):
```

```
    start = sum(all_size[:layer_idx])
```

```
    for i in range(start, start + all_size[layer_idx]):
```

```
        left_side = (start - all_size[layer_idx - 1]) + (i - start) * window_size[layer_idx - 1]
```

```
        if i == (start + all_size[layer_idx] - 1):
```

```
            right_side = start
```

```
        else:
```

```
            right_side = (start - all_size[layer_idx - 1]) + (i - start + 1) * window_size[layer_idx - 1]
```

```
mask[i, left_side:right_side] = 1
```

```
mask[left_side:right_side, i] = 1
```

```
# mask = (1 - mask).bool()
```

```
mask = mask.bool()
```

```
return mask, all_size
```

```
class ConvLayer(nn.Module):
```

```
def __init__(self, c_in, window_size):
```

```
    super(ConvLayer, self).__init__()
```

```
    self.downConv = nn.Conv2d(in_channels=c_in,
```

```
                               out_channels=c_in,
```

```
                               kernel_size=[1, window_size],
```

```
                               stride=[1, window_size])
```

```
    torch.nn.init.xavier_uniform_(self.downConv.weight)
```

```
def forward(self, x):
```

```
    # (B,T,N,D)-> (B, D, N, T)
```

```
    x = x.permute(0, 3, 2, 1)
```

```
    x = self.downConv(x)
```

```
x = F.relu_(x)
```

```
return x.permute(0, 3, 2, 1) # (B, D, N, T)-->(B,T,N,D)
```

```
class Conv_Construct(nn.Module):
```

```
    """Convolution CSCM"""
```

```
    def __init__(self, d_model, window_size):
```

```
        super(Conv_Construct, self).__init__()
```

```
        if not isinstance(window_size, list):
```

```
            self.conv_layers = nn.ModuleList([
```

```
                ConvLayer(d_model, window_size),
```

```
                ConvLayer(d_model, window_size),
```

```
                ConvLayer(d_model, window_size)
```

```
            ])
```

```
        else:
```

```
            self.conv_layers = nn.ModuleList([
```

```
                ConvLayer(d_model, window_size[0]),
```

```
                ConvLayer(d_model, window_size[1]),
```

```
                ConvLayer(d_model, window_size[2])
```

```
            ])
```

```
        self.norm = nn.LayerNorm(d_model)
```

```
def forward(self, enc_input):
```

```
    # enc_input: (B, T, N, D)
```

```
    all_inputs = []
```

```
    all_inputs.append(enc_input) # 先把初始输入放进列表内
```

```
    for i in range(len(self.conv_layers)):
```

```
        enc_input = self.conv_layers[i](enc_input) # 以卷积核[2,2,2]为例: 1th conv:
```

```
(B,T,N,D)-->(B,T/2,N,D); 2th conv: (B,T/2,N,D)-->(B,T/4,N,D); 3th conv: (B,T/4,N,D)-->(B,T/8,N,D)
```

```
        all_inputs.append(enc_input)
```

```
    all_inputs = torch.cat(all_inputs, dim=1) # 在时间维度上拼接多尺度的序列: (B,T,N,D) + (B,T/2,N,D) +
```

```
(B,T/4,N,D) + (B,T/8,N,D) = (B,M,N,D); 令 M=T+T/2+T/4+T/8
```

```
    all_inputs = self.norm(all_inputs)
```

```
    return all_inputs
```

```
class MultiHeadAttention(nn.Module):
```

```
    """ Multi-Head Attention module """
```

```
    def __init__(self, n_head, d_model, d_k, dropout, normalize_before, all_size):
```

```
        super().__init__()
```

```
D = n_head * d_k
```

```
self.all_size = all_size
```

```
self.d = d_k
```

```
self.K = n_head
```

```
self.mask = True
```

```
self.FC_q = nn.Linear(D,D)
```

```
self.FC_k = nn.Linear(D,D)
```

```
self.FC_v = nn.Linear(D,D)
```

```
self.FC = nn.Linear(D,D)
```

```
def forward(self, q, k, v, mask=None):
```

```
    batch_size_ = q.shape[0]
```

```
    query = self.FC_q(q) #生成 q 矩阵: (B,M,N,D)--> (B,M,N,D)
```

```
    key = self.FC_k(k) #生成 k 矩阵: (B,M,N,D)--> (B,M,N,D)
```

```
    value = self.FC_v(v) #生成 v 矩阵: (B,M,N,D)--> (B,M,N,D)
```

```
    query = torch.cat(torch.split(query, self.d, dim=-1), dim=0) # 划分为多头: (B,M,N,D)-->(B*k,M,N,d);
```

D=k\*d

```
    key = torch.cat(torch.split(key, self.d, dim=-1), dim=0) # 划分为多头: (B,M,N,D)-->(B*k,M,N,d);
```

D=k\*d

```
    value = torch.cat(torch.split(value, self.d, dim=-1), dim=0) #划分为多头: (B,M,N,D)-->(B*k,M,N,d);
```

D=k\*d

```
    query = query.permute(0, 2, 1, 3) # 进行变换,为了便于计算: (B*k,M,N,d)-->(B*k,N,M,d)
```



```
key = key.permute(0, 2, 3, 1) # (B*k,M,N,d)-->(B*k,N,d,M)
```

```
value = value.permute(0, 2, 1, 3) # (B*k,M,N,d)-->(B*k,N,M,d)
```

```
attention = torch.matmul(query, key) # 得到注意力矩阵: (B*k,N,M,d) @ (B*k,N,d,M) = (B*k,N,M,M)
```

```
attention /= (self.d ** 0.5)
```

```
# 屏蔽掉注意力矩阵中那些没有连接的节点对
```

```
if self.mask:
```

```
num = torch.tensor(-2 ** 15 + 1)
```

```
num = num.to(torch.float32).to(device)
```

```
attention = torch.where(mask, attention, num) # 如果 mask 某元素是 false, 那么 attention 矩阵的对应位置应填入负无穷数值(即 num), 这样的话在执行 softmax 之后负无穷对应的位置应当趋近于 0
```

```
# softmax
```

```
attention = F.softmax(attention, dim=-1)
```

```
# [batch_size, num_step, num_vertex, D]
```

```
X = torch.matmul(attention, value) # 通过注意力矩阵聚合对应节点的信息: (B*k,N,M,M) @ (B*k,N,M,d) = (B*k,N,M,d)
```

```
X = X.permute(0, 2, 1, 3) # (B*k,N,M,d)-->(B*k,M,N,d)
```

```
X = torch.cat(torch.split(X, batch_size_, dim=0), dim=-1) # 在通道上拼接多头注意力的输出: (B*k,M,N,d)-->(B,M,N,k*d)=(B,M,N,D)
```

```
X = self.FC(X) # 通过映射层融合多个子空间的特征
```

```
return X[:, :self.all_size[0]] # 只选择原始序列的长度进行输出: (B,M,N,D)-->(B,T,N,D)
```

```
if __name__ == '__main__':
```

```
    # (B,T,N,D) N:序列的个数, T:时间序列的长度; 注意: 输入长度是 8,window_size=[2,2,2]; 如果输入长度是
    12,window_size=[2,2,3]; 确保除到最后一层长度为 1: 8/2/2/2=1; 12/2/2/3=1;
```

```
    X = torch.randn(1,8,1,64)
```

```
    seq_length = X.shape[1]
```

```
    # 得到金字塔的 mask 矩阵; 以输入序列长度等于 8,三层卷积核分别为[2, 2, 2]为例子: all_size=[8,4,2,1],存放每一个尺
    度对应的序列长度
```

```
    mask, all_size = get_mask(input_size=seq_length, window_size=[2, 2, 2], inner_size=3, device=device)
```

```
    # 通过卷积构造金字塔结构 (粗尺度构造模块)
```

```
    conv_layers = Conv_Construct(d_model=64, window_size=[2, 2, 2])
```

```
    # 定义多头注意力机制
```

```
    Model = MultiHeadAttention(n_head=8, d_model=64, d_k=8, dropout=0., normalize_before=False,
    all_size=all_size)
```

```
    X = conv_layers(X) #执行粗尺度构造模块(B,T,N,D)-->(B,M,N,D)
```

```
    output = Model(X, X, X, mask=mask) # 执行注意力机制: (B,M,N,D)--> (B,T,N,D)
```

```
    print(output.shape)
```

## 19、SAFM 模块

论文《Spatially-Adaptive Feature Modulation for Efficient Image Super-Resolution》

## 1、作用

这篇论文通过提出空间自适应特征调制（Spatially-Adaptive Feature Modulation, SAFM）机制，旨在解决图像超分辨率（Super-Resolution, SR）的高效设计问题。在图像超分辨率重建性能上取得了显著的成果，这些模型通常具有大型复杂的架构，不适用于低功耗设备，限于计算和存储资源。SAFM 层通过独立计算学习多尺度特征表示，并动态聚合这些特征进行空间调制，克服了这些挑战。

## 2、机制

### 1、空间自适应特征调制（SAFM）层：

SAFM 层利用多尺度特征表示独立学习，并动态进行空间调制。SAFM 着重于利用非局部特征依赖性，进一步引入卷积通道混合器（Convolutional Channel Mixer, CCM），以编码局部上下文信息并同时混合通道。

### 2、卷积通道混合器（CCM）：

为了补充局部上下文信息，提出了基于 FMBCConv 的 CCM，用于编码局部特征并混合通道，增强了模型处理特征的能力。

## 3、独特优势

### 1、高效性和灵活性：

SAFMN 模型相比于现有的高效 SR 方法小 3 倍，如 IMDN 等，同时以更少的内存使用实现了可比的性能。

### 2、动态空间调制：

通过利用多尺度特征表示进行动态空间调制，SAFMN 能够高效地聚合特征，提升重建性能，同时保持低计算和存储成本。

### 3、局部和非局部特征的有效整合：

通过 SAFM 层和 CCM 的结合，SAFMN 有效整合了局部和非局部特征信息，实现了更精准的图像超分辨率重建。

## 4、代码

```
import torch
```

```
import torch.nn as nn

import torch.nn.functional as F

# 定义 SAFM 类，继承自 nn.Module

class SAFM(nn.Module):

    def __init__(self, dim, n_levels=4):

        super().__init__()

        # n_levels 表示特征会被分割成多少个不同的尺度

        self.n_levels = n_levels

        # 每个尺度的特征通道数

        chunk_dim = dim // n_levels

        # Spatial Weighting: 针对每个尺度的特征，使用深度卷积进行空间加权

        self.mfr = nn.ModuleList([nn.Conv2d(chunk_dim, chunk_dim, 3, 1, 1, groups=chunk_dim) for i in
range(self.n_levels)])

        # Feature Aggregation: 用于聚合不同尺度处理过的特征

        self.aggr = nn.Conv2d(dim, dim, 1, 1, 0)

        # Activation: 使用 GELU 激活函数

        self.act = nn.GELU()

    def forward(self, x):
```

```
# x 的形状为(B,C,H,W)，其中 B 是批次大小，C 是通道数，H 和 W 是高和宽
```

```
h, w = x.size()[-2:]
```

```
# 将输入特征在通道维度上分割成 n_levels 个尺度
```

```
xc = x.chunk(self.n_levels, dim=1)
```

```
out = []
```

```
for i in range(self.n_levels):
```

```
    if i > 0:
```

```
        # 计算每个尺度下采样后的大小
```

```
        p_size = (h // 2**i, w // 2**i)
```

```
        # 对特征进行自适应最大池化，降低分辨率
```

```
        s = F.adaptive_max_pool2d(xc[i], p_size)
```

```
        # 对降低分辨率的特征应用深度卷积
```

```
        s = self.mfr[i](s)
```

```
        # 使用最近邻插值将特征上采样到原始大小
```

```
        s = F.interpolate(s, size=(h, w), mode='nearest')
```

```
    else:
```

```
        # 第一尺度直接应用深度卷积，不进行下采样
```

```
        s = self.mfr[i](xc[i])
```

```
    out.append(s)
```

```
# 将处理过的所有尺度的特征在通道维度上进行拼接
```

```
out = torch.cat(out, dim=1)
```

```
# 通过 1x1 卷积聚合拼接后的特征
```

```
out = self.aggr(out)
```

```
# 应用 GELU 激活函数并与原始输入相乘，实现特征调制
```

```
out = self.act(out) * x
```

```
return out
```

```
if __name__ == '__main__':
```

```
# 创建一个 SAFM 实例并对一个随机输入进行处理
```

```
x = torch.randn(1, 36, 224, 224)
```

```
Model = SAFM(dim=36)
```

```
out = Model(x)
```

```
print(out.shape)
```

## 20、EfficientAdditiveAttnetion 模块

论文《SwiftFormer: Efficient Additive Attention for Transformer-based Real-time Mobile Vision Applications》

### 1、作用

SwiftFormer 通过引入高效的加性注意力机制，解决了在移动设备上实时运行视觉应用时自注意力的高计算复杂性问题。这种机制通过线性的元素级乘法替换了传统的自注意力中的二次矩阵乘法操作，显著减少了计算复杂性，使得模型能够在资源受限的移动设备上高效运行，同时保持高准确率。

## 2、机制

### 1、高效加性注意力：

提出了一种新的注意力机制，通过线性的元素级乘法替代传统的二次矩阵乘法操作，有效减少计算复杂性。关键-值（key-value）的交互被简化为通过线性层，无需显式的 key-value 交互，从而进一步降低了计算量。

### 2、全局上下文的计算：

通过仅利用查询（query）-键（key）交互和一个线性变换来计算全局上下文，避免了昂贵的矩阵乘法操作，使得模型在所有网络阶段都能使用提出的注意力块。

## 3、独特优势

### 1、计算效率：

与以往的方法相比，SwiftFormer 的高效加性注意力显著降低了计算复杂性，实现了与移动设备上常用的 CNN 模型相媲美的速度，同时保持了较高的准确率。

### 2、全阶段适用：

不同于其他方法只在网络的后期阶段使用自注意力机制，SwiftFormer 的设计允许在网络的所有阶段使用高效加性注意力，从而在整个网络中更有效地捕获上下文信息，实现了更好的速度-准确率权衡。

### 3、模型性能：

SwiftFormer 在图像分类、目标检测、实例分割等多个任务上达到了最新的性能，特别是在移动设备上的推理速度和准确率方面，显著超越了现有的 MobileViT-v2 和 EfficientFormer 模型。

## 4、代码

```
import torch

import torch.nn as nn

import torch.nn.functional as F

import numpy as np

import math
```

```
import einops
```

```
# 定义高效加性注意力模块
```

```
class EfficientAdditiveAttnetion(nn.Module):
```

```
    """
```

```
    高效加性注意力模块，用于 SwiftFormer 中。
```

```
    输入：形状为[B, N, D]的张量
```

```
    输出：形状为[B, N, D]的张量
```

```
    """
```

```
    def __init__(self, in_dims=512, token_dim=512):
```

```
        super().__init__()
```

```
        # 初始化查询和键的线性变换
```

```
        self.to_query = nn.Linear(in_dims, token_dim)
```

```
        self.to_key = nn.Linear(in_dims, token_dim)
```

```
        # 初始化可学习的权重向量和缩放因子
```

```
        self.w_a = nn.Parameter(torch.randn(token_dim, 1))
```

```
        self.scale_factor = token_dim ** -0.5
```

```
        # 初始化后续的线性变换
```

```
        self.Proj = nn.Linear(token_dim, token_dim)
```



```
self.final = nn.Linear(token_dim, token_dim)
```

```
def forward(self, x):
```

```
    B, N, D = x.shape # B:批次大小, N:序列长度, D:特征维度
```

```
    # 生成初步的查询和键矩阵
```

```
    query = self.to_query(x)
```

```
    key = self.to_key(x)
```

```
    # 对查询和键进行标准化处理
```

```
    query = torch.nn.functional.normalize(query, dim=-1)
```

```
    key = torch.nn.functional.normalize(key, dim=-1)
```

```
    # 学习查询的注意力权重, 并进行缩放和标准化
```

```
    query_weight = query @ self.w_a
```

```
    A = query_weight * self.scale_factor
```

```
    A = torch.nn.functional.normalize(A, dim=1)
```

```
    # 通过注意力权重对查询进行加权, 以生成全局查询向量
```

```
    q = torch.sum(A * query, dim=1)
```

```
    q = q.reshape(B, 1, -1)
```

```
# 计算全局查询向量和每个键的交互，再与原始查询进行逐元素相加
```

```
out = self.Proj(q * key) + query
```

```
out = self.final(out) # 通过最终的线性层输出调制后的特征
```

```
return out
```

```
if __name__ == '__main__':
```

```
# 示例： 对一个形状为[B, N, D]的随机张量应用高效加性注意力模块
```

```
X = torch.randn(1, 50, 512)
```

```
Model = EfficientAdditiveAttnetion(in_dims=512, token_dim=512)
```

```
out = Model(X)
```

```
print(out.shape) # 输出的形状应为[B, N, D]
```

## 21、gam 模块

论文《Global Attention Mechanism: Retain Information to Enhance Channel-Spatial Interactions》

### 1、作用

这篇论文提出了全局注意力机制（Global Attention Mechanism, GAM），旨在通过保留通道和空间方面的信息来增强跨维度交互，从而提升深度神经网络的性能。GAM 通过引入 3D 排列与多层感知器（MLP）用于通道注意力，并辅以卷积空间注意力子模块，提高了图像分类任务的表现。该方法在 CIFAR-100 和 ImageNet-1K 数据集上的图像分类任务中均稳定地超越了几种最新的注意力机制，包括在 ResNet 和轻量级 MobileNet 模型上的应用。

### 2、机制

1、通道注意力子模块：

利用 3D 排列保留跨三个维度的信息，并通过两层 MLP 放大跨维度的通道-空间依赖性。这个子模块通过编码器-解码器结构，以一个缩减比例  $r$ （与 BAM 相同）来实现。

## 2、空间注意力子模块：

为了聚焦空间信息，使用了两个卷积层进行空间信息的融合。同时，为了进一步保留特征图，移除了池化操作。此外，为了避免参数数量显著增加，当应用于 ResNet50 时，采用了分组卷积与通道混洗。

## 3、独特优势

### 1、效率与灵活性：

GAM 展示了与现有的高效 SR 方法相比，如 IMDN，其模型大小小了 3 倍，同时实现了可比的性能，展现了在内存使用上的高效性。

### 2、动态空间调制：

通过利用独立学习的多尺度特征表示并动态地进行空间调制，GAM 能够高效地聚合特征，提升重建性能，同时保持低计算和存储成本。

### 3、有效整合局部和非局部特征：

GAM 通过其层和 CCM 的结合，有效地整合了局部和非局部特征信息，实现了更精确的图像超分辨率重建。

## 4、代码

```
import torch.nn as nn

import torch

class GAM_Attention(nn.Module):

    def __init__(self, in_channels, rate=4):

        super(GAM_Attention, self).__init__()

        # 通道注意力子模块
```

```
self.channel_attention = nn.Sequential(
```

```
    # 降维，减少参数数量和计算复杂度
```

```
    nn.Linear(in_channels, int(in_channels / rate)),
```

```
    nn.ReLU(inplace=True), # 非线性激活
```

```
    # 升维，恢复到原始通道数
```

```
    nn.Linear(int(in_channels / rate), in_channels)
```

```
)
```

```
    # 空间注意力子模块
```

```
self.spatial_attention = nn.Sequential(
```

```
    # 使用 7x7 卷积核进行空间特征的降维处理
```

```
    nn.Conv2d(in_channels, int(in_channels / rate), kernel_size=7, padding=3),
```

```
    nn.BatchNorm2d(int(in_channels / rate)), # 批归一化，加速收敛，提升稳定性
```

```
    nn.ReLU(inplace=True), # 非线性激活
```

```
    # 使用 7x7 卷积核进行空间特征的升维处理
```

```
    nn.Conv2d(int(in_channels / rate), in_channels, kernel_size=7, padding=3),
```

```
    nn.BatchNorm2d(in_channels) # 批归一化
```

```
)
```

```
def forward(self, x):
```

```
    b, c, h, w = x.shape # 输入张量的维度信息
```

```
    # 调整张量形状以适配通道注意力处理
```

```
x_permute = x.permute(0, 2, 3, 1).view(b, -1, c)
```

```
# 应用通道注意力，并恢复原始张量形状
```

```
x_att_permute = self.channel_attention(x_permute).view(b, h, w, c)
```

```
# 生成通道注意力图
```

```
x_channel_att = x_att_permute.permute(0, 3, 1, 2).sigmoid()
```

```
# 应用通道注意力图进行特征加权
```

```
x = x * x_channel_att
```

```
# 生成空间注意力图并应用进行特征加权
```

```
x_spatial_att = self.spatial_attention(x).sigmoid()
```

```
out = x * x_spatial_att
```

```
return out
```

```
# 示例代码：使用 GAM_Attention 对一个随机初始化的张量进行处理
```

```
if __name__ == '__main__':
```

```
    x = torch.randn(1, 64, 20, 20) # 随机生成输入张量
```

```
    b, c, h, w = x.shape # 获取输入张量的维度信息
```

```
    net = GAM_Attention(in_channels=c) # 实例化 GAM_Attention 模块
```

```
    y = net(x) # 通过 GAM_Attention 模块处理输入张量
```

```
    print(y.shape) # 打印输出张量的维度信息
```

# 22、PSA 模块

论文《EPSANet: An Efficient Pyramid Squeeze Attention Block on Convolutional Neural Network》

## 1、作用

EPSANet 通过引入高效的金字塔挤压注意力（Pyramid Squeeze Attention, PSA）模块，显著提升了深度卷积神经网络在图像分类、对象检测和实例分割等计算机视觉任务中的性能。通过在 ResNet 的瓶颈块中替换 3x3 卷积为 PSA 模块，EPSANet 能够在不增加显著计算负担的情况下，提供更丰富的多尺度特征表示和更有效的通道空间交互。

## 2、机制

### 1、金字塔挤压注意力模块（PSA）：

通过利用多尺度金字塔卷积结构整合输入特征图的信息，并通过挤压输入张量的通道维度来有效地从每个通道的特征图中提取不同尺度的空间信息。此外，通过提取多尺度特征图的通道注意力权重并使用 Softmax 操作重新校准对应通道的注意力权重，建立了长程通道依赖性。

### 2、高效金字塔挤压注意力（EPSA）块：

将 PSA 模块替换到 ResNet 的瓶颈块中，获取了名为 EPSA 的新型表示块。EPSA 块易于作为即插即用组件添加到现有的骨干网络中，并且能够在模型性能上获得显著提升。

### 3、EPSANet 架构：

通过堆叠 ResNet 风格的 EPSA 块，开发了简单且高效的 EPSANet 骨干架构。EPSANet 通过提出的 PSA 模块，为各种计算机视觉任务提供了更强的多尺度表示能力，并能够适应性地重新校准跨维度的通道注意力权重。

## 3、独特优势

### 1、性能提升：

相比于 SENet-50，EPSANet 在 ImageNet 数据集上的 Top-1 准确率提高了 1.93%，在 MS COCO 数据集上，对象检测的 box AP 提高了 2.7 个百分点，实例分割的 mask AP 提高了 1.7 个百分点。

### 2、计算效率：

EPSANet 模型尺寸小于高效 SR 方法，如 IMDN，同时在内存使用上更为高效。

### 3、动态空间调制：

EPSANet 通过动态空间调制有效聚合了特征，提升了重建性能，同时保持了低计算和存储成本。

### 4、有效整合局部和非局部特征：

EPSANet 通过 PSA 层和卷积通道混合器（CCM）的结合，有效整合了局部和非局部特征信息，实现了更精确的图像超分辨率重建。

## 4、代码

```
import torch

import torch.nn as nn

import torch.nn.functional as F


class PSA(nn.Module):

    def __init__(self, channel=512, reduction=4, S=4):

        super(PSA, self).__init__()

        self.S = S # 尺度的数量


        # 定义不同尺度的卷积层

        self.convs = nn.ModuleList([

            nn.Conv2d(channel // S, channel // S, kernel_size=2 * (i + 1) + 1, padding=i + 1)

            for i in range(S)

        ])
```

```
# 定义每个尺度对应的 SE 模块
```

```
self.se_blocks = nn.ModuleList([
```

```
    nn.Sequential(
```

```
        nn.AdaptiveAvgPool2d(1),
```

```
        nn.Conv2d(channel // S, channel // (S * reduction), kernel_size=1, bias=False),
```

```
        nn.ReLU(inplace=True),
```

```
        nn.Conv2d(channel // (S * reduction), channel // S, kernel_size=1, bias=False),
```

```
        nn.Sigmoid()
```

```
    ) for i in range(S)
```

```
    ])
```

```
self.softmax = nn.Softmax(dim=1) # 用于归一化注意力权重
```

```
def forward(self, x):
```

```
    b, c, h, w = x.size()
```

```
# 将输入在通道维度上按尺度分割
```

```
SPC_out = x.view(b, self.S, c // self.S, h, w)
```

```
# 应用不同尺度的卷积操作
```

```
conv_out = []
```

```
for idx, conv in enumerate(self.convs):
```



```
conv_out.append(conv(SPC_out[:, idx, :, :, :]))
```

```
SPC_out = torch.stack(conv_out, dim=1)
```

```
# 应用 SE 模块进行通道注意力加权
```

```
se_out = [se(SPC_out[:, idx, :, :, :]) for idx, se in enumerate(self.se_blocks)]
```

```
SE_out = torch.stack(se_out, dim=1)
```

```
SE_out = SE_out.expand(-1, -1, -1, h, w) # 扩展以匹配 SPC_out 的尺寸
```

```
# 应用 Softmax 归一化注意力权重
```

```
softmax_out = self.softmax(SE_out)
```

```
# 应用注意力权重并合并多尺度特征
```

```
PSA_out = SPC_out * softmax_out
```

```
PSA_out = torch.sum(PSA_out, dim=1) # 沿尺度维度合并特征
```

```
return PSA_out
```

```
if __name__ == '__main__':
```

```
input = torch.randn(3, 512, 64, 64)
```

```
psa = PSA(channel=512, reduction=4, S=4)
```

```
output = psa(input)
```

```
print(output.shape)
```

# 23、HaLoAttention 模块

论文《Scaling Local Self-Attention for Parameter Efficient Visual Backbones》

## 1、作用

HaloNet 通过引入 Haloing 机制和高效的注意力实现，在图像识别任务中达到了最先进的准确性。这些模型通过局部自注意力机制，有效地捕获像素间的全局交互，同时通过分块和 Haloing 策略，显著提高了处理速度和内存效率。

## 2、机制

### 1、Haloing 策略：

为了克服传统自注意力的计算和内存限制，HaloNet 采用了 Haloing 策略，将图像分割成多个块，并为每个块扩展一定的 Halo 区域，仅在这些区域内计算自注意力。这种方法减少了计算量，同时保持了较大的感受野。

### 2、多尺度特征层次：

HaloNet 构建了多尺度特征层次结构，通过分层采样和跨尺度的信息流，有效捕获不同尺度的图像特征，增强了模型对图像中对象大小变化的适应性。

### 3、高效的自注意力实现：

通过改进的自注意力算法，包括非中心化的局部注意力和分层自注意力下采样操作，HaloNet 在保持高准确性的同时，提高了训练和推理速度。

## 3、独特优势

### 1、参数效率：

HaloNet 通过局部自注意力机制和 Haloing 策略，大幅度减少了所需的计算量和内存需求，实现了与当前最佳卷积模型相当甚至更好的性能，但使用更少的参数。

### 2、适应多尺度：

多尺度特征层次结构使得 HaloNet 能够有效处理不同尺度的对象，提高了对复杂视觉任务的适应性和准确性。

### 3、提升速度和效率：

通过优化的自注意力实现，HaloNet 在不牺牲准确性的前提下，实现了比现有技术更快的训练和推理速度，使其更适合实际应用。

### 4、代码

```
import torch

from torch import nn, einsum

import torch.nn.functional as F

from einops import rearrange, repeat

# relative positional embedding

def to(x):

    return {'device': x.device, 'dtype': x.dtype}

def pair(x):

    return (x, x) if not isinstance(x, tuple) else x

def expand_dim(t, dim, k):
```

```

t = t.unsqueeze(dim=dim)

expand_shape = [-1] * len(t.shape)

expand_shape[dim] = k

return t.expand(*expand_shape)

```

```

def rel_to_abs(x):

    b, l, m = x.shape

    r = (m + 1) // 2

    col_pad = torch.zeros((b, l, 1), **to(x))

    x = torch.cat((x, col_pad), dim=2)

    flat_x = rearrange(x, 'b l c -> b (l c)')

    flat_pad = torch.zeros((b, m - 1), **to(x))

    flat_x_padded = torch.cat((flat_x, flat_pad), dim=1)

    final_x = flat_x_padded.reshape(b, l + 1, m)

    final_x = final_x[:, :l, -r:]

    return final_x

```

```

def relative_logits_1d(q, rel_k):

    b, h, w, _ = q.shape

```

```

r = (rel_k.shape[0] + 1) // 2

logits = einsum('b x y d, r d -> b x y r', q, rel_k)

logits = rearrange(logits, 'b x y r -> (b x) y r')

logits = rel_to_abs(logits)


logits = logits.reshape(b, h, w, r)

logits = expand_dim(logits, dim=2, k=r)

return logits

```

```

class RelPosEmb(nn.Module):

    def __init__(
        self,
        block_size,
        rel_size,
        dim_head
    ):

        super().__init__()

        height = width = rel_size

        scale = dim_head ** -0.5

```

```
self.block_size = block_size
```

```
self.rel_height = nn.Parameter(torch.randn(height * 2 - 1, dim_head) * scale)
```

```
self.rel_width = nn.Parameter(torch.randn(width * 2 - 1, dim_head) * scale)
```

```
def forward(self, q):
```

```
    block = self.block_size
```

```
    q = rearrange(q, 'b (x y) c -> b x y c', x=block)
```

```
    rel_logits_w = relative_logits_1d(q, self.rel_width)
```

```
    rel_logits_w = rearrange(rel_logits_w, 'b x i y j-> b (x y) (i j)')
```

```
    q = rearrange(q, 'b x y d -> b y x d')
```

```
    rel_logits_h = relative_logits_1d(q, self.rel_height)
```

```
    rel_logits_h = rearrange(rel_logits_h, 'b x i y j -> b (y x) (j i)')
```

```
    return rel_logits_w + rel_logits_h
```

```
# classes
```

```
class HaloAttention(nn.Module):
```

```
    def __init__(
```

```
        self,
```

```
        *,

        dim,

        block_size,

        halo_size,

        dim_head=64,

        heads=8

    ):

        super().__init__()

        assert halo_size > 0, 'halo size must be greater than 0'

        self.dim = dim

        self.heads = heads

        self.scale = dim_head ** -0.5

        self.block_size = block_size

        self.halo_size = halo_size

        inner_dim = dim_head * heads

        self.rel_pos_emb = RelPosEmb(

            block_size=block_size,

            rel_size=block_size + (halo_size * 2),
```

```

        dim_head=dim_head

    )

    self.to_q = nn.Linear(dim, inner_dim, bias=False)

    self.to_kv = nn.Linear(dim, inner_dim * 2, bias=False)

    self.to_out = nn.Linear(inner_dim, dim)

def forward(self, x):

    b, c, h, w, block, halo, heads, device = *x.shape, self.block_size, self.halo_size, self.heads, x.device

    assert h % block == 0 and w % block == 0, 'fmap dimensions must be divisible by the block size'

    assert c == self.dim, f'channels for input ({c}) does not equal to the correct dimension ({self.dim})'

    # get block neighborhoods, and prepare a halo-ed version (blocks with padding) for deriving key values

    q_inp = rearrange(x, 'b c (h p1) (w p2) -> (b h w) (p1 p2) c', p1=block, p2=block)

    kv_inp = F.unfold(x, kernel_size=block + halo * 2, stride=block, padding=halo)

    kv_inp = rearrange(kv_inp, 'b (c j) i -> (b i) j c', c=c)

    # derive queries, keys, values

    q = self.to_q(q_inp)

```



```

k, v = self.to_kv(kv_inp).chunk(2, dim=-1)

# split heads

q, k, v = map(lambda t: rearrange(t, 'b n (h d) -> (b h) n d', h=heads), (q, k, v))

# scale

q *= self.scale

# attention

sim = einsum('b i d, b j d -> b i j', q, k)

# add relative positional bias

sim += self.rel_pos_emb(q)

# mask out padding (in the paper, they claim to not need masks, but what about padding?)

mask = torch.ones(1, 1, h, w, device=device)

mask = F.unfold(mask, kernel_size=block + (halo * 2), stride=block, padding=halo)

```

```
mask = repeat(mask, '() j i -> (b i h) () j', b=b, h=heads)

mask = mask.bool()

max_neg_value = -torch.finfo(sim.dtype).max

sim.masked_fill_(mask, max_neg_value)

# attention

attn = sim.softmax(dim=-1)

# aggregate

out = einsum('b i j, b j d -> b i d', attn, v)

# merge and combine heads

out = rearrange(out, '(b h) n d -> b n (h d)', h=heads)

out = self.to_out(out)

# merge blocks back to original feature map

out = rearrange(out, '(b h w) (p1 p2) c -> b c (h p1) (w p2)', b=b, h=(h // block), w=(w // block), p1=block,
```

```

        p2=block)

    return out

# 输入 N C H W, 输出 N C H W

if __name__ == '__main__':

    block = HaloAttention(dim=512,

                           block_size=2,

                           halo_size=1, ).cuda()

    input = torch.rand(1, 512, 64, 64).cuda()

    output = block(input)

    print(output.shape)

```

## 24、ViP 模块

论文《VISION PERMUTATOR: A PERMUTABLE MLP-LIKE ARCHITECTURE FOR VISUAL RECOGNITION》

### 1、作用

论文提出的 Vision Permutator 是一种简单、数据高效的类 MLP（多层感知机）架构，用于视觉识别。不同于其他 MLP 类模型，它通过线性投影分别对特征表示在高度和宽度维度进行编码，能够保留 2D 特征表示中的位置信息，有效捕获沿一个空间方向的长距离依赖关系，同时保留另一方向上的精确位置信息。

### 2、机制

### 1、视觉置换器：

Vision Permutator 采用与视觉变换器类似的令牌化操作，将输入图像均匀划分为小块，并通过线性投影将它们映射为令牌嵌入。随后，这些令牌嵌入被送入一系列 Permutator 块中进行特征编码。

### 2、Permute-MLP：

Permutator 块包含一个用于空间信息编码的 Permute-MLP 和一个用于通道信息混合的 Channel-MLP。Permute-MLP 通过独立处理令牌表示沿高度和宽度的维度，生成具有特定方向信息的令牌，这对于视觉识别至关重要。

### 3、加权 Permute-MLP：

在简单的 Permute-MLP 基础上，引入加权 Permute-MLP 来重新校准不同分支的重要性，进一步提高模型性能。

## 3、独特优势

### 1、空间信息编码：

Vision Permutator 通过在高度和宽度维度上分别对特征进行编码，相比于其他将两个空间维度混合为一个进行处理的 MLP 类模型，能够更有效地保留空间位置信息，从而提高模型对图像中对象的识别能力。

### 2、性能提升：

实验表明，即使在不使用额外大规模训练数据的情况下，Vision Permutator 也能达到 81.5% 的 ImageNet 顶级-1 准确率，并且仅使用 25M 可学习参数，这比大多数同等大小模型的 CNNs 和视觉变换器都要好。

### 3、模型高效：

Vision Permutator 的结构简单、数据高效，在确保高准确性的同时提高了训练和推理速度，展现了 MLP 类模型在视觉识别任务中的潜力。

## 4、代码

```
import torch

from torch import nn
```

```
class MLP(nn.Module):

    def __init__(self, in_features, hidden_features, out_features, act_layer=nn.GELU, drop=0.1):

        super().__init__()

        # 第一层全连接层

        self.fc1 = nn.Linear(in_features, hidden_features)

        # 激活函数

        self.act = act_layer()

        # 第二层全连接层

        self.fc2 = nn.Linear(hidden_features, out_features)

        # Dropout 层

        self.drop = nn.Dropout(drop)

    def forward(self, x):

        # 顺序通过第一层全连接层、激活函数、Dropout、第二层全连接层、Dropout

        return self.drop(self.fc2(self.drop(self.act(self.fc1(x)))))

class WeightedPermuteMLP(nn.Module):

    def __init__(self, dim, seg_dim=8, qkv_bias=False, proj_drop=0.):

        super().__init__()

        # 分段维度，用于在特定维度上分段处理特征

        self.seg_dim = seg_dim
```

```
# 定义对通道 C、高度 H、宽度 W 的 MLP 处理层
```

```
self.mlp_c = nn.Linear(dim, dim, bias=qkv_bias)
```

```
self.mlp_h = nn.Linear(dim, dim, bias=qkv_bias)
```

```
self.mlp_w = nn.Linear(dim, dim, bias=qkv_bias)
```

```
# 重置权重的 MLP 层
```

```
self.reweighting = MLP(dim, dim // 4, dim * 3)
```

```
# 最终投影层
```

```
self.proj = nn.Linear(dim, dim)
```

```
self.proj_drop = nn.Dropout(proj_drop)
```

```
def forward(self, x):
```

```
    B, H, W, C = x.shape
```

```
# 通道维度的处理
```

```
c_embed = self.mlp_c(x)
```

```
# 高度维度的处理
```

```
S = C // self.seg_dim
```

```
h_embed = x.reshape(B, H, W, self.seg_dim, S).permute(0, 3, 2, 1, 4).reshape(B, self.seg_dim, W, H *
```

```
S)
```

```

h_embed = self.mlp_h(h_embed).reshape(B, self.seg_dim, W, H, S).permute(0, 3, 2, 1, 4).reshape(B, H,
W, C)

# 宽度维度的处理

w_embed = x.reshape(B, H, W, self.seg_dim, S).permute(0, 3, 1, 2, 4).reshape(B, self.seg_dim, H, W *
S)

w_embed = self.mlp_w(w_embed).reshape(B, self.seg_dim, H, W, S).permute(0, 2, 3, 1, 4).reshape(B, H,
W, C)

# 计算三个维度的权重并应用 softmax 进行归一化

weight = (c_embed + h_embed + w_embed).permute(0, 3, 1, 2).flatten(2).mean(2)

weight = self.reweighting(weight).reshape(B, C, 3).permute(2, 0,
1).softmax(0).unsqueeze(2).unsqueeze(2)

# 加权融合处理后的特征

x = c_embed * weight[0] + w_embed * weight[1] + h_embed * weight[2]

# 应用投影层和 Dropout

x = self.proj_drop(self.proj(x))

return x

if __name__ == '__main__':

```

```
input = torch.randn(64, 8, 8, 512) # 模拟输入数据

seg_dim = 8 # 定义分段维度

vip = WeightedPermuteMLP(512, seg_dim) # 初始化模型

out = vip(input) # 前向传播

print(out.shape)
```

## 25、SKAttention 模块

论文《Selective Kernel Networks》

### 1、作用

该论文介绍了选择性核网络（SKNets），这是一种在卷积神经网络（CNN）中的动态选择机制，允许每个神经元根据输入自适应地调整其感受野大小。这种方法受到视觉皮层神经元对不同刺激响应时感受野大小变化的启发，在 CNN 设计中不常利用此特性。

### 2、机制

SKNets 利用了一个称为选择性核（SK）单元的构建模块，该模块包含具有不同核大小的多个分支。这些分支通过一个 softmax 注意力机制融合，由这些分支中的信息引导。这个融合过程使得神经元能够根据输入自适应地调整其有效感受野大小。

### 3、独特优势

1、自适应感受野：

SKNets 中的神经元可以基于输入动态调整其感受野大小，模仿生物神经元的适应能力。这允许在不同尺度上更有效地处理视觉信息。

2、计算效率：



尽管为了适应性而纳入了多种核大小，SKNets 仍然保持了较低的模型复杂度，与现有最先进的架构相比。通过仔细的设计选择，如使用高效的分组/深度卷积和注意力机制中的缩减比率来控制参数数量，实现了这种效率。

### 3、性能提升：

在 ImageNet 和 CIFAR 等基准测试上的实验结果显示，SKNets 在具有相似或更低模型复杂度的情况下，超过了其他最先进的架构。适应性调整感受野的能力可能有助于更有效地捕捉不同尺度的目标对象，提高识别性能。

## 4、代码

```
import numpy as np

import torch

from torch import nn

from torch.nn import init

from collections import OrderedDict


class SKAttention(nn.Module):

    def __init__(self, channel=512, kernels=[1, 3, 5, 7], reduction=16, group=1, L=32):

        super().__init__()

        # 计算维度压缩后的向量长度

        self.d = max(L, channel // reduction)

        # 不同尺寸的卷积核组成的卷积层列表

        self.convs = nn.ModuleList([])

        for k in kernels:

            self.convs.append(

                nn.Sequential(OrderedDict([
```

```
        ('conv', nn.Conv2d(channel, channel, kernel_size=k, padding=k // 2, groups=group)),
```

```
        ('bn', nn.BatchNorm2d(channel)),
```

```
        ('relu', nn.ReLU()))
```

```
    ]))
```

```
)
```

```
    # 通道数压缩的全连接层
```

```
    self.fc = nn.Linear(channel, self.d)
```

```
    # 为每个卷积核尺寸对应的特征图计算注意力权重的全连接层列表
```

```
    self.fcs = nn.ModuleList([])
```

```
    for i in range(len(kernels)):
```

```
        self.fcs.append(nn.Linear(self.d, channel))
```

```
    # 注意力权重的 Softmax 层
```

```
    self.softmax = nn.Softmax(dim=0)
```

```
    def forward(self, x):
```

```
        bs, c, _, _ = x.size()
```

```
        conv_outs = []
```

```
        # 通过不同尺寸的卷积核处理输入
```

```
        for conv in self.convs:
```

```
            conv_outs.append(conv(x))
```

```
        feats = torch.stack(conv_outs, 0) # k,bs,channel,h,w
```

```
# 将所有卷积核的输出求和得到融合特征图 U
```

```
U = sum(conv_outs) # bs,c,h,w
```

```
# 对融合特征图 U 进行全局平均池化，并通过全连接层降维得到 Z
```

```
S = U.mean(-1).mean(-1) # bs,c
```

```
Z = self.fc(S) # bs,d
```

```
# 计算每个卷积核对应的注意力权重
```

```
weights = []
```

```
for fc in self.fcs:
```

```
    weight = fc(Z)
```

```
    weights.append(weight.view(bs, c, 1, 1)) # bs,channel
```

```
attention_weights = torch.stack(weights, 0) # k,bs,channel,1,1
```

```
attention_weights = self.softmax(attention_weights) # k,bs,channel,1,1
```

```
# 将注意力权重应用到对应的特征图上，并对所有特征图进行加权求和得到最终的输出 V
```

```
V = (attention_weights * feats).sum(0)
```

```
return V
```

```
# 示例用法
```

```
if __name__ == '__main__':
```

```
    input = torch.randn(50, 512, 7, 7)
```

```
sk = SKAttention(channel=512, reduction=8)
```

```
output = sk(input)
```

```
print(output.shape) # 输出经过 SK 注意力处理后的特征图形状
```

## 26、UFO 模块

论文《UFO-ViT: High Performance Linear Vision Transformer without Softmax》

### 1、作用

UFO-ViT 旨在解决传统 Transformer 在视觉任务中所面临的主要挑战之一：SA 机制的计算资源需求随输入尺寸的平方增长，这使得处理高分辨率输入变得不切实际。UFO-ViT 通过提出一种新的 SA 机制，消除了非线性操作，实现了对计算复杂度的线性控制，同时保持了高性能。

### 2、机制

#### 1、自注意力机制的简化：

UFO-ViT 通过消除 softmax 非线性，简化了自注意力机制。它采用简单的 L2 范数替代 softmax 函数，利用矩阵乘法的结合律先计算键和值的乘积，然后再与查询相乘。

#### 2、跨标准化（XNorm）：

UFO-ViT 引入了一种新的规范化方法——跨标准化（XNorm），用于替换 softmax。这种方法将键和值的自注意力乘积直接相乘，并通过线性核方法生成聚类，以线性复杂度处理自注意力。

### 3、独特优势

#### 1、线性复杂度：

与传统的具有  $O(N^2)$  复杂度的 SA 机制不同，UFO-ViT 的 SA 机制具有线性复杂度，使其能够有效处理高分辨率输入。

#### 2、适用于通用目的：

UFO-ViT 在图像分类和密集预测任务中的表现证明了其作为通用模型的潜力。尽管复杂度线性，但 UFO-ViT 模型在较低容量和 FLOPS 下仍然超越了大多数现有的基于 Transformer 的模型。

### 3、推理速度快，GPU 内存需求低：

UFO-ViT 模型具有更快的推理速度，并且在各种分辨率下所需的 GPU 内存较少。特别是在处理高分辨率输入时，所需的计算资源并没有显著增加。

## 4、代码

```
import numpy as np

import torch

from torch import nn

from torch.functional import norm

from torch.nn import init


# 定义 XNorm 函数，对输入 x 进行规范化

def XNorm(x, gamma):

    norm_tensor = torch.norm(x, 2, -1, True)

    return x * gamma / norm_tensor


# UFOAttention 类继承自 nn.Module

class UFOAttention(nn.Module):

    ...

    实现一个改进的自注意力机制，具有线性复杂度。

    ...
```

```
# 初始化函数
```

```
def __init__(self, d_model, d_k, d_v, h, dropout=.1):
```

```
    '''
```

```
        :param d_model: 模型的维度
```

```
        :param d_k: 查询和键的维度
```

```
        :param d_v: 值的维度
```

```
        :param h: 注意力头数
```

```
    '''
```

```
    super(UFOAttention, self).__init__()
```

```
    # 初始化四个线性层: 为查询、键、值和输出转换使用
```

```
    self.fc_q = nn.Linear(d_model, h * d_k)
```

```
    self.fc_k = nn.Linear(d_model, h * d_k)
```

```
    self.fc_v = nn.Linear(d_model, h * d_v)
```

```
    self.fc_o = nn.Linear(h * d_v, d_model)
```

```
    self.dropout = nn.Dropout(dropout)
```

```
    # gamma 参数用于规范化
```

```
    self.gamma = nn.Parameter(torch.randn((1, h, 1, 1)))
```

```
    self.d_model = d_model
```

```
    self.d_k = d_k
```

```
    self.d_v = d_v
```

```
self.h = h
```

```
self.init_weights()
```

```
# 权重初始化
```

```
def init_weights(self):
```

```
    for m in self.modules():
```

```
        if isinstance(m, nn.Conv2d):
```

```
            init.kaiming_normal_(m.weight, mode='fan_out')
```

```
            if m.bias is not None:
```

```
                init.constant_(m.bias, 0)
```

```
            elif isinstance(m, nn.BatchNorm2d):
```

```
                init.constant_(m.weight, 1)
```

```
                init.constant_(m.bias, 0)
```

```
            elif isinstance(m, nn.Linear):
```

```
                init.normal_(m.weight, std=0.001)
```

```
                if m.bias is not None:
```

```
                    init.constant_(m.bias, 0)
```

```
# 前向传播
```

```
def forward(self, queries, keys, values):
```

```
    b_s, nq = queries.shape[:2]
```

```
nk = keys.shape[1]
```

```
# 通过线性层将查询、键、值映射到新的空间
```

```
q = self.fc_q(queries).view(b_s, nq, self.h, self.d_k).permute(0, 2, 1, 3)
```

```
k = self.fc_k(keys).view(b_s, nk, self.h, self.d_k).permute(0, 2, 3, 1)
```

```
v = self.fc_v(values).view(b_s, nk, self.h, self.d_v).permute(0, 2, 1, 3)
```

```
# 计算键和值的乘积，然后对结果进行规范化
```

```
kv = torch.matmul(k, v) # bs,h,c,c
```

```
kv_norm = XNorm(kv, self.gamma) # bs,h,c,c
```

```
q_norm = XNorm(q, self.gamma) # bs,h,n,c
```

```
out = torch.matmul(q_norm, kv_norm).permute(0, 2, 1, 3).contiguous().view(b_s, nq, self.h * self.d_v)
```

```
out = self.fc_o(out) # (b_s, nq, d_model)
```

```
return out
```

```
if __name__ == '__main__':
```

```
# 示例用法
```

```
block = UFOAttention(d_model=512, d_k=512, d_v=512, h=8).cuda()
```

```
input = torch.rand(64, 64, 512).cuda()
```

```
output = block(input, input, input)
```

```
print(output.shape)
```



# 27、ASPP 模块

论文《Rethinking Atrous Convolution for Semantic Image Segmentation》

## 1、作用

DeepLabv3 是一种先进的语义图像分割系统，它通过使用空洞卷积捕获多尺度上下文来显著提升性能，无需依赖 DenseCRF 后处理。

## 2、机制

DeepLabv3 的核心机制围绕空洞（扩张）卷积展开。这种技术允许模型控制滤波器的视野，使其能够在多个尺度上捕获空间上下文。DeepLabv3 在串联和并联架构中使用空洞卷积来提取密集的特征图，并有效地整合多尺度信息。文章还介绍了 Atrous Spatial Pyramid Pooling (ASPP) 模块，该模块通过在多个尺度上探索卷积特征并结合图像级特征，用于编码全局上下文。

## 3、独特优势

### 1、多尺度上下文捕获：

通过在不同配置中使用空洞卷积，DeepLabv3 能够从多个尺度捕获上下文信息，这对于准确分割不同大小的对象至关重要。

### 2、高效密集特征提取：

空洞卷积使得模型能够在不需要额外参数或计算资源的情况下提取密集特征图，提高了部署效率。

### 3、性能提升：

ASPP 与图像级特征的结合显著提高了模型性能，使其在 PASCAL VOC 2012 等基准数据集上与其他最先进方法竞争。

### 4、灵活性和泛化能力：

DeepLabv3 的框架是通用的，可以应用于任何网络架构，为适应不同的分割任务提供了灵活性。

## 4、代码

```
from torch import nn

import torch

import torch.nn.functional as F


# 定义一个包含空洞卷积、批量归一化和 ReLU 激活函数的子模块

class ASPPConv(nn.Sequential):

    def __init__(self, in_channels, out_channels, dilation):

        modules = [

            # 空洞卷积，通过调整 dilation 参数来捕获不同尺度的信息

            nn.Conv2d(in_channels, out_channels, 3, padding=dilation, dilation=dilation, bias=False),

            nn.BatchNorm2d(out_channels), # 批量归一化

            nn.ReLU() # ReLU 激活函数

        ]

        super(ASPPConv, self).__init__(*modules)


# 定义一个全局平均池化后接卷积、批量归一化和 ReLU 的子模块

class ASPPPooling(nn.Sequential):

    def __init__(self, in_channels, out_channels):

        super(ASPPPooling, self).__init__(
```

```
nn.AdaptiveAvgPool2d(1), # 全局平均池化
```

```
nn.Conv2d(in_channels, out_channels, 1, bias=False), # 1x1 卷积
```

```
nn.BatchNorm2d(out_channels), # 批量归一化
```

```
nn.ReLU()) # ReLU 激活函数
```

```
def forward(self, x):
```

```
    size = x.shape[-2:] # 保存输入特征图的空间维度
```

```
    x = super(ASPPPooling, self).forward(x)
```

```
    # 通过双线性插值将特征图大小调整回原始输入大小
```

```
    return F.interpolate(x, size=size, mode='bilinear', align_corners=False)
```

```
# ASPP 模块主体，结合不同膨胀率的空洞卷积和全局平均池化
```

```
class ASPP(nn.Module):
```

```
    def __init__(self, in_channels, atrous_rates):
```

```
        super(ASPP, self).__init__()
```

```
        out_channels = 256 # 输出通道数
```

```
        modules = []
```

```
        modules.append(nn.Sequential(
```

```
            nn.Conv2d(in_channels, out_channels, 1, bias=False), # 1x1 卷积用于降维
```

```
            nn.BatchNorm2d(out_channels),
```

```
            nn.ReLU()))
```

```
# 根据不同的膨胀率添加空洞卷积模块
```

```
for rate in atrous_rates:
```

```
    modules.append(ASPPConv(in_channels, out_channels, rate))
```

```
# 添加全局平均池化模块
```

```
modules.append(ASPPPooling(in_channels, out_channels))
```

```
self.convs = nn.ModuleList(modules)
```

```
# 将所有模块的输出融合后的投影层
```

```
self.project = nn.Sequential(
```

```
    nn.Conv2d(5 * out_channels, out_channels, 1, bias=False), # 融合特征后降维
```

```
    nn.BatchNorm2d(out_channels),
```

```
    nn.ReLU(),
```

```
    nn.Dropout(0.5)) # 防止过拟合的 Dropout 层
```

```
def forward(self, x):
```

```
    res = []
```

```
# 对每个模块的输出进行收集
```

```
for conv in self.convs:
```

```
    res.append(conv(x))
```

```
# 将收集到的特征在通道维度上拼接
```

```
res = torch.cat(res, dim=1)
```

```
# 对拼接后的特征进行处理
```

```
return self.project(res)
```

```
# 示例使用 ASPP 模块
```

```
aspp = ASPP(256, [6, 12, 18])
```

```
x = torch.rand(2, 256, 13, 13)
```

```
print(aspp(x).shape) # 输出处理后的特征图维度
```

## 28、ShuffleAttention 模块

论文《SA-NET: SHUFFLE ATTENTION FOR DEEP CONVOLUTIONAL NEURAL NETWORKS》

### 1、作用

SA 模块主要用于增强深度卷积网络在处理图像分类、对象检测和实例分割等任务时的性能。它通过在神经网络中引入注意力机制，使网络能够更加关注于图像中的重要特征，同时抑制不相关的信息。

### 2、机制

#### 1、特征分组：

SA 模块首先将输入特征图沿通道维度分成多个子特征组，这样每个子特征组可以并行处理。

#### 2、混合注意力：

对每个子特征组，SA 模块使用一个 Shuffle 单元来同时构建通道注意力和空间注意力。这通过在所有位置上设计一个注意力掩码来实现，该掩码能够压制可能的噪声并突出显示正确的语义特征区域。

### 3、子特征聚合：

处理完毕后，所有子特征重新聚合，然后采用“通道混洗”操作以实现不同子特征间的信息交流。

## 3、独特优势

### 1、效率与效果的平衡：

SA 模块有效地融合了两种类型的注意力机制，不仅保持了模型的轻量化，还显著提高了模型的性能。

### 2、并行处理能力：

通过对输入特征图进行分组并行处理，SA 模块能够有效减少计算资源的消耗，同时加速信息的处理速度。

### 3、灵活性：

SA 模块可以轻松集成到现有的 CNN 架构中，为各种视觉任务提供了一种简单而有效的注意力增强策略。

### 4、广泛的适用性：

通过在多个标准数据集上的实验验证，SA 模块在图像分类、对象检测和实例分割等任务上均取得了优于当前最先进方法的性能，显示了其优越的泛化能力。

## 4、代码

```
import numpy as np

import torch

from torch import nn

from torch.nn import init

from torch.nn.parameter import Parameter
```

```
class ShuffleAttention(nn.Module):

    # 初始化 Shuffle Attention 模块

    def __init__(self, channel=512, reduction=16, G=8):

        super().__init__()

        self.G = G # 分组数量

        self.channel = channel # 通道数

        self.avg_pool = nn.AdaptiveAvgPool2d(1) # 全局平均池化，用于生成通道注意力

        self.gn = nn.GroupNorm(channel // (2 * G), channel // (2 * G)) # 分组归一化，用于空间注意力

        # 以下为通道注意力和空间注意力的权重和偏置参数

        self.cweight = Parameter(torch.zeros(1, channel // (2 * G), 1, 1))

        self.cbias = Parameter(torch.ones(1, channel // (2 * G), 1, 1))

        self.sweight = Parameter(torch.zeros(1, channel // (2 * G), 1, 1))

        self.sbias = Parameter(torch.ones(1, channel // (2 * G), 1, 1))

        self.sigmoid = nn.Sigmoid() # Sigmoid 函数，用于生成注意力图

    # 权重初始化方法

    def init_weights(self):

        for m in self.modules():

            if isinstance(m, nn.Conv2d):

                init.kaiming_normal_(m.weight, mode='fan_out')

                if m.bias is not None:

                    init.constant_(m.bias, 0)
```

```
elif isinstance(m, nn.BatchNorm2d):
```

```
    init.constant_(m.weight, 1)
```

```
    init.constant_(m.bias, 0)
```

```
elif isinstance(m, nn.Linear):
```

```
    init.normal_(m.weight, std=0.001)
```

```
    if m.bias is not None:
```

```
        init.constant_(m.bias, 0)
```

```
# 通道混洗方法，用于在分组处理后重组特征
```

```
@staticmethod
```

```
def channel_shuffle(x, groups):
```

```
    b, c, h, w = x.shape
```

```
    x = x.reshape(b, groups, -1, h, w)
```

```
    x = x.permute(0, 2, 1, 3, 4)
```

```
    x = x.reshape(b, -1, h, w)
```

```
    return x
```

```
# 前向传播方法
```

```
def forward(self, x):
```

```
    b, c, h, w = x.size()
```

```
    x = x.view(b * self.G, -1, h, w) # 将输入特征图按照分组维度进行重排
```



```
x_0, x_1 = x.chunk(2, dim=1) # 将特征图分为两部分，分别用于通道注意力和空间注意力
```

```
# 通道注意力分支
```

```
x_channel = self.avg_pool(x_0) # 对第一部分应用全局平均池化
```

```
x_channel = self.cweight * x_channel + self.cbias # 应用学习到的权重和偏置
```

```
x_channel = x_0 * self.sigmoid(x_channel) # 通过 sigmoid 激活函数和原始特征图相乘，得到加权的特征图
```

```
# 空间注意力分支
```

```
x_spatial = self.gn(x_1) # 对第二部分应用分组归一化
```

```
x_spatial = self.sweight * x_spatial + self.sbias # 应用学习到的权重和偏置
```

```
x_spatial = x_1 * self.sigmoid(x_spatial) # 通过 sigmoid 激活函数和原始特征图相乘，得到加权的特征图
```

```
# 将通道注意力和空间注意力的结果沿通道维度拼接
```

```
out = torch.cat([x_channel, x_spatial], dim=1)
```

```
out = out.contiguous().view(b, -1, h, w) # 重新调整形状以匹配原始输入的维度
```

```
# 应用通道混洗，以便不同分组间的特征可以交换信息
```

```
out = self.channel_shuffle(out, 2)
```

```
return out
```

```
# 输入 N C H W， 输出 N C H W
```

```
if __name__ == '__main__':
```

```
    input = torch.randn(50, 512, 7, 7)
```

```
    se = ShuffleAttention(channel=512, G=8)
```

```
    output = se(input)
```

```
    print(output.shape)
```

## 29、ResNeSt 模块

论文《ResNeSt: Split-Attention Networks》

### 1、作用

ResNeSt 提出了一种新的模块化分裂注意力（Split-Attention）块，通过在特征图组间实现注意力机制。通过堆叠这些分裂注意力块，以 ResNet 风格构建，形成了新的 ResNet 变体，即 ResNeSt。该网络保留了整体的 ResNet 结构，便于在不增加额外计算成本的情况下，直接用于下游任务。

### 2、机制

1、ResNeSt 通过分裂注意力块对特征图组进行处理，使得每个组的特征表示通过其子组的加权组合得到，权重基于全局上下文信息。这种方法有效地增强了跨通道信息的交互，从而获得更丰富的特征表示。

2、分裂注意力块包括特征图分组和分裂注意力两个操作。首先将输入特征图分为多个组（卡片），然后在每个卡片内进一步细分为若干子组（基数），通过学习得到的权重对这些子组进行加权和，以获得每个卡片的表示，最后将所有卡片的表示合并起来，形成块的输出。

### 3、独特优势

1、ResNeSt 在不增加额外计算成本的前提下，显著提高了模型的性能。例如，ResNeSt-50 在 ImageNet 上达到了 81.13% 的顶级 1 准确率，比以前最好的 ResNet 变体提高了 1% 以上。这一改进也有助于下游任务，包括目标检测、实例分割和语义分割。

2、通过简单替换 ResNet-50 背骨为 ResNeSt-50, 即可在 MS-COCO 上将 Faster-RCNN 的 mAP 从 39.3% 提高到 42.3%, 并将 ADE20K 上 DeeplabV3 的 mIoU 从 42.1% 提高到 45.1% 。

#### 4、代码

```
import torch

from torch import nn

import torch.nn.functional as F

# 用于调整数值，使其可以被某个除数整除，常用于网络层中通道数的设置。

def make_divisible(v, divisor=8, min_value=None, round_limit=.9):

    min_value = min_value or divisor

    new_v = max(min_value, int(v + divisor / 2) // divisor * divisor)

    # 确保减小的百分比不超过一定的比例 (round_limit)

    if new_v < round_limit * v:

        new_v += divisor

    return new_v

# Radix Softmax 用于处理分组特征的归一化

class RadixSoftmax(nn.Module):

    def __init__(self, radix, cardinality):

        super().__init__()

        self.radix = radix

        self.cardinality = cardinality
```

```
def forward(self, x):
```

```
    batch = x.size(0)
```

```
    # 根据 radix 是否大于 1 来决定使用 softmax 还是 sigmoid 进行归一化
```

```
    if self.radix > 1:
```

```
        x = x.view(batch, self.cardinality, self.radix, -1).transpose(1, 2)
```

```
        x = F.softmax(x, dim=1)
```

```
        x = x.reshape(batch, -1)
```

```
    else:
```

```
        x = x.sigmoid()
```

```
    return x
```

```
# SplitAttn 模块实现分裂注意力机制
```

```
class SplitAttn(nn.Module):
```

```
    def __init__(self, in_channels, out_channels=None, kernel_size=3, stride=1, padding=None,
```

```
                  dilation=1, groups=1, bias=False, radix=2, rd_ratio=0.25, rd_channels=None, rd_divisor=8,
```

```
                  act_layer=nn.ReLU, norm_layer=None, drop_block=None, **kwargs):
```

```
        super(SplitAttn, self).__init__()
```

```
        out_channels = out_channels or in_channels
```

```
        self.radix = radix
```

```
        self.drop_block = drop_block
```

```
        mid_chs = out_channels * radix
```

```
# 根据输入通道数、radix 和 rd_ratio 计算注意力机制的中间层通道数
```

```
if rd_channels is None:
```

```
    attn_chs = make_divisible(
```

```
        in_channels * radix * rd_ratio, min_value=32, divisor=rd_divisor)
```

```
    else:
```

```
        attn_chs = rd_channels * radix
```

```
padding = kernel_size // 2 if padding is None else padding
```

```
# 核心卷积层
```

```
self.conv = nn.Conv2d(
```

```
    in_channels, mid_chs, kernel_size, stride, padding, dilation,
```

```
    groups=groups * radix, bias=bias, **kwargs)
```

```
# 后续层以及 RadixSoftmax
```

```
self.bn0 = norm_layer(mid_chs) if norm_layer else nn.Identity()
```

```
self.act0 = act_layer()
```

```
self.fc1 = nn.Conv2d(out_channels, attn_chs, 1, groups=groups)
```

```
self.bn1 = norm_layer(attn_chs) if norm_layer else nn.Identity()
```

```
self.act1 = act_layer()
```

```
self.fc2 = nn.Conv2d(attn_chs, mid_chs, 1, groups=groups)
```

```
self.rsoftmax = RadixSoftmax(radix, groups)
```

```
def forward(self, x):
```

```
# 卷积和激活
```

```
x = self.conv(x)
```

```
x = self.bn0(x)
```

```
if self.drop_block is not None:
```

```
    x = self.drop_block(x)
```

```
x = self.act0(x)
```

```
# 计算分裂注意力
```

```
B, RC, H, W = x.shape
```

```
if self.radix > 1:
```

```
    # 对特征进行重组和聚合
```

```
    x = x.reshape((B, self.radix, RC // self.radix, H, W))
```

```
    x_gap = x.sum(dim=1)
```

```
else:
```

```
    x_gap = x
```

```
# 全局平均池化和两层全连接网络，应用 RadixSoftmax
```

```
x_gap = x_gap.mean(2, keepdims=True).mean(3, keepdims=True)
```

```
x_gap = self.fc1(x_gap)
```

```
x_gap = self.bn1(x_gap)
```

```
x_gap = self.act1(x_gap)
```

```
x_attn = self.fc2(x_gap)
```

```
x_attn = self.rsoftmax(x_attn).view(B, -1, 1, 1)
```

```
if self.radix > 1:
```

```
    out = (x * x_attn.reshape((B, self.radix, RC // self.radix, 1, 1))).sum(dim=1)
```

```
else:
```

```
    out = x * x_attn
```

```
return out
```

```
# 输入 N C H W, 输出 N C H W
```

```
if __name__ == '__main__':
```

```
    block = SplitAttn(64)
```

```
    input = torch.rand(1, 64, 64, 64)
```

```
    output = block(input)
```

```
    print(output.shape)
```

## 30、Partnet 模块

论文《NON-DEEP NETWORKS》

### 1、作用

论文提出了“Partnet”，这是一种新型的神经网络架构，旨在不依赖传统的深层架构就能在视觉识别任务中达到高性能。展示了在大规模基准测试如 ImageNet、CIFAR-10 和 CIFAR-100 上，即便是层数大大减少（大约 12 层）的网络也能够保持竞争力。

## 2、机制

1、Partnet 采用并行子网络而不是传统的顺序层叠，从而减少了网络深度，同时保持或甚至增强了性能。这种“不尴不尬的并行”设计使得在不妥协模型学习复杂特征的能力下有效减少了计算深度。

2、该架构通过改良的 VGG 风格块，加入了跳跃连接、压缩和激励（Skip-Squeeze-and-Excitation, SSE）层，并使用了 SiLU（Sigmoid Linear Unit）激活函数，以解决由于深度减少可能导致的表示能力限制问题。

## 3、独特优势

1、并行化：

Partnet 的并行结构允许有效地将计算分布在多个处理单元上，显著提高了推理速度，而不增加网络的深度。

2、低延迟：

通过保持浅层架构，Partnet 能够实现低延迟推理，适用于需要快速响应的应用场景。

3、可扩展性：

论文探索了如何通过增加宽度、分辨率和并行分支的数量来扩展 Partnet，同时保持网络深度恒定。这种可扩展方法为进一步提高性能提供了路径，而不增加延迟

## 4、代码

```
import numpy as np

import torch

from torch import nn

from torch.nn import init


class ParNetAttention(nn.Module):

    # 初始化 ParNet 注意力模块
```



```
def __init__(self, channel=512):
```

```
    super().__init__()
```

```
    # 使用自适应平均池化和 1x1 卷积实现空间压缩，然后通过 Sigmoid 激活函数产生权重图
```

```
    self.sse = nn.Sequential(
```

```
        nn.AdaptiveAvgPool2d(1), # 全局平均池化，将空间维度压缩到 1x1
```

```
        nn.Conv2d(channel, channel, kernel_size=1), # 1x1 卷积，用于调整通道的权重
```

```
        nn.Sigmoid() # Sigmoid 函数，用于生成注意力图
```

```
    )
```

```
    # 通过 1x1 卷积实现特征重映射，不改变空间尺寸
```

```
    self.conv1x1 = nn.Sequential(
```

```
        nn.Conv2d(channel, channel, kernel_size=1), # 1x1 卷积，不改变特征图的空间尺寸
```

```
        nn.BatchNorm2d(channel) # 批量归一化
```

```
    )
```

```
    # 通过 3x3 卷积捕获空间上下文信息
```

```
    self.conv3x3 = nn.Sequential(
```

```
        nn.Conv2d(channel, channel, kernel_size=3, padding=1), # 3x3 卷积，保持特征图尺寸不变
```

```
        nn.BatchNorm2d(channel) # 批量归一化
```

```
    )
```

```
    self.silu = nn.SiLU() # SiLU 激活函数，也被称为 Swish 函数
```

```
def forward(self, x):
```

```
# x 是输入的特征图，形状为(Batch, Channel, Height, Width)
```

```
b, c, _, _ = x.size()
```

```
x1 = self.conv1x1(x) # 通过 1x1 卷积处理 x
```

```
x2 = self.conv3x3(x) # 通过 3x3 卷积处理 x
```

```
x3 = self.sse(x) * x # 应用空间压缩的注意力权重到 x 上
```

```
y = self.silu(x1 + x2 + x3) # 将上述三个结果相加并通过 SiLU 激活函数激活，获得最终输出
```

```
return y
```

```
# 测试 ParNetAttention 模块
```

```
if __name__ == '__main__':
```

```
input = torch.randn(3, 512, 7, 7) # 创建一个随机输入
```

```
pna = ParNetAttention(channel=512) # 实例化 ParNet 注意力模块
```

```
output = pna(input) # 对输入进行处理
```

```
print(output.shape) # 打印输出的形状，预期为(3, 512, 7, 7)
```

## 31、Cloattention 模块

论文《Rethinking Local Perception in Lightweight Vision Transformer》

### 1、作用

CloFormer（Context-aware Local Enhancement Vision Transformer）是一种轻量级的视觉 Transformer，用于在保持模型轻量化的同时，提高在各种视觉任务中的性能，包括图像分类、目标检测和语义分割。其主要目的是提升移动设备上的视觉模型性能，克服直接缩减标准 ViT（Vision Transformer）模型尺寸导致的性能下降问题。

## 2、机制

CloFormer 通过引入 AttnConv（Attention Style Convolution Operator）来实现上下文感知的局部增强，从而有效捕获高频局部信息。该模型采用两分支结构：

### 1、局部分支：

利用 AttnConv 融合共享权重和上下文感知权重来聚合高频局部信息。首先，使用深度可分离卷积（Depthwise Convolution，DWconv）提取局部表示，然后部署上下文感知权重来增强局部特征。

### 2、全局分支：

采用标准的注意力机制，通过对 K 和 V 进行下采样来降低 FLOPs，帮助模型捕捉低频全局信息。

## 3、独特优势

### 1、上下文感知的局部增强：

通过 AttnConv，CloFormer 有效地结合了共享权重和上下文感知权重的优势，实现了高质量的局部特征增强。

### 2、两分支结构：

通过同时捕获高频和低频信息，模型能够在不同的视觉任务中达到更好的性能。

### 3、轻量化设计：

CloFormer 专为移动设备设计，通过精心的模型架构设计和权重共享机制，实现了在保持轻量化的同时提高模型性能。

## 4、代码

```
import torch

from torch import nn
```

```
from efficientnet_pytorch.model import MemoryEfficientSwish
```

```
# 定义一个通过卷积和激活函数生成注意力图的模块
```

```
class AttnMap(nn.Module):
```

```
    def __init__(self, dim):
```

```
        super().__init__()
```

```
        self.act_block = nn.Sequential(
```

```
            nn.Conv2d(dim, dim, 1, 1, 0), # 1x1 卷积用于调整通道数
```

```
            MemoryEfficientSwish(), # 使用 MemoryEfficientSwish 作为激活函数
```

```
            nn.Conv2d(dim, dim, 1, 1, 0) # 再次 1x1 卷积
```

```
        )
```

```
    def forward(self, x):
```

```
        return self.act_block(x)
```

```
# 定义高效注意力机制的主体模块
```

```
class EfficientAttention(nn.Module):
```

```
    def __init__(self, dim, num_heads=8, group_split=[4, 4], kernel_sizes=[5], window_size=4,
```

```
                attn_drop=0., proj_drop=0., qkv_bias=True):
```

```
        super().__init__()
```

```
        # 参数初始化和定义
```

```
        assert sum(group_split) == num_heads # 确保分组数量之和等于头的数量
```

```
assert len(kernel_sizes) + 1 == len(group_split) # 确保核大小列表加一等于分组数量
```

```
self.dim = dim # 输入通道数
```

```
self.num_heads = num_heads # 注意力头的数量
```

```
self.dim_head = dim // num_heads # 每个头的维度
```

```
self.scalor = self.dim_head ** -0.5 # 缩放因子
```

```
self.kernel_sizes = kernel_sizes # 核大小列表
```

```
self.window_size = window_size # 窗口大小
```

```
self.group_split = group_split # 分组列表
```

```
# 根据核大小和分组定义卷积层、注意力映射层和 QKV 层
```

```
convs = []
```

```
act_blocks = []
```

```
qkvs = []
```

```
for i in range(len(kernel_sizes)):
```

```
    kernel_size = kernel_sizes[i]
```

```
    group_head = group_split[i]
```

```
    if group_head == 0:
```

```
        continue
```

```
    convs.append(nn.Conv2d(3 * self.dim_head * group_head, 3 * self.dim_head * group_head, kernel_size,
```

```
                           1, kernel_size // 2, groups=3 * self.dim_head * group_head))
```

```
    act_blocks.append(AttnMap(self.dim_head * group_head))
```

```
    qkvs.append(nn.Conv2d(dim, 3 * group_head * self.dim_head, 1, 1, 0, bias=qkv_bias))
```

```
if group_split[-1] != 0:
```

```
# 对最后一个全局注意力头的定义
```

```
self.global_q = nn.Conv2d(dim, group_split[-1] * self.dim_head, 1, 1, 0, bias=qkv_bias)
```

```
self.global_kv = nn.Conv2d(dim, group_split[-1] * self.dim_head * 2, 1, 1, 0, bias=qkv_bias)
```

```
self.avgpool = nn.AvgPool2d(window_size, window_size) if window_size != 1 else nn.Identity()
```

```
# 将定义的模块注册为子模块
```

```
self.convs = nn.ModuleList(convs)
```

```
self.act_blocks = nn.ModuleList(act_blocks)
```

```
self.qkvs = nn.ModuleList(qkvs)
```

```
self.proj = nn.Conv2d(dim, dim, 1, 1, 0, bias=qkv_bias) # 输出投影层
```

```
self.attn_drop = nn.Dropout(attn_drop) # 注意力 dropout
```

```
self.proj_drop = nn.Dropout(proj_drop) # 投影 dropout
```

```
# 高频注意力处理函数
```

```
def high_fre_attntion(self, x: torch.Tensor, to_qkv: nn.Module, mixer: nn.Module, attn_block: nn.Module):
```

```
    '''
```

```
    x: (b c h w)
```

```
    '''
```

```
    b, c, h, w = x.size()
```

```
    qkv = to_qkv(x) # (b (3 m d) h w)
```

```
    qkv = mixer(qkv).reshape(b, 3, -1, h, w).transpose(0, 1).contiguous() # (3 b (m d) h w)
```

```
    q, k, v = qkv # (b (m d) h w)
```

```
attn = attn_block(q.mul(k)).mul(self.scalor)
```

```
attn = self.attn_drop(torch.tanh(attn))
```

```
res = attn.mul(v) # (b (m d) h w)
```

```
return res
```

```
# 低频注意力处理函数
```

```
def low_fre_attention(self, x: torch.Tensor, to_q: nn.Module, to_kv: nn.Module, avgpool: nn.Module):
```

```
'''
```

```
x: (b c h w)
```

```
'''
```

```
b, c, h, w = x.size()
```

```
q = to_q(x).reshape(b, -1, self.dim_head, h * w).transpose(-1, -2).contiguous() # (b m (h w) d)
```

```
kv = avgpool(x) # (b c h w)
```

```
kv = to_kv(kv).view(b, 2, -1, self.dim_head, (h * w) // (self.window_size ** 2)).permute(1, 0, 2, 4,
```

```
3).contiguous() #
```

```
(2 b m (H W) d)
```

```
k, v = kv # (b m (H W) d)
```

```
attn = self.scalor * q @ k.transpose(-1, -2) # (b m (h w) (H W))
```

```
attn = self.attn_drop(attn.softmax(dim=-1))
```

```
res = attn @ v # (b m (h w) d)
```

```
res = res.transpose(2, 3).reshape(b, -1, h, w).contiguous()
```

```
return res
```

```
# 模块的前向传播
```

```
def forward(self, x: torch.Tensor):
```

```
    '''
```

```
    x: (b c h w)
```

```
    '''
```

```
    res = []
```

```
    for i in range(len(self.kernel_sizes)):
```

```
        if self.group_split[i] == 0:
```

```
            continue
```

```
            res.append(self.high_fre_attntion(x, self.qkvs[i], self.convs[i], self.act_blocks[i]))
```

```
        if self.group_split[-1] != 0:
```

```
            res.append(self.low_fre_attention(x, self.global_q, self.global_kv, self.avgpool))
```

```
    return self.proj_drop(self.proj(torch.cat(res, dim=1)))
```

```
# 输入 N C HW, 输出 N C H W
```

```
if __name__ == '__main__':
```

```
    block = EfficientAttention(64).cuda()
```

```
    input = torch.rand(1, 64, 64, 64).cuda()
```

```
    output = block(input)
```

```
    print(output.shape)
```



# 32、BiFormer 模块

论文《BiFormer: Vision Transformer with Bi-Level Routing Attention》

## 1、作用

BiFormer 旨在解决视觉 Transformer 在处理图像时的计算和内存效率问题。它通过引入双层路由注意力（Bi-Level Routing Attention, BRA），实现了动态的、基于内容的稀疏注意力机制，以更灵活、高效地分配计算资源。

## 2、机制

BiFormer 的核心是双层路由注意力（BRA），该机制包含两个主要步骤：区域到区域的路由和令牌到令牌的注意力。首先，通过构建一个区域级别的关联图并对其进行修剪，来确定哪些区域是相关的，并应该被进一步考虑。其次，在这些选定的区域中，应用细粒度的令牌到令牌注意力，以便每个查询仅与少数最相关的键-值对进行交互。这种方法允许 BiFormer 动态地关注图像中与特定查询最相关的部分，而不是在所有空间位置上计算成对的令牌交互，从而显著减少了计算复杂度和内存占用。

## 3、独特优势

### 1、计算效率：

BiFormer 通过其双层路由注意力机制，实现了与传统全局注意力相比的显著计算和内存效率改进，具体体现在能够动态地仅对最相关的令牌子集进行计算。

### 2、动态稀疏性：

与其他稀疏注意力方法不同，BiFormer 能够根据内容动态选择关注的区域和令牌，使其能够更有效地处理各种视觉任务。

### 3、高性能：

实验结果表明，BiFormer 在图像分类、对象检测和语义分割等多个视觉任务上实现了优异的性能，尤其是在与模型大小和计算复杂度相当的情况下，其性能超越了现有的最先进方法。

## 4、代码

```
from typing import Tuple, Optional
```

```
import torch
```

```
import torch.nn as nn
```

```
import torch.nn.functional as F
```

```
from einops import rearrange
```

```
from torch import Tensor, LongTensor
```

```
class TopkRouting(nn.Module):
```

```
    """
```

```
    differentiable topk routing with scaling
```

```
    Args:
```

```
        qk_dim: int, feature dimension of query and key
```

```
        topk: int, the 'topk'
```

```
        qk_scale: int or None, temperature (multiply) of softmax activation
```

```
        with_param: bool, wether incorporate learnable params in routing unit
```

```
        diff_routing: bool, wether make routing differentiable
```

```
        soft_routing: bool, wether make output value multiplied by routing weights
```

```
    """
```

```
    def __init__(self, qk_dim, topk=4, qk_scale=None, param_routing=False, diff_routing=False):
```

```

super().__init__()

self.topk = topk

self.qk_dim = qk_dim

self.scale = qk_scale or qk_dim ** -0.5

self.diff_routing = diff_routing

# TODO: norm layer before/after linear?

self.emb = nn.Linear(qk_dim, qk_dim) if param_routing else nn.Identity()

# routing activation

self.routing_act = nn.Softmax(dim=-1)


def forward(self, query: Tensor, key: Tensor) -> Tuple[Tensor]:

    """

    Args:

        q, k: (n, p^2, c) tensor

    Return:

        r_weight, topk_index: (n, p^2, topk) tensor

    """

    if not self.diff_routing:

        query, key = query.detach(), key.detach()

    query_hat, key_hat = self.emb(query), self.emb(key) # per-window pooling -> (n, p^2, c)

    attn_logits = (query_hat * self.scale) @ key_hat.transpose(-2, -1) # (n, p^2, p^2)

    topk_attn_logits, topk_index = torch.topk(attn_logits, k=self.topk, dim=-1) # (n, p^2, k), (n, p^2, k)

```

```
r_weight = self.routing_act(topk_attn_logits) # (n, p^2, k)
```

```
return r_weight, topk_index
```

```
class KVgather(nn.Module):
```

```
def __init__(self, mul_weight='none'):
```

```
    super().__init__()
```

```
    assert mul_weight in ['none', 'soft', 'hard']
```

```
    self.mul_weight = mul_weight
```

```
def forward(self, r_idx: Tensor, r_weight: Tensor, kv: Tensor):
```

```
    """
```

```
    r_idx: (n, p^2, topk) tensor
```

```
    r_weight: (n, p^2, topk) tensor
```

```
    kv: (n, p^2, w^2, c_kq+c_v)
```

```
    Return:
```

```
        (n, p^2, topk, w^2, c_kq+c_v) tensor
```

```
    """
```

```
    # select kv according to routing index
```

```
    n, p2, w2, c_kv = kv.size()
```

```

topk = r_idx.size(-1)

# print(r_idx.size(), r_weight.size())

# FIXME: gather consumes much memory (topk times redundancy), write cuda kernel?

topk_kv = torch.gather(kv.view(n, 1, p2, w2, c_kv).expand(-1, p2, -1, -1, -1),

                        # (n, p^2, p^2, w^2, c_kv) without mem cpy

                        dim=2,

                        index=r_idx.view(n, p2, topk, 1, 1).expand(-1, -1, -1, w2, c_kv)

                        # (n, p^2, k, w^2, c_kv)

                        )

```

```

if self.mul_weight == 'soft':

    topk_kv = r_weight.view(n, p2, topk, 1, 1) * topk_kv # (n, p^2, k, w^2, c_kv)

elif self.mul_weight == 'hard':

    raise NotImplementedError('differentiable hard routing TBA')

# else: # 'none'

#     topk_kv = topk_kv # do nothing

return topk_kv

```

```

class QKVLinear(nn.Module):

    def __init__(self, dim, qk_dim, bias=True):

```

```

    super().__init__()

    self.dim = dim

    self.qk_dim = qk_dim

    self.qkv = nn.Linear(dim, qk_dim + qk_dim + dim, bias=bias)

def forward(self, x):

    q, kv = self.qkv(x).split([self.qk_dim, self.qk_dim + self.dim], dim=-1)

    return q, kv

    # q, k, v = self.qkv(x).split([self.qk_dim, self.qk_dim, self.dim], dim=-1)

    # return q, k, v


class BiLevelRoutingAttention(nn.Module):

    """

    n_win: number of windows in one side (so the actual number of windows is n_win*n_win)

    kv_per_win: for kv_downsample_mode='ada_xxxpool' only, number of key/values per window. Similar to n_win,
    the actual number is kv_per_win*kv_per_win.

    topk: topk for window filtering

    param_attention: 'qkvo'-linear for q,k,v and o, 'none': param free attention

    param_routing: extra linear for routing

    diff_routing: wether to set routing differentiable

    soft_routing: wether to multiply soft routing weights

    """

```

```

def __init__(self, dim, n_win=7, num_heads=8, qk_dim=None, qk_scale=None,

              kv_per_win=4, kv_downsample_ratio=4, kv_downsample_kernel=None,
kv_downsample_mode='identity',

              topk=4, param_attention="qkvo", param_routing=False, diff_routing=False, soft_routing=False,

              side_dwconv=3,

              auto_pad=True):

    super().__init__()

    # local attention setting

    self.dim = dim

    self.n_win = n_win # Wh, Ww

    self.num_heads = num_heads

    self.qk_dim = qk_dim or dim

    assert self.qk_dim % num_heads == 0 and self.dim % num_heads == 0, 'qk_dim and dim must be divisible
by num_heads!'

    self.scale = qk_scale or self.qk_dim ** -0.5

    #####side_dwconv (i.e. LCE in ShuntedTransformer)#####

    self.lepe = nn.Conv2d(dim, dim, kernel_size=side_dwconv, stride=1, padding=side_dwconv // 2,

                           groups=dim) if side_dwconv > 0 else \

        lambda x: torch.zeros_like(x)

    ##### global routing setting #####

```

```

self.topk = topk

self.param_routing = param_routing

self.diff_routing = diff_routing

self.soft_routing = soft_routing

# router

assert not (self.param_routing and not self.diff_routing) # cannot be with_param=True and
diff_routing=False

self.router = TopkRouting(qk_dim=self.qk_dim,

                           qk_scale=self.scale,

                           topk=self.topk,

                           diff_routing=self.diff_routing,

                           param_routing=self.param_routing)

if self.soft_routing: # soft routing, always differentiable (if no detach)

    mul_weight = 'soft'

elif self.diff_routing: # hard differentiable routing

    mul_weight = 'hard'

else: # hard non-differentiable routing

    mul_weight = 'none'

self.kv_gather = KV Gather(mul_weight=mul_weight)

# qkv mapping (shared by both global routing and local attention)

self.param_attention = param_attention

if self.param_attention == 'qkvo':

```



```

        self.qkv = QKVLinear(self.dim, self.qk_dim)

        self.wo = nn.Linear(dim, dim)

    elif self.param_attention == 'qkv':

        self.qkv = QKVLinear(self.dim, self.qk_dim)

        self.wo = nn.Identity()

    else:

        raise ValueError(f'param_attention mode {self.param_attention} is not supported!')

    self.kv_downsample_mode = kv_downsample_mode

    self.kv_per_win = kv_per_win

    self.kv_downsample_ratio = kv_downsample_ratio

    self.kv_downsample_kernel = kv_downsample_kernel

    if self.kv_downsample_mode == 'ada_avgpool':

        assert self.kv_per_win is not None

        self.kv_down = nn.AdaptiveAvgPool2d(self.kv_per_win)

    elif self.kv_downsample_mode == 'ada_maxpool':

        assert self.kv_per_win is not None

        self.kv_down = nn.AdaptiveMaxPool2d(self.kv_per_win)

    elif self.kv_downsample_mode == 'maxpool':

        assert self.kv_downsample_ratio is not None

        self.kv_down = nn.MaxPool2d(self.kv_downsample_ratio) if self.kv_downsample_ratio > 1 else
nn.Identity()

    elif self.kv_downsample_mode == 'avgpool':

```

```

        assert self.kv_downsample_ratio is not None

        self.kv_down = nn.AvgPool2d(self.kv_downsample_ratio) if self.kv_downsample_ratio > 1 else
nn.Identity()

    elif self.kv_downsample_mode == 'identity': # no kv downsampling

        self.kv_down = nn.Identity()

    elif self.kv_downsample_mode == 'fracpool':

        # assert self.kv_downsample_ratio is not None

        # assert self.kv_downsample_kenel is not None

        # TODO: fracpool

        # 1. kernel size should be input size dependent

        # 2. there is a random factor, need to avoid independent sampling for k and v

        raise NotImplementedError('fracpool policy is not implemented yet!')

    elif kv_downsample_mode == 'conv':

        # TODO: need to consider the case where k != v so that need two downsample modules

        raise NotImplementedError('conv policy is not implemented yet!')

    else:

        raise ValueError(f'kv_down_sample_mode {self.kv_downsaple_mode} is not surpported!')

    # softmax for local attention

    self.attn_act = nn.Softmax(dim=-1)

    self.auto_pad = auto_pad

```

```

def forward(self, x, ret_attn_mask=False):

    """

    x: NHWC tensor


    Return:

        NHWC tensor

    """

    x = rearrange(x, "n c h w -> n h w c")

    # NOTE: use padding for semantic segmentation

    #####

    if self.auto_pad:

        N, H_in, W_in, C = x.size()

        pad_l = pad_t = 0

        pad_r = (self.n_win - W_in % self.n_win) % self.n_win

        pad_b = (self.n_win - H_in % self.n_win) % self.n_win

        x = F.pad(x, (0, 0, # dim=-1

                     pad_l, pad_r, # dim=-2

                     pad_t, pad_b)) # dim=-3

        _, H, W, _ = x.size() # padded size

    else:

        N, H, W, C = x.size()

```

```

        assert H % self.n_win == 0 and W % self.n_win == 0 #

#####

# patchify, (n, p^2, w, w, c), keep 2d window as we need 2d pooling to reduce kv size

x = rearrange(x, "n (j h) (i w) c -> n (j i) h w c", j=self.n_win, i=self.n_win)

#####qkv projection#####

# q: (n, p^2, w, w, c_qk)

# kv: (n, p^2, w, w, c_qk+c_v)

# NOTE: separte kv if there were memory leak issue caused by gather

q, kv = self.qkv(x)

# pixel-wise qkv

# q_pix: (n, p^2, w^2, c_qk)

# kv_pix: (n, p^2, h_kv*w_kv, c_qk+c_v)

q_pix = rearrange(q, 'n p2 h w c -> n p2 (h w) c')

kv_pix = self.kv_down(rearrange(kv, 'n p2 h w c -> (n p2) c h w'))

kv_pix = rearrange(kv_pix, '(n j i) c h w -> n (j i) (h w) c', j=self.n_win, i=self.n_win)

q_win, k_win = q.mean([2, 3]), kv[..., 0:self.qk_dim].mean(

    [2, 3]) # window-wise qk, (n, p^2, c_qk), (n, p^2, c_qk)

```

```

#####side_dwconv(lepe)#####

# NOTE: call contiguous to avoid gradient warning when using ddp

lepe = self.lepe(rearrange(kv[... , self.qk_dim:], 'n (j i) h w c -> n c (j h) (i w)', j=self.n_win,
                           i=self.n_win).contiguous())

lepe = rearrange(lepe, 'n c (j h) (i w) -> n (j h) (i w) c', j=self.n_win, i=self.n_win)

##### gather q dependent k/v #####

r_weight, r_idx = self.router(q_win, k_win) # both are (n, p^2, topk) tensors

kv_pix_sel = self.kv_gather(r_idx=r_idx, r_weight=r_weight, kv=kv_pix) # (n, p^2, topk, h_kv*w_kv,
c_qk+c_v)

k_pix_sel, v_pix_sel = kv_pix_sel.split([self.qk_dim, self.dim], dim=-1)

# kv_pix_sel: (n, p^2, topk, h_kv*w_kv, c_qk)

# v_pix_sel: (n, p^2, topk, h_kv*w_kv, c_v)

##### do attention as normal #####

k_pix_sel = rearrange(k_pix_sel, 'n p2 k w2 (m c) -> (n p2) m c (k w2)',
                      m=self.num_heads) # flatten to BMLC, (n*p^2, m, topk*h_kv*w_kv, c_kq//m)
transpose here?

v_pix_sel = rearrange(v_pix_sel, 'n p2 k w2 (m c) -> (n p2) m (k w2) c',
                      m=self.num_heads) # flatten to BMLC, (n*p^2, m, topk*h_kv*w_kv, c_v//m)

q_pix = rearrange(q_pix, 'n p2 w2 (m c) -> (n p2) m w2 c',

```

```

        m=self.num_heads) # to BMLC tensor (n*p^2, m, w^2, c_qk//m)

# param-free multihead attention

attn_weight = (

    q_pix * self.scale) @ k_pix_sel # (n*p^2, m, w^2, c) @ (n*p^2, m, c, topk*h_kv*w_kv)
-> (n*p^2, m, w^2, topk*h_kv*w_kv)

    attn_weight = self.attn_act(attn_weight)

    out = attn_weight @ v_pix_sel # (n*p^2, m, w^2, topk*h_kv*w_kv) @ (n*p^2, m, topk*h_kv*w_kv, c) -> (n*p^2,
m, w^2, c)

    out = rearrange(out, '(n j i) m (h w) c -> n (j h) (i w) (m c)', j=self.n_win, i=self.n_win,

        h=H // self.n_win, w=W // self.n_win)

    out = out + lepe

# output linear

    out = self.wo(out)

# NOTE: use padding for semantic segmentation

# crop padded region

    if self.auto_pad and (pad_r > 0 or pad_b > 0):

        out = out[:, :H_in, :W_in, :].contiguous()

    if ret_attn_mask:

        return out, r_weight, r_idx, attn_weight

```

```
else:
```

```
    return rearrange(out, "n h w c -> n c h w")
```

```
class Attention(nn.Module):
```

```
    """
```

```
    vanilla attention
```

```
    """
```

```
def __init__(self, dim, num_heads=8, qkv_bias=False, qk_scale=None, attn_drop=0., proj_drop=0.):
```

```
    super().__init__()
```

```
    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 prev weights
```

```
    self.scale = qk_scale or head_dim ** -0.5
```

```
    self.qkv = nn.Linear(dim, dim * 3, bias=qkv_bias)
```

```
    self.attn_drop = nn.Dropout(attn_drop)
```

```
    self.proj = nn.Linear(dim, dim)
```

```
    self.proj_drop = nn.Dropout(proj_drop)
```

```
def forward(self, x):
```

```

"""

args:

    x: NCHW tensor

return:

    NCHW tensor

"""

_, _, H, W = x.size()

x = rearrange(x, 'n c h w -> n (h w) c')

#####

B, N, C = x.shape

qkv = self.qkv(x).reshape(B, N, 3, self.num_heads, C // self.num_heads).permute(2, 0, 3, 1, 4)

q, k, v = qkv[0], qkv[1], qkv[2] # make torchscript happy (cannot use tensor as tuple)

attn = (q @ k.transpose(-2, -1)) * self.scale

attn = attn.softmax(dim=-1)

attn = self.attn_drop(attn)

x = (attn @ v).transpose(1, 2).reshape(B, N, C)

x = self.proj(x)

x = self.proj_drop(x)

#####

```



```
x = rearrange(x, 'n (h w) c -> n c h w', h=H, w=W)
```

```
return x
```

```
class AttentionLePE(nn.Module):
```

```
    """
```

```
    vanilla attention
```

```
    """
```

```
    def __init__(self, dim, num_heads=8, qkv_bias=False, qk_scale=None, attn_drop=0., proj_drop=0.,  
side_dwconv=5):
```

```
        super().__init__()
```

```
        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 prev weights
```

```
        self.scale = qk_scale or head_dim ** -0.5
```

```
        self.qkv = nn.Linear(dim, dim * 3, bias=qkv_bias)
```

```
        self.attn_drop = nn.Dropout(attn_drop)
```

```
        self.proj = nn.Linear(dim, dim)
```

```
        self.proj_drop = nn.Dropout(proj_drop)
```

```
        self.lepe = nn.Conv2d(dim, dim, kernel_size=side_dwconv, stride=1, padding=side_dwconv // 2,
```

```

        groups=dim) if side_dwconv > 0 else \

    lambda x: torch.zeros_like(x)

def forward(self, x):

    """

    args:

        x: NCHW tensor

    return:

        NCHW tensor

    """

    _, _, H, W = x.size()

    x = rearrange(x, 'n c h w -> n (h w) c')

    #####

    B, N, C = x.shape

    qkv = self.qkv(x).reshape(B, N, 3, self.num_heads, C // self.num_heads).permute(2, 0, 3, 1, 4)

    q, k, v = qkv[0], qkv[1], qkv[2] # make torchscript happy (cannot use tensor as tuple)

    lepe = self.lepe(rearrange(x, 'n (h w) c -> n c h w', h=H, w=W))

    lepe = rearrange(lepe, 'n c h w -> n (h w) c')

    attn = (q @ k.transpose(-2, -1)) * self.scale

```

```

attn = attn.softmax(dim=-1)

attn = self.attn_drop(attn)

x = (attn @ v).transpose(1, 2).reshape(B, N, C)

x = x + lepe

```

```

x = self.proj(x)

x = self.proj_drop(x)

```

```

#####

```

```

x = rearrange(x, 'n (h w) c -> n c h w', h=H, w=W)

```

```

return x

```

```

def _grid2seq(x: Tensor, region_size: Tuple[int], num_heads: int):

```

```

    """

```

Args:

x: BCHW tensor

region size: int

num\_heads: number of attention heads

Return:

out: rearranged x, has a shape of (bs, nhead, nregion, reg\_size, head\_dim)

```

        region_h, region_w: number of regions per col/row

    """

    B, C, H, W = x.size()

    region_h, region_w = H // region_size[0], W // region_size[1]

    x = x.view(B, num_heads, C // num_heads, region_h, region_size[0], region_w, region_size[1])

    x = torch.einsum('bmdhwpq->bmhwpqd', x).flatten(2, 3).flatten(-3, -2) # (bs, nhead, nregion, reg_size,
head_dim)

    return x, region_h, region_w


def _seq2grid(x: Tensor, region_h: int, region_w: int, region_size: Tuple[int]):

    """

    Args:

        x: (bs, nhead, nregion, reg_size^2, head_dim)

    Return:

        x: (bs, C, H, W)

    """

    bs, nhead, nregion, reg_size_square, head_dim = x.size()

    x = x.view(bs, nhead, region_h, region_w, region_size[0], region_size[1], head_dim)

    x = torch.einsum('bmhwpqd->bmdhwpq', x).reshape(bs, nhead * head_dim,

                                                    region_h * region_size[0], region_w * region_size[1])

    return x

```

```

def regional_routing_attention_torch(

    query: Tensor, key: Tensor, value: Tensor, scale: float,

    region_graph: LongTensor, region_size: Tuple[int],

    kv_region_size: Optional[Tuple[int]] = None,

    auto_pad=True) -> Tensor:

    """

    Args:

        query, key, value: (B, C, H, W) tensor

        scale: the scale/temperature for dot product attention

        region_graph: (B, nhead, h_q*w_q, topk) tensor, topk <= h_k*w_k

        region_size: region/window size for queries, (rh, rw)

        key_region_size: optional, if None, key_region_size=region_size

        auto_pad: required to be true if the input sizes are not divisible by the region_size

    Return:

        output: (B, C, H, W) tensor

        attn: (bs, nhead, q_nregion, reg_size, topk*kv_region_size) attention matrix

    """

    kv_region_size = kv_region_size or region_size

    bs, nhead, q_nregion, topk = region_graph.size()

    # Auto pad to deal with any input size

```

```

q_pad_b, q_pad_r, kv_pad_b, kv_pad_r = 0, 0, 0, 0

if auto_pad:

    _, _, Hq, Wq = query.size()

    q_pad_b = (region_size[0] - Hq % region_size[0]) % region_size[0]

    q_pad_r = (region_size[1] - Wq % region_size[1]) % region_size[1]

    if (q_pad_b > 0 or q_pad_r > 0):

        query = F.pad(query, (0, q_pad_r, 0, q_pad_b)) # zero padding

    _, _, Hk, Wk = key.size()

    kv_pad_b = (kv_region_size[0] - Hk % kv_region_size[0]) % kv_region_size[0]

    kv_pad_r = (kv_region_size[1] - Wk % kv_region_size[1]) % kv_region_size[1]

    if (kv_pad_r > 0 or kv_pad_b > 0):

        key = F.pad(key, (0, kv_pad_r, 0, kv_pad_b)) # zero padding

        value = F.pad(value, (0, kv_pad_r, 0, kv_pad_b)) # zero padding

# to sequence format, i.e. (bs, nhead, nregion, reg_size, head_dim)

query, q_region_h, q_region_w = _grid2seq(query, region_size=region_size, num_heads=nhead)

key, _, _ = _grid2seq(key, region_size=kv_region_size, num_heads=nhead)

value, _, _ = _grid2seq(value, region_size=kv_region_size, num_heads=nhead)

# gather key and values.

# TODO: is seperate gathering slower than fused one (our old version) ?

```

```

# torch.gather does not support broadcasting, hence we do it manually

bs, nhead, kv_nregion, kv_region_size, head_dim = key.size()

broadcasted_region_graph = region_graph.view(bs, nhead, q_nregion, topk, 1, 1). \
    expand(-1, -1, -1, -1, kv_region_size, head_dim)

key_g = torch.gather(key.view(bs, nhead, 1, kv_nregion, kv_region_size, head_dim). \
    expand(-1, -1, query.size(2), -1, -1, -1), dim=3,
    index=broadcasted_region_graph) # (bs, nhead, q_nregion, topk, kv_region_size,
head_dim)

value_g = torch.gather(value.view(bs, nhead, 1, kv_nregion, kv_region_size, head_dim). \
    expand(-1, -1, query.size(2), -1, -1, -1), dim=3,
    index=broadcasted_region_graph) # (bs, nhead, q_nregion, topk, kv_region_size,
head_dim)

# token-to-token attention

# (bs, nhead, q_nregion, reg_size, head_dim) @ (bs, nhead, q_nregion, head_dim, topk*kv_region_size)

# -> (bs, nhead, q_nregion, reg_size, topk*kv_region_size)

# TODO: mask padding region

attn = (query * scale) @ key_g.flatten(-3, -2).transpose(-1, -2)

attn = torch.softmax(attn, dim=-1)

# (bs, nhead, q_nregion, reg_size, topk*kv_region_size) @ (bs, nhead, q_nregion, topk*kv_region_size,
head_dim)

# -> (bs, nhead, q_nregion, reg_size, head_dim)

output = attn @ value_g.flatten(-3, -2)

```

```
# to BCHW format
```

```
output = _seq2grid(output, region_h=q_region_h, region_w=q_region_w, region_size=region_size)
```

```
# remove paddings if needed
```

```
if auto_pad and (q_pad_b > 0 or q_pad_r > 0):
```

```
    output = output[:, :, :Hq, :Wq]
```

```
return output, attn
```

```
class BiLevelRoutingAttention_nchw(nn.Module):
```

```
    """Bi-Level Routing Attention that takes nchw input
```

```
    Compared to legacy version, this implementation:
```

```
    * removes unused args and components
```

```
    * uses nchw input format to avoid frequent permutation
```

```
    When the size of inputs is not divisible by the region size, there is also a numerical difference
```

```
    than legacy implementation, due to:
```

```
    * different way to pad the input feature map (padding after linear projection)
```

```
    * different pooling behavior (count_include_pad=False)
```



Current implementation is more reasonable, hence we do not keep backward numerical compatiability

"""

```
def __init__(self, dim, num_heads=8, n_win=7, qk_scale=None, topk=4, side_dwconv=3, auto_pad=False,

              attn_backend='torch'):

    super().__init__()

    # local attention setting

    self.dim = dim

    self.num_heads = num_heads

    assert self.dim % num_heads == 0, 'dim must be divisible by num_heads!'

    self.head_dim = self.dim // self.num_heads

    self.scale = qk_scale or self.dim ** -0.5 # NOTE: to be consistent with old models.

    #####side_dwconv (i.e. LCE in Shunted Transformer)#####

    self.lepe = nn.Conv2d(dim, dim, kernel_size=side_dwconv, stride=1, padding=side_dwconv // 2,

                           groups=dim) if side_dwconv > 0 else \

        lambda x: torch.zeros_like(x)

    ##### regional routing setting #####

    self.topk = topk

    self.n_win = n_win # number of windows per row/col
```

```
#####
```

```
self.qkv_linear = nn.Conv2d(self.dim, 3 * self.dim, kernel_size=1)
```

```
self.output_linear = nn.Conv2d(self.dim, self.dim, kernel_size=1)
```

```
if attn_backend == 'torch':
```

```
    self.attn_fn = regional_routing_attention_torch
```

```
else:
```

```
    raise ValueError('CUDA implementation is not available yet. Please stay tuned.')
```

```
def forward(self, x: Tensor, ret_attn_mask=False):
```

```
    """
```

```
    Args:
```

```
        x: NCHW tensor, better to be channel_last
```

```
(https://pytorch.org/tutorials/intermediate/memory\_format\_tutorial.html)
```

```
    Return:
```

```
        NCHW tensor
```

```
    """
```

```
N, C, H, W = x.size()
```

```
region_size = (H // self.n_win, W // self.n_win)
```

```
# STEP 1: linear projection
```

```
qkv = self.qkv_linear.forward(x) # ncHW
```

```

q, k, v = qkv.chunk(3, dim=1) # nCHW

# STEP 2: region-to-region routing

# NOTE: ceil_mode=True, count_include_pad=False = auto padding

# NOTE: gradients backward through token-to-token attention. See Appendix A for the intuition.

q_r = F.avg_pool2d(q.detach(), kernel_size=region_size, ceil_mode=True, count_include_pad=False)

k_r = F.avg_pool2d(k.detach(), kernel_size=region_size, ceil_mode=True, count_include_pad=False) #
nchw

q_r: Tensor = q_r.permute(0, 2, 3, 1).flatten(1, 2) # n(hw)c

k_r: Tensor = k_r.flatten(2, 3) # nc(hw)

a_r = q_r @ k_r # n(hw)(hw), adj matrix of regional graph

_, idx_r = torch.topk(a_r, k=self.topk, dim=-1) # n(hw)k long tensor

idx_r: LongTensor = idx_r.unsqueeze_(1).expand(-1, self.num_heads, -1, -1)

# STEP 3: token to token attention (non-parametric function)

output, attn_mat = self.attn_fn(query=q, key=k, value=v, scale=self.scale,

                                region_graph=idx_r, region_size=region_size

                                )

output = output + self.lepe(v) # nCHW

output = self.output_linear(output) # nCHW

if ret_attn_mask:

```

```
        return output, attn_mat

    return output

# 输入 N C HW, 输出 N C H W

if __name__ == '__main__':

    block = BiLevelRoutingAttention_nchw(64).cuda()

    input = torch.rand(1, 64, 64, 64).cuda()

    output = block(input)

    print(output.shape)
```

## 33、STVit 模块

论文《Vision Transformer with Super Token Sampling》

### 1、作用

STVit 旨在通过改进视觉 Transformer 的空间-时间效率，解决在处理视频和图像任务时常见的计算冗余问题。该模型尝试减少早期层次捕捉局部特征时的冗余计算，从而减少不必要的计算成本。

### 2、机制

STVit 引入了一种类似于图像处理中“超像素”的概念，称为“超级令牌”（super tokens），以减少自注意力计算中元素的数量，同时保留对全局关系建模的能力。该过程涉及从视觉令牌中采样超级令牌，对这些超级令牌执行自注意力操作，并将它们映射回原始令牌空间。

### 3、独特优势

STViT 在不同的视觉任务中展示了强大的性能，包括图像分类、对象检测和分割，同时拥有更少的参数和较低的计算成本。例如，STViT 在没有额外训练数据的情况下，在 ImageNet-1K 分类任务上达到了 86.4% 的顶级 1 准确率，且参数少于 100M。

### 4、代码

```
import torch

import torch.nn as nn

import torch.nn.functional as F


class Unfold(nn.Module):

    def __init__(self, kernel_size=3):

        super().__init__()

        self.kernel_size = kernel_size

        weights = torch.eye(kernel_size ** 2)

        weights = weights.reshape(kernel_size ** 2, 1, kernel_size, kernel_size)

        self.weights = nn.Parameter(weights, requires_grad=False)

    def forward(self, x):

        b, c, h, w = x.shape
```

```
x = F.conv2d(x.reshape(b * c, 1, h, w), self.weights, stride=1, padding=self.kernel_size // 2)
```

```
return x.reshape(b, c * 9, h * w)
```

```
class Fold(nn.Module):
```

```
def __init__(self, kernel_size=3):
```

```
    super().__init__()
```

```
    self.kernel_size = kernel_size
```

```
    weights = torch.eye(kernel_size ** 2)
```

```
    weights = weights.reshape(kernel_size ** 2, 1, kernel_size, kernel_size)
```

```
    self.weights = nn.Parameter(weights, requires_grad=False)
```

```
def forward(self, x):
```

```
    b, _, h, w = x.shape
```

```
    x = F.conv_transpose2d(x, self.weights, stride=1, padding=self.kernel_size // 2)
```

```
    return x
```

```
class Attention(nn.Module):
```

```
def __init__(self, dim, window_size=None, num_heads=8, qkv_bias=False, qk_scale=None, attn_drop=0.,  
proj_drop=0.):
```

```
super().__init__()
```

```
self.dim = dim
```

```
self.num_heads = num_heads
```

```
head dim = dim // num heads
```

```
self.window_size = window_size
```

```
self.scale = qk_scale or head_dim ** -0.5
```

```
self.qkv = nn.Conv2d(dim, dim * 3, 1, bias=qkv bias)
```

```
self.attn_drop = nn.Dropout(attn_drop)
```

```
self.proj = nn.Conv2d(dim, dim, 1)
```

```
self.proj_drop = nn.Dropout(proj_drop)
```

```
def forward(self, x):
```

```
B, C, H, W = x.shape
```

$$N = H * W$$

```
q, k, v = self.qkv(x).reshape(B, self.num_heads, C // self.num_heads * 3, N).chunk(3,
```

```
dim=2) # (B, num_heads,
head dim, N)
```

```
attn = (k.transpose(-1, -2) @ q) * self.scale
```

```
attn = attn.softmax(dim=-2) # (B, h, N, N)
```

```
attn = self.attn_drop(attn)
```

```
x = (v @ attn).reshape(B, C, H, W)
```

```
x = self.proj(x)
```

```
x = self.proj_drop(x)
```

```
return x
```

```
class StokenAttention(nn.Module):
```

```
    def __init__(self, dim, stoken_size, n_iter=1, num_heads=8, qkv_bias=False, qk_scale=None, attn_drop=0.,
```

```
               proj_drop=0.):
```

```
        super().__init__()
```

```
        self.n_iter = n_iter
```

```
        self.stoken_size = stoken_size
```

```
        self.scale = dim ** - 0.5
```



```
self.unfold = Unfold(3)
```

```
self.fold = Fold(3)
```

```
self.stoken_refine = Attention(dim, num_heads=num_heads, qkv_bias=qkv_bias, qk_scale=qk_scale,
```

```
attn_drop=attn_drop, proj_drop=proj_drop)
```

```
def stoken_forward(self, x):
```

```
'''
```

```
x: (B, C, H, W)
```

```
'''
```

```
B, C, H0, W0 = x.shape
```

```
h, w = self.stoken_size
```

```
pad_l = pad_t = 0
```

```
pad_r = (w - W0 % w) % w
```

```
pad_b = (h - H0 % h) % h
```

```
if pad_r > 0 or pad_b > 0:
```

```
x = F.pad(x, (pad_l, pad_r, pad_t, pad_b))
```

```
_, _, H, W = x.shape
```

```
hh, ww = H // h, W // w
```

```
token_features = F.adaptive_avg_pool2d(x, (hh, ww)) # (B, C, hh, ww)
```

```
pixel_features = x.reshape(B, C, hh, h, ww, w).permute(0, 2, 4, 3, 5, 1).reshape(B, hh * ww, h * w, C)
```

```
with torch.no_grad():
```

```
    for idx in range(self.n_iter):
```

```
        token_features = self.unfold(token_features) # (B, C*9, hh*ww)
```

```
        token_features = token_features.transpose(1, 2).reshape(B, hh * ww, C, 9)
```

```
        affinity_matrix = pixel_features @ token_features * self.scale # (B, hh*ww, h*w, 9)
```

```
        affinity_matrix = affinity_matrix.softmax(-1) # (B, hh*ww, h*w, 9)
```

```
        affinity_matrix_sum = affinity_matrix.sum(2).transpose(1, 2).reshape(B, 9, hh, ww)
```

```
        affinity_matrix_sum = self.fold(affinity_matrix_sum)
```

```
        if idx < self.n_iter - 1:
```

```
            token_features = pixel_features.transpose(-1, -2) @ affinity_matrix # (B, hh*ww, C, 9)
```

```
            token_features = self.fold(token_features.permute(0, 2, 3, 1).reshape(B * C, 9, hh,  
ww)).reshape(
```

```
            B, C, hh, ww)
```

```
token_features = token_features / (affinity_matrix_sum + 1e-12) # (B, C, hh, ww)
```

```
token_features = pixel_features.transpose(-1, -2) @ affinity_matrix # (B, hh*ww, C, 9)
```

```
token_features = self.fold(token_features.permute(0, 2, 3, 1).reshape(B * C, 9, hh, ww)).reshape(B, C, hh, ww)
```

```
token_features = token_features / (affinity_matrix_sum.detach() + 1e-12) # (B, C, hh, ww)
```

```
token_features = self.token_refine(token_features)
```

```
token_features = self.unfold(token_features) # (B, C*9, hh*ww)
```

```
token_features = token_features.transpose(1, 2).reshape(B, hh * ww, C, 9) # (B, hh*ww, C, 9)
```

```
pixel_features = token_features @ affinity_matrix.transpose(-1, -2) # (B, hh*ww, C, h*w)
```

```
pixel_features = pixel_features.reshape(B, hh, ww, C, h, w).permute(0, 3, 1, 4, 2, 5).reshape(B, C, H, W)
```

```
if pad_r > 0 or pad_b > 0:
```

```
pixel_features = pixel_features[:, :, :H0, :W0]
```

```
return pixel_features
```

```

def direct_forward(self, x):

    B, C, H, W = x.shape

    token_features = x

    token_features = self.token_refine(token_features)

    return token_features


def forward(self, x):

    if self.token_size[0] > 1 or self.token_size[1] > 1:

        return self.token_forward(x)

    else:

        return self.direct_forward(x)


# 输入 N C H W, 输出 N C H W

if __name__ == '__main__':

    input = torch.randn(3, 64, 64, 64).cuda()

    se = TokenAttention(64, token_size=[8,8]).cuda()

    output = se(input)

    print(output.shape)

```

## 34、EMO 模块

## 1、作用

本文提出了一种有效的轻量级模型设计方法，旨在开发现代高效的轻量级模型，用于密集预测任务，同时平衡参数、FLOPs 和性能。作者通过重新思考高效的 Inverted Residual Block (IRB) 和 Transformer 的有效组件，从统一的视角出发,扩展了基于 CNN 的 IRB 到基于 Meta attention 的模型,并抽象出了一种一次残差的 Meta Mobile Block (MMB)，用于轻量级模型设计。

## 2、机制

本研究通过简单但有效的设计准则，提出了一种现代的 Inverted Residual Mobile Block (iRMB)，并使用 iRMB 构建了一个类似于 ResNet 的高效模型 (EMO)，仅用于下游任务。EMO 通过将 CNN 的效率和 Transformer 的动态建模能力结合在 iRMB 中，有效地提高了模型性能。同时，EMO 在不引入复杂结构的情况下，实现了与当前最先进的轻量级注意力模型的竞争性能。

## 3、独特优势

EMO 在 ImageNet-1K、COCO2017 和 ADE20K 基准上的广泛实验展示了其优越性，例如，EMO-1M/2M/5M 分别达到了 71.5%、75.1%和 78.4%的 Top-1 准确率，超过了同等级别的 CNN-/Attention-based 模型。同时，在参数效率和准确性之间取得了良好的平衡：在 iPhone14 上运行速度比 EdgeNeXt 快 2.8-4.0 倍。此外，EMO 不使用复杂的操作，但仍然在多个视觉任务中获得了非常竞争性的结果，这证明了其作为轻量级注意力模型的有效性和实用性。

## 4、代码

```
import math

from functools import partial

import torch
```

```
from timm.models.efficientnet_blocks import SqueezeExcite as SE
```

```
from einops import rearrange, reduce
```

```
from timm.models.layers.activations import *
```

```
from timm.models.layers import DropPath
```

```
inplace = True
```

```
# ===== For Common =====
```

```
class LayerNorm2d(nn.Module):
```

```
    def __init__(self, normalized_shape, eps=1e-6, elementwise_affine=True):
```

```
        super().__init__()
```

```
        self.norm = nn.LayerNorm(normalized_shape, eps, elementwise_affine)
```

```
    def forward(self, x):
```

```
        x = rearrange(x, 'b c h w -> b h w c').contiguous()
```

```
        x = self.norm(x)
```

```
        x = rearrange(x, 'b h w c -> b c h w').contiguous()
```

```
        return x
```

```
def get_norm(norm_layer='in_1d'):
```

```
    eps = 1e-6
```

```
    norm_dict = {
```

```
        'none': nn.Identity,
```

```
        'in_1d': partial(nn.InstanceNorm1d, eps=eps),
```

```
        'in_2d': partial(nn.InstanceNorm2d, eps=eps),
```

```
        'in_3d': partial(nn.InstanceNorm3d, eps=eps),
```

```
        'bn_1d': partial(nn.BatchNorm1d, eps=eps),
```

```
        'bn_2d': partial(nn.BatchNorm2d, eps=eps),
```

```
        # 'bn_2d': partial(nn.SyncBatchNorm, eps=eps),
```

```
        'bn_3d': partial(nn.BatchNorm3d, eps=eps),
```

```
        'gn': partial(nn.GroupNorm, eps=eps),
```

```
        'ln_1d': partial(nn.LayerNorm, eps=eps),
```

```
        'ln_2d': partial(LayerNorm2d, eps=eps),
```

```
    }
```

```
    return norm_dict[norm_layer]
```

```
def get_act(act_layer='relu'):
```

```
    act_dict = {
```

```
        'none': nn.Identity,
```

```
        'sigmoid': Sigmoid,
```

```
'swish': Swish,
```

```
'mish': Mish,
```

```
'hsigmoid': HardSigmoid,
```

```
'hswish': HardSwish,
```

```
'hmish': HardMish,
```

```
'tanh': Tanh,
```

```
'relu': nn.ReLU,
```

```
'relu6': nn.ReLU6,
```

```
'prelu': PReLU,
```

```
'gelu': GELU,
```

```
'silu': nn.SiLU
```

```
}
```

```
return act_dict[act_layer]
```

```
class LayerScale(nn.Module):
```

```
    def __init__(self, dim, init_values=1e-5, inplace=True):
```

```
        super().__init__()
```

```
        self.inplace = inplace
```

```
        self.gamma = nn.Parameter(init_values * torch.ones(1, 1, dim))
```

```
    def forward(self, x):
```



```
return x.mul_(self.gamma) if self.inplace else x * self.gamma
```

```
class LayerScale2D(nn.Module):
```

```
    def __init__(self, dim, init_values=1e-5, inplace=True):
```

```
        super().__init__()
```

```
        self.inplace = inplace
```

```
        self.gamma = nn.Parameter(init_values * torch.ones(1, dim, 1, 1))
```

```
    def forward(self, x):
```

```
        return x.mul_(self.gamma) if self.inplace else x * self.gamma
```

```
class ConvNormAct(nn.Module):
```

```
    def __init__(self, dim_in, dim_out, kernel_size, stride=1, dilation=1, groups=1, bias=False,
```

```
                skip=False, norm_layer='bn_2d', act_layer='relu', inplace=True, drop_path_rate=0.):
```

```
        super(ConvNormAct, self).__init__()
```

```
        self.has_skip = skip and dim_in == dim_out
```

```
        padding = math.ceil((kernel_size - stride) / 2)
```

```
        self.conv = nn.Conv2d(dim_in, dim_out, kernel_size, stride, padding, dilation, groups, bias)
```

```
        self.norm = get_norm(norm_layer)(dim_out)
```

```
self.act = get_act(act_layer)(inplace=inplace)
```

```
self.drop_path = DropPath(drop_path_rate) if drop_path_rate else nn.Identity()
```

```
def forward(self, x):
```

```
    shortcut = x
```

```
    x = self.conv(x)
```

```
    x = self.norm(x)
```

```
    x = self.act(x)
```

```
    if self.has_skip:
```

```
        x = self.drop_path(x) + shortcut
```

```
    return x
```

```
# ===== Multi-Scale Populations, for down-sampling and inductive bias =====
```

```
class MSPatchEmb(nn.Module):
```

```
    def __init__(self, dim_in, emb_dim, kernel_size=2, c_group=-1, stride=1, dilations=[1, 2, 3],
```

```
                norm_layer='bn_2d', act_layer='silu'):
```

```
        super().__init__()
```

```
        self.dilation_num = len(dilations)
```

```
        assert dim_in % c_group == 0
```

```
        c_group = math.gcd(dim_in, emb_dim) if c_group == -1 else c_group
```

```
self.convs = nn.ModuleList()
```

```
for i in range(len(dilations)):
```

```
padding = math.ceil(((kernel_size - 1) * dilations[i] + 1 - stride) / 2)
```

```
self.convs.append(nn.Sequential(
```

```
nn.Conv2d(dim_in, emb_dim, kernel_size, stride, padding, dilations[i], groups=c_group),
```

```
get_norm(norm_layer)(emb_dim),
```

```
get_act(act_layer)(emb_dim)))
```

```
def forward(self, x):
```

```
if self.dilation_num == 1:
```

```
x = self.convs[0](x)
```

```
else:
```

```
x = torch.cat([self.convs[i](x).unsqueeze(dim=-1) for i in range(self.dilation_num)], dim=-1)
```

```
x = reduce(x, 'b c h w n -> b c h w', 'mean').contiguous()
```

```
return x
```

```
class iRMB(nn.Module):
```

```
def __init__(self, dim_in, dim_out, norm_in=True, has_skip=True, exp_ratio=1.0, norm_layer='bn_2d',
```

```
act_layer='relu', v_proj=True, dw_ks=3, stride=1, dilation=1, se_ratio=0.0, dim_head=64, window_size=7,
```

```
attn_s=True, qkv_bias=False, attn_drop=0., drop=0., drop_path=0., v_group=False, attn_pre=False):
```

```
super().__init__()
```

```
self.norm = get_norm(norm_layer)(dim_in) if norm_in else nn.Identity()
```

```
dim_mid = int(dim_in * exp_ratio)
```

```
self.has_skip = (dim_in == dim_out and stride == 1) and has_skip
```

```
self.attn_s = attn_s
```

```
if self.attn_s:
```

```
assert dim_in % dim_head == 0, 'dim should be divisible by num_heads'
```

```
self.dim_head = dim_head
```

```
self.window_size = window_size
```

```
self.num_head = dim_in // dim_head
```

```
self.scale = self.dim_head ** -0.5
```

```
self.attn_pre = attn_pre
```

```
self.qk = ConvNormAct(dim_in, int(dim_in * 2), kernel_size=1, bias=qkv_bias, norm_layer='none',
```

```
act_layer='none')
```

```
self.v = ConvNormAct(dim_in, dim_mid, kernel_size=1, groups=self.num_head if v_group else 1, bias=qkv_bias,
```

```
norm_layer='none', act_layer=act_layer, inplace=inplace)
```

```
self.attn_drop = nn.Dropout(attn_drop)
```

```
else:
```

```
if v_proj:
```

```
self.v = ConvNormAct(dim_in, dim_mid, kernel_size=1, bias=qkv_bias, norm_layer='none',
```

```
act_layer=act_layer, inplace=inplace)
```

```
else:
```

```
self.v = nn.Identity()
```

```
self.conv_local = ConvNormAct(dim_mid, dim_mid, kernel_size=dw_ks, stride=stride, dilation=dilation,
```

```
groups=dim_mid, norm_layer='bn_2d', act_layer='silu', inplace=inplace)
```

```
self.se = SE(dim_mid, rd_ratio=se_ratio, act_layer=get_act(act_layer)) if se_ratio > 0.0 else nn.Identity()
```

```
self.proj_drop = nn.Dropout(drop)
```

```
self.proj = ConvNormAct(dim_mid, dim_out, kernel_size=1, norm_layer='none', act_layer='none', inplace=inplace)
```

```
self.drop_path = DropPath(drop_path) if drop_path else nn.Identity()
```

```
def forward(self, x):
```

```
shortcut = x
```

```
x = self.norm(x)
```

```
B, C, H, W = x.shape
```

```
if self.attn_s:
```

```
# padding
```

```
if self.window_size <= 0:
```

```
window_size_W, window_size_H = W, H
```

```
else:
```

```
window_size_W, window_size_H = self.window_size, self.window_size
```

```
pad_l, pad_t = 0, 0
```

```
pad_r = (window_size_W - W % window_size_W) % window_size_W
```

```
pad_b = (window_size_H - H % window_size_H) % window_size_H
```

```
x = F.pad(x, (pad_l, pad_r, pad_t, pad_b, 0, 0,))
```

```
n1, n2 = (H + pad_b) // window_size_H, (W + pad_r) // window_size_W
```

```
x = rearrange(x, 'b c (h1 n1) (w1 n2) -> (b n1 n2) c h1 w1', n1=n1, n2=n2).contiguous()
```

```
# attention
```

```
b, c, h, w = x.shape
```

```
qk = self.qk(x)
```

```
qk = rearrange(qk, 'b (qk heads dim_head) h w -> qk b heads (h w) dim_head', qk=2, heads=self.num_head,
```

```
dim_head=self.dim_head).contiguous()
```

```
q, k = qk[0], qk[1]
```

```
attn_spa = (q @ k.transpose(-2, -1)) * self.scale
```

```
attn_spa = attn_spa.softmax(dim=-1)
```

```
attn_spa = self.attn_drop(attn_spa)
```

```
if self.attn_pre:
```

```
x = rearrange(x, 'b (heads dim_head) h w -> b heads (h w) dim_head', heads=self.num_head).contiguous()
```

```
x_spa = attn_spa @ x
```

```
x_spa = rearrange(x_spa, 'b heads (h w) dim_head -> b (heads dim_head) h w', heads=self.num_head, h=h,
```

```
w=w).contiguous()
```

```
x_spa = self.v(x_spa)
```

```
else:
```

```
v = self.v(x)
```

```
v = rearrange(v, 'b (heads dim_head) h w -> b heads (h w) dim_head', heads=self.num_head).contiguous()
```

```
x_spa = attn_spa @ v
```

```
x_spa = rearrange(x_spa, 'b heads (h w) dim_head -> b (heads dim_head) h w', heads=self.num_head, h=h,
```

```
w=w).contiguous()
```

```
# unpadding
```

```
x = rearrange(x_spa, '(b n1 n2) c h1 w1 -> b c (h1 n1) (w1 n2)', n1=n1, n2=n2).contiguous()
```

```
if pad_r > 0 or pad_b > 0:
```

```
x = x[:, :, :H, :W].contiguous()
```

```
else:
```

```
x = self.v(x)
```

```
x = x + self.se(self.conv_local(x)) if self.has_skip else self.se(self.conv_local(x))
```

```
x = self.proj_drop(x)
```

```
x = self.proj(x)
```

```
x = (shortcut + self.drop_path(x)) if self.has_skip else x
```

```
return x
```

```
# 输入 N C H W, 输出 N C H W
```

```
if __name__ == '__main__':
```

```
input = torch.randn(3, 64, 64, 64).cuda()
```

```
model = iRMB(64, 64).cuda()
```

```
output = model(input)
```

```
print(output.shape)
```

# 35、AFT 模块

论文《An Attention Free Transformer》

## 1、作用

注意力自由变换器 (AFT) 旨在通过去除传统 Transformer 中的点积自注意力机制，提供一种更高效的变换器模型。它特别适用于需要高计算效率和较低内存消耗的应用场景，如移动设备和边缘计算。

## 2、机制

AFT 通过直接对输入特征进行变换来实现序列间的关联，不再需要复杂的自注意力计算。它使用一种简单的基于位置的加权策略，通过这种方式，每个输出元素是输入元素的加权和，权重由元素的相对位置决定。这种方法极大地降低了模型的复杂性和运行时内存需求。

## 3、独特优势

- 1、高效性：AFT 由于避免了昂贵的自注意力计算，因此在执行速度和计算效率上有明显优势。
- 2、简化模型结构：通过消除自注意力机制，AFT 简化了模型结构，使得模型更加轻量化，易于实现和部署。
- 3、适应性强：AFT 的结构使其更容易适应于不同的任务和数据集，具有良好的泛化能力。
- 4、资源占用低：对于资源受限的环境，如移动设备和边缘计算设备，AFT 提供了一种实用的解决方案，能够在保持较高性能的同时，降低资源消耗。

## 4、代码

```
import numpy as np
```



```
import torch

from torch import nn

from torch.nn import init


class AFT_FULL(nn.Module):

    # 初始化 AFT_FULL 模块

    def __init__(self, d_model, n=49, simple=False):

        super(AFT_FULL, self).__init__()

        # 定义 QKV 三个线性变换层

        self.fc_q = nn.Linear(d_model, d_model)

        self.fc_k = nn.Linear(d_model, d_model)

        self.fc_v = nn.Linear(d_model, d_model)

        # 根据 simple 参数决定位置偏置的初始化方式

        if (simple):

            self.position_biases = torch.zeros((n, n)) # 简单模式下为零矩阵

        else:

            self.position_biases = nn.Parameter(torch.ones((n, n))) # 非简单模式下为可学习的参数

        self.d_model = d_model

        self.n = n # 输入序列的长度

        self.sigmoid = nn.Sigmoid() # 使用 Sigmoid 函数

        self.init_weights() # 初始化模型权重
```

```
def init_weights(self):
```

```
    # 对模块中的参数进行初始化
```

```
    for m in self.modules():
```

```
        if isinstance(m, nn.Conv2d):
```

```
            init.kaiming_normal_(m.weight, mode='fan_out')
```

```
            if m.bias is not None:
```

```
                init.constant_(m.bias, 0)
```

```
            elif isinstance(m, nn.BatchNorm2d):
```

```
                init.constant_(m.weight, 1)
```

```
                init.constant_(m.bias, 0)
```

```
            elif isinstance(m, nn.Linear):
```

```
                init.normal_(m.weight, std=0.001)
```

```
                if m.bias is not None:
```

```
                    init.constant_(m.bias, 0)
```

```
def forward(self, input):
```

```
    bs, n, dim = input.shape # 输入的批大小、序列长度和特征维度
```

```
    # 通过 QKV 变换生成查询、键和值
```

```
    q = self.fc_q(input) # bs,n,dim
```

```
    k = self.fc_k(input).view(1, bs, n, dim) # 1,bs,n,dim, 为了后续运算方便
```

```
v = self.fc_v(input).view(1, bs, n, dim) # 1,bs,n,dim
```

```
# 使用位置偏置和键值对进行加权求和
```

```
numerator = torch.sum(torch.exp(k + self.position_biases.view(n, 1, -1, 1)) * v, dim=2) # n,bs,dim
```

```
denominator = torch.sum(torch.exp(k + self.position_biases.view(n, 1, -1, 1)), dim=2) # n,bs,dim
```

```
# 计算加权求和的结果，并通过 sigmoid 函数调制查询向量
```

```
out = (numerator / denominator) # n,bs,dim
```

```
out = self.sigmoid(q) * (out.permute(1, 0, 2)) # bs,n,dim, 最后将结果重新排列
```

```
return out
```

```
# 示例使用
```

```
if __name__ == '__main__':
```

```
    block = AFT_FULLL(d_model=512, n=64).cuda()
```

```
    input = torch.rand(64, 64, 512).cuda()
```

```
    output = block(input)
```

```
    print(output.shape)
```

## 36、CrissCrossAttention 模块

论文《CROSSFORMER: A VERSATILE VISION TRANSFORMER HINGING ON CROSS-SCALE ATTENTION》

## 1、作用

CrossFormer 通过跨尺度的特征提取和注意力机制，有效处理计算机视觉任务。它克服了现有视觉 Transformer 无法在不同尺度间建立有效交互的限制，提升了模型对图像的理解能力。

## 2、机制

### 1、跨尺度嵌入层（CEL）：

通过不同尺度的内核采样图像补丁并将它们合并，为自注意力模块提供了跨尺度特征。

### 2、长短距离注意力（LSDA）：

将自注意力模块分为短距离注意力（SDA）和长距离注意力（LDA）两部分，既降低了计算成本，又保留了不同尺度的特征。

### 3、动态位置偏差（DPB）：

提出了一种动态位置偏差模块，使得相对位置偏差可以应用于不同大小的图像，提高了模型的灵活性和适用性。

## 3、独特优势

1、跨尺度交互：CrossFormer 通过 CEL 和 LSDA 实现了特征在不同尺度间的有效交互，这对于处理具有不同尺寸对象的图像至关重要。

2、灵活性和适用性：通过动态位置偏差模块，CrossFormer 能够适应不同尺寸的输入图像，提高了模型在各种视觉任务上的适用性。

3、优异的性能：广泛的实验表明，CrossFormer 在图像分类、对象检测、实例分割和语义分割等任务上超越了其他最先进的视觉 Transformer 模型。

## 4、代码

```
import torch

import torch.nn as nn

from torch.nn import Softmax
```

# 定义一个无限小的矩阵，用于在注意力矩阵中屏蔽特定位置

```
def INF(B, H, W):
```

```
    return -torch.diag(torch.tensor(float("inf")).repeat(H), 0).unsqueeze(0).repeat(B * W, 1, 1)
```

```
class CrissCrossAttention(nn.Module):
```

```
    """ Criss-Cross Attention Module """
```

```
    def __init__(self, in_dim):
```

```
        super(CrissCrossAttention, self).__init__()
```

```
        # Q, K, V 转换层
```

```
        self.query_conv = nn.Conv2d(in_channels=in_dim, out_channels=in_dim // 8, kernel_size=1)
```

```
        self.key_conv = nn.Conv2d(in_channels=in_dim, out_channels=in_dim // 8, kernel_size=1)
```

```
        self.value_conv = nn.Conv2d(in_channels=in_dim, out_channels=in_dim, kernel_size=1)
```

```
        # 使用 softmax 对注意力分数进行归一化
```

```
        self.softmax = Softmax(dim=3)
```

```
        self.INF = INF
```

```
        # 学习一个缩放参数，用于调节注意力的影响
```

```
        self.gamma = nn.Parameter(torch.zeros(1))
```

```
    def forward(self, x):
```

```
        m_batchsize, _, height, width = x.size()
```

```
        # 计算查询(Q)、键(K)、值(V)矩阵
```

```

proj_query = self.query_conv(x)

proj_query_H = proj_query.permute(0, 3, 1, 2).contiguous().view(m_batchsize * width, -1,
height).permute(0, 2, 1)

proj_query_W = proj_query.permute(0, 2, 1, 3).contiguous().view(m_batchsize * height, -1,
width).permute(0, 2, 1)

proj_key = self.key_conv(x)

proj_key_H = proj_key.permute(0, 3, 1, 2).contiguous().view(m_batchsize * width, -1, height)

proj_key_W = proj_key.permute(0, 2, 1, 3).contiguous().view(m_batchsize * height, -1, width)

proj_value = self.value_conv(x)

proj_value_H = proj_value.permute(0, 3, 1, 2).contiguous().view(m_batchsize * width, -1, height)

proj_value_W = proj_value.permute(0, 2, 1, 3).contiguous().view(m_batchsize * height, -1, width)

# 计算垂直和水平方向上的注意力分数，并应用无穷小掩码屏蔽自注意

energy_H = (torch.bmm(proj_query_H, proj_key_H) + self.INF(m_batchsize, height,
width)).view(m_batchsize, width, height, height).permute(0, 2, 1, 3)

energy_W = torch.bmm(proj_query_W, proj_key_W).view(m_batchsize, height, width, width)

# 在垂直和水平方向上应用 softmax 归一化

concat = self.softmax(torch.cat([energy_H, energy_W], 3))

# 分离垂直和水平方向上的注意力，应用到值(V)矩阵上

```

```

        att_H = concatenate[:, :, :, 0:height].permute(0, 2, 1, 3).contiguous().view(m_batchsize * width, height,
height)

        att_W = concatenate[:, :, :, height:height + width].contiguous().view(m_batchsize * height, width, width)

        # 计算最终的输出，加上输入 x 以应用残差连接

        out_H = torch.bmm(proj_value_H, att_H.permute(0, 2, 1)).view(m_batchsize, width, -1, height).permute(0,
2, 3, 1)

        out_W = torch.bmm(proj_value_W, att_W.permute(0, 2, 1)).view(m_batchsize, height, -1, width).permute(0,
2, 1, 3)

        return self.gamma * (out_H + out_W) + x

if __name__ == '__main__':

    block = CrissCrossAttention(64)

    input = torch.rand(1, 64, 64, 64)

    output = block(input)

    print( output.shape)

```

## 37、A2Attention 模块

论文《A2-Nets: Double Attention Networks》

### 1、作用

A2-Nets 通过引入双重注意力机制，有效地捕获长距离特征依赖，提高图像和视频识别任务的性能。它允许卷积层直接感知整个时空空间的特征，而无需通过增加深度来扩大感受野，从而提高了模型的效率和性能。

## 2、机制

A2-Nets 设计了一个双重注意力模块（A2-Block），通过两步注意力机制聚合和分配全局特征。第一步通过第二阶注意力池化从整个空间中选择关键特征形成一个紧凑的集合。第二步则利用另一个注意力机制，根据每个位置的需求适应性地选择和分配关键特征，使得后续的卷积层能够有效地访问整个空间的特征。

## 3、独特优势

- 1、通过捕获第二阶特征统计信息，可以捕获不能通过全局平均池化获得的复杂外观和运动相关性。
- 2、第二个注意力操作使得模型可以从紧凑的特征集合中，根据每个位置的具体需求适应性地分配特征，比全面相关特征的方法更高效。
- 3、A2-Nets 可以被方便地插入到现有的深度神经网络中，对计算和内存资源的额外需求小，同时在多个标准图像和视频识别任务上显著提高性能。

## 4、代码

```
# A2-Nets: Double Attention Networks

import torch

from torch import nn

from torch.nn import init

from torch.nn import functional as F


class DoubleAttention(nn.Module):
```



```
def __init__(self, in_channels, c_m, c_n, reconstruct=True):

    super().__init__()

    self.in_channels = in_channels# 输入通道数

    self.reconstruct = reconstruct # 是否需要重构输出以匹配输入的维度

    self.c_m = c_m # 第一个注意力机制的输出通道数

    self.c_n = c_n # 第二个注意力机制的输出通道数

    # 定义三个 1x1 卷积层，用于生成 A、B 和 V 特征

    self.convA = nn.Conv2d(in_channels, c_m, 1)

    self.convB = nn.Conv2d(in_channels, c_n, 1)

    self.convV = nn.Conv2d(in_channels, c_n, 1)

    # 如果需要重构，定义一个 1x1 卷积层用于输出重构

    if self.reconstruct:

        self.conv_reconstruct = nn.Conv2d(c_m, in_channels, kernel_size=1)

    self.init_weights()

    def init_weights(self):

        # 权重初始化

        for m in self.modules():

            if isinstance(m, nn.Conv2d):

                init.kaiming_normal_(m.weight, mode='fan_out')

                if m.bias is not None:

                    init.constant_(m.bias, 0)
```

```
elif isinstance(m, nn.BatchNorm2d):
```

```
    init.constant_(m.weight, 1)
```

```
    init.constant_(m.bias, 0)
```

```
elif isinstance(m, nn.Linear):
```

```
    init.normal_(m.weight, std=0.001)
```

```
    if m.bias is not None:
```

```
        init.constant_(m.bias, 0)
```

```
def forward(self, x):
```

```
    # 前向传播
```

```
    b, c, h, w = x.shape
```

```
    assert c == self.in_channels # 确保输入通道数与初始化时一致
```

```
    A = self.convA(x) # b,c_m,h,w# 生成 A 特征图
```

```
    B = self.convB(x) # b,c_n,h,w# 生成 B 特征图
```

```
    V = self.convV(x) # b,c_n,h,w# 生成 V 特征图
```

```
    # 将特征图维度调整为方便矩阵乘法的形状
```

```
    tmpA = A.view(b, self.c_m, -1)
```

```
    attention_maps = F.softmax(B.view(b, self.c_n, -1))
```

```
    attention_vectors = F.softmax(V.view(b, self.c_n, -1))
```

```
    # 步骤 1: 特征门控
```

```
    global_descriptors = torch.bmm(tmpA, attention_maps.permute(0, 2, 1)) # b.c_m,c_n
```

```
    # 步骤 2: 特征分配
```

```
tmpZ = global_descriptors.matmul(attention_vectors) # b,c_m,h*w
```

```
tmpZ = tmpZ.view(b, self.c_m, h, w) # b,c_m,h,w
```

```
if self.reconstruct:
```

```
tmpZ = self.conv_reconstruct(tmpZ)# 如果需要，通过重构层调整输出通道数
```

```
return tmpZ
```

```
# 输入 N C H W, 输出 N C H W
```

```
if __name__ == '__main__':
```

```
block = DoubleAttention(64, 128, 128)
```

```
input = torch.rand(1, 64, 64, 64)
```

```
output = block(input)
```

```
print(input.size(), output.size())
```

## 38、fcanet 模块

论文《FcaNet: Frequency Channel Attention Networks》

### 1、作用

FcaNet 通过采用频率通道注意力机制，显著提高了图像识别和分类任务的性能。该网络通过在频率域而不是传统的空间域对特征进行建模，使得网络能够更有效地捕捉到图像的细节和纹理信息。

### 2、机制

### 1、频率分解：

首先，FcaNet 将输入的特征图通过快速傅立叶变换（FFT）转换到频率域，允许网络直接在频率层面进行学习和表示。

### 2、频率通道注意力：

然后，网络通过一个频率通道注意力模块来自动学习和强调更重要的频率成分，从而更有效地聚焦于对分类任务有益的频率特征。

### 3、重构和映射：

最后，通过逆快速傅立叶变换（IFFT）将加权的频率特征映射回空间域，并通过卷积层进一步细化特征，最终输出用于分类的特征表示。

## 3、独特优势

### 1、提高了特征的区分力：

通过直接在频率域对特征进行操作，FcaNet 能够捕捉到细微的纹理变化，这些在空间域中可能不容易区分，从而提高了模型对图像的理解能力。

### 2、效率与性能的平衡：

虽然涉及到频率变换，但 FcaNet 设计了高效的注意力机制和网络结构，保证了计算的高效性，同时在多个标准数据集上达到了优越的性能。

### 3、灵活性和泛化能力：

FcaNet 不仅在图像分类任务上表现出色，还因其对特征的细粒度建模，展现了在跨领域任务（如目标检测和分割）中的泛化潜力。

## 4、代码

```
# Fcanet: Frequency channel attention networks (ICCV 2021)

import math

import torch

from torch import nn
```

```

def get_freq_indices(method):

    assert method in ['top1', 'top2', 'top4', 'top8', 'top16', 'top32',

                      'bot1', 'bot2', 'bot4', 'bot8', 'bot16', 'bot32',

                      'low1', 'low2', 'low4', 'low8', 'low16', 'low32']

    num_freq = int(method[3:])

    if 'top' in method:

        all_top_indices_x = [0, 0, 6, 0, 0, 1, 1, 4, 5, 1, 3, 0, 0,

                             0, 3, 2, 4, 6, 3, 5, 5, 2, 6, 5, 5, 3, 3, 4, 2, 2, 6, 1]

        all_top_indices_y = [0, 1, 0, 5, 2, 0, 2, 0, 0, 6, 0, 4, 6,

                             3, 5, 2, 6, 3, 3, 3, 5, 1, 1, 2, 4, 2, 1, 1, 3, 0, 5, 3]

        mapper_x = all_top_indices_x[:num_freq]

        mapper_y = all_top_indices_y[:num_freq]

    elif 'low' in method:

        all_low_indices_x = [0, 0, 1, 1, 0, 2, 2, 1, 2, 0, 3, 4, 0,

                             1, 3, 0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4, 5, 6, 1, 2, 3, 4]

        all_low_indices_y = [0, 1, 0, 1, 2, 0, 1, 2, 2, 3, 0, 0, 4,

                             3, 1, 5, 4, 3, 2, 1, 0, 6, 5, 4, 3, 2, 1, 0, 6, 5, 4, 3]

        mapper_x = all_low_indices_x[:num_freq]

        mapper_y = all_low_indices_y[:num_freq]

    elif 'bot' in method:

```

```

all_bot_indices_x = [6, 1, 3, 3, 2, 4, 1, 2, 4, 4, 5, 1, 4,
                     6, 2, 5, 6, 1, 6, 2, 2, 4, 3, 3, 5, 5, 6, 2, 5, 5, 3, 6]

all_bot_indices_y = [6, 4, 4, 6, 6, 3, 1, 4, 4, 5, 6, 5, 2,
                     2, 5, 1, 4, 3, 5, 0, 3, 1, 1, 2, 4, 2, 1, 1, 5, 3, 3, 3]

mapper_x = all_bot_indices_x[:num_freq]

mapper_y = all_bot_indices_y[:num_freq]

```

```

else:

```

```

    raise NotImplementedError

```

```

return mapper_x, mapper_y

```

```

class MultiSpectralAttentionLayer(nn.Module):

```

```

    def __init__(self, channel, dct_h, dct_w, reduction=16, freq_sel_method='top16'):

```

```

        super(MultiSpectralAttentionLayer, self).__init__()

```

```

        self.reduction = reduction

```

```

        self.dct_h = dct_h

```

```

        self.dct_w = dct_w

```

```

        mapper_x, mapper_y = get_freq_indices(freq_sel_method)

```

```

        self.num_split = len(mapper_x)

```

```

        mapper_x = [temp_x * (dct_h // 7) for temp_x in mapper_x]

```

```

        mapper_y = [temp_y * (dct_w // 7) for temp_y in mapper_y]

```

```

# make the frequencies in different sizes are identical to a 7x7 frequency space

# eg, (2,2) in 14x14 is identical to (1,1) in 7x7


self.dct_layer = MultiSpectralDCTLayer(

    dct_h, dct_w, mapper_x, mapper_y, channel)

self.fc = nn.Sequential(

    nn.Linear(channel, channel // reduction, bias=False),

    nn.ReLU(),

    nn.Linear(channel // reduction, channel, bias=False),

    nn.Sigmoid()

)

self.avgpool = nn.AdaptiveAvgPool2d((self.dct_h, self.dct_w))


def forward(self, x):

    n, c, h, w = x.shape

    x_pooled = x

    if h != self.dct_h or w != self.dct_w:

        x_pooled = self.avgpool(x)

        # If you have concerns about one-line-change, don't worry. :)

        # In the ImageNet models, this line will never be triggered.

        # This is for compatibility in instance segmentation and object detection.

    y = self.dct_layer(x_pooled)

```

```
y = self.fc(y).view(n, c, 1, 1)
```

```
return x * y.expand_as(x)
```

```
class MultiSpectralDCTLayer(nn.Module):
```

```
    """
```

```
    Generate dct filters
```

```
    """
```

```
def __init__(self, height, width, mapper_x, mapper_y, channel):
```

```
    super(MultiSpectralDCTLayer, self).__init__()
```

```
    assert len(mapper_x) == len(mapper_y)
```

```
    assert channel % len(mapper_x) == 0
```

```
    self.num_freq = len(mapper_x)
```

```
    # fixed DCT init
```

```
    self.weight = self.get_dct_filter(
```

```
        height, width, mapper_x, mapper_y, channel)
```



```

def forward(self, x):

    assert len(x.shape) == 4, 'x must be 4 dimensions, but got ' + \

        str(len(x.shape))

    # n, c, h, w = x.shape

    x = x * self.weight

    result = torch.sum(torch.sum(x, dim=2), dim=2)

    return result


def build_filter(self, pos, freq, POS):

    result = math.cos(math.pi * freq * (pos + 0.5) / POS) / math.sqrt(POS)

    if freq == 0:

        return result

    else:

        return result * math.sqrt(2)


def get_dct_filter(self, tile_size_x, tile_size_y, mapper_x, mapper_y, channel):

    dct_filter = torch.zeros((channel, tile_size_x, tile_size_y))

    c_part = channel // len(mapper_x)

    for i, (u_x, v_y) in enumerate(zip(mapper_x, mapper_y)):

```

```

        for t_x in range(tile_size_x):

            for t_y in range(tile_size_y):

                dct_filter[i * c_part: (i + 1) * c_part, t_x, t_y] = self.build_filter(

                    t_x, u_x, tile_size_x) * self.build_filter(t_y, v_y, tile_size_y)

    return dct_filter

# 输入 N C H W, 输出 N C H W

if __name__ == '__main__':

    block = MultiSpectralAttentionLayer(64, 16, 16)

    input = torch.rand(1, 64, 64, 64)

    output = block(input)

    print(output.shape)

```

## 39、Temporal\_conv 卷积模块

论文《Connecting the Dots: Multivariate Time Series Forecasting with Graph Neural Networks》

### 1、作用

MTGNN 框架旨在通过利用多图卷积网络来预测交通流量，通过考虑空间-时间相关性和交通网络中的动态变化，以提高交通预测的准确性。

## 2、机制

### 1、多图构建：

MTGNN 通过构建多个图来模拟交通网络中的多种关系，包括空间关系和功能关系等，有效捕获交通流的复杂性。

### 2、图卷积网络：

利用图卷积网络处理多图数据，提取空间特征，并通过堆叠多层图卷积层来增强模型的学习能力。

### 3、时间序列建模：

结合时间卷积网络（TCN）来捕获时间序列数据中的长期依赖性，优化交通流量的预测效果。

### 4、自适应图学习：

MTGNN 通过自适应图学习机制动态学习图结构，自动识别并强化重要的空间连接，提高模型对交通网络动态变化的适应性。

## 3、独特优势

### 1、复杂关系建模：

MTGNN 通过构建和学习多个图，能够综合考虑交通网络中的多重复杂关系，提高预测的准确性和鲁棒性。

### 2、动态适应能力：

自适应图学习机制使得 MTGNN 能够动态适应交通网络中的变化，捕获关键空间连接，提升预测性能。

### 3、长期依赖性建模：

结合图卷积网络和时间卷积网络，MTGNN 有效捕获空间特征和时间序列数据中的长期依赖性，优化了交通流量预测。

## 4、代码

```
import torch

import torch.nn as nn

import torch.nn.functional as F
```

```
import numpy as np

import math

# 膨胀卷积模块，包含了膨胀卷积和一个前馈网络

class dilated_inception(nn.Module):

    def __init__(self, cin, cout, dilation_factor, seq_len):

        super(dilated_inception, self).__init__()

        self.tconv = nn.ModuleList() # 用于存储多个膨胀卷积层

        self.padding = 0 # 卷积层的填充值

        self.seq_len = seq_len # 输入序列的长度

        self.kernel_set = [2, 3, 6, 7] # 定义一组不同大小的卷积核

        cout = int(cout / len(self.kernel_set)) # 计算每个卷积层的输出通道数

        for kern in self.kernel_set: # 为每个卷积核大小创建一个膨胀卷积层

            self.tconv.append(nn.Conv2d(cin, cout, (1, kern), dilation=(1, dilation_factor)))

        # 通过线性层调整时间维度的大小

        self.out = nn.Sequential(

            nn.Linear(self.seq_len - dilation_factor * (self.kernel_set[-1] - 1) + self.padding * 2 - 1 + 1,
cin),

            nn.ReLU(),

            nn.Linear(cin, self.seq_len)

        )
```

```
def forward(self, input):
```

```
    x = []
```

```
    for i in range(len(self.kernel_set)):
```

```
        x.append(self.tconv[i](input)) # 对输入应用每个膨胀卷积层
```

```
    for i in range(len(self.kernel_set)):
```

```
        x[i] = x[i][..., -x[-1].size(3):] # 根据最短序列调整序列长度，以确保所有输出具有相同的时间维度长度
```

```
    x = torch.cat(x, dim=1) # 将所有膨胀卷积层的输出沿通道维度拼接
```

```
    x = self.out(x) # 通过前馈网络调整时间序列的长度
```

```
    return x
```

```
# 时间卷积网络，整合了滤波器和门控机制
```

```
class temporal_conv(nn.Module):
```

```
    def __init__(self, cin, cout, dilation_factor, seq_len):
```

```
        super(temporal_conv, self).__init__()
```

```
        self.filter_convs = dilated_inception(cin=cin, cout=cout, dilation_factor=dilation_factor, seq_len=seq_len)
```

```
        self.gated_convs = dilated_inception(cin=cin, cout=cout, dilation_factor=dilation_factor, seq_len=seq_len)
```

```
    def forward(self, X):
```

```
        filter = self.filter_convs(X) # 计算滤波器分支的输出
```

```
filter = torch.tanh(filter) # 对滤波器分支应用 tanh 激活函数
```

```
gate = self.gated_convs(X) # 计算门控分支的输出
```

```
gate = torch.sigmoid(gate) # 对门控分支应用 sigmoid 函数
```

```
out = filter * gate # 将滤波器分支和门控分支的输出相乘，实现门控机制
```

```
return out
```

```
if __name__ == '__main__':
```

```
    X = torch.randn(1, 32, 1, 24) # 示例输入
```

```
    Model = temporal_conv(cin=32, cout=32, dilation_factor=1, seq_len=24) # 实例化模型
```

```
    out = Model(X) # 计算输出
```

```
    print(out.shape) # 打印输出形状
```

## 40、ScConv 卷积模块

论文《SCConv: Spatial and Channel Reconstruction Convolution for Feature Redundancy》

### 1、作用

旨在通过空间和通道重构来减少卷积层中的冗余特征，从而降低计算成本并提高特征表达的代表性。SCConv 包含两个单元：空间重构单元（SRU）和通道重构单元（CRU）。SRU 通过分离和重建方法来抑制空间冗余，而 CRU 采用分割-转换-融合策略来减少通道冗余。此外，SCConv 是一个即插即用的架构单元，可以直接替换各种卷积神经网络中的标准卷积。实验结果表明，嵌入 SCConv 的模型能够在显著降低复杂度和计算成本的同时，实现更好的性能。

### 2、机制

1、空间重构单元（SRU）：通过分离操作将输入特征图分为信息丰富和信息较少的两部分，然后通过重建操作将这两部分的特征图结合起来，以增强特征表达并抑制空间维度上的冗余。

2、通道重构单元（CRU）：采用分割-转换-融合策略，首先将特征图在通道维度上分割成两部分，一部分通过高效的卷积操作进行特征提取，另一部分则直接利用，最后将两部分的特征图通过自适应融合策略合并，以减少通道维度上的冗余并提升特征的代表性。

### 3、独特优势

1、即插即用：

SCConv 可以作为一个模块直接嵌入到现有的卷积神经网络中，替换标准的卷积层，无需调整模型架构。

2、减少冗余：

通过空间和通道重构有效减少了特征图中的冗余信息，降低了计算成本和模型参数量。

3、提高性能：

实验表明，SCConv 不仅减少了模型的复杂度和计算量，还在多个任务上取得了比原始模型更好的性能。

### 4、代码

```
import torch

import torch.nn.functional as F

import torch.nn as nn

# GroupBatchnorm2d 模块是对标准批量归一化的扩展，它将特征通道分组进行归一化。

class GroupBatchnorm2d(nn.Module):

    def __init__(self, c_num: int,

                  group_num: int = 16,

                  eps: float = 1e-10

                  ):

        super(GroupBatchnorm2d, self).__init__()
```

```
assert c_num >= group_num # 确保通道数大于等于分组数
```

```
self.group_num = group_num # 分组数
```

```
self.weight = nn.Parameter(torch.randn(c_num, 1, 1)) # 权重参数
```

```
self.bias = nn.Parameter(torch.zeros(c_num, 1, 1)) # 偏置参数
```

```
self.eps = eps # 防止除以零
```

```
def forward(self, x):
```

```
    # 实现分组批量归一化的前向传播
```

```
    N, C, H, W = x.size()
```

```
    x = x.view(N, self.group_num, -1) # 根据分组数重新排列 x 的形状
```

```
    mean = x.mean(dim=2, keepdim=True) # 计算每组的均值
```

```
    std = x.std(dim=2, keepdim=True) # 计算每组的标准差
```

```
    x = (x - mean) / (std + self.eps) # 归一化
```

```
    x = x.view(N, C, H, W) # 恢复 x 的原始形状
```

```
    return x * self.weight + self.bias # 应用权重和偏置
```

```
# SRU 模块用于抑制空间冗余。它通过分组归一化和一个门控机制实现。
```

```
class SRU(nn.Module):
```

```
    def __init__(self,
```

```
        out_channels: int,
```

```
        group_num: int = 16,
```

```
        gate_treshold: float = 0.5,
```



```
torch_gn: bool = False
```

```
):
```

```
super().__init__()
```

```
# 选择使用 torch 自带的 GroupNorm 还是自定义的 GroupBatchnorm2d
```

```
self.gn = nn.GroupNorm(num_channels=out_channels, num_groups=group_num) if torch_gn else  
GroupBatchnorm2d(
```

```
c_num=out_channels, group_num=group_num)
```

```
self.gate_threshold = gate_threshold # 设置门控阈值
```

```
self.sigomid = nn.Sigmoid() # 使用 Sigmoid 函数作为激活函数
```

```
def forward(self, x):
```

```
# 实现 SRU 的前向传播
```

```
gn_x = self.gn(x) # 应用分组归一化
```

```
w_gamma = self.gn.weight / torch.sum(self.gn.weight) # 根据归一化权重计算重要性权重
```

```
w_gamma = w_gamma.view(1, -1, 1, 1)
```

```
reweights = self.sigomid(gn_x * w_gamma) # 计算重构权重
```

```
# 根据门控阈值，将特征图分为信息丰富和信息较少的两部分
```

```
info_mask = reweights >= self.gate_threshold
```

```
noninfo_mask = reweights < self.gate_threshold
```

```
x_1 = info_mask * gn_x
```

```
x_2 = noninfo_mask * gn_x
```

```
x = self.reconstruct(x_1, x_2) # 重构特征图
```

```
return x
```

```
def reconstruct(self, x_1, x_2):
```

```
    # 实现特征图的重构
```

```
    x_11, x_12 = torch.split(x_1, x_1.size(1) // 2, dim=1) # 将信息丰富的特征图分为两部分
```

```
    x_21, x_22 = torch.split(x_2, x_2.size(1) // 2, dim=1) # 将信息较少的特征图分为两部分
```

```
    return torch.cat([x_11 + x_22, x_12 + x_21], dim=1) # 通过特定方式合并特征图，增强特征表达
```

```
# CRU 模块用于处理通道冗余。它通过一个压缩-卷积-扩展策略来增强特征的代表性。
```

```
class CRU(nn.Module):
```

```
    def __init__(self,
```

```
        op_channel: int,
```

```
        alpha: float = 1 / 2,
```

```
        squeeze_ratio: int = 2,
```

```
        group_size: int = 2,
```

```
        group_kernel_size: int = 3,
```

```
    ):
```

```
        super().__init__()
```

```
        self.up_channel = up_channel = int(alpha * op_channel) # 计算上分支的通道数
```

```
        self.low_channel = low_channel = op_channel - up_channel # 计算下分支的通道数
```

```
        self.squeeze1 = nn.Conv2d(up_channel, up_channel // squeeze_ratio, kernel_size=1, bias=False) # 上分支的压缩层
```

```
        self.squeeze2 = nn.Conv2d(low_channel, low_channel // squeeze_ratio, kernel_size=1, bias=False) # 下分支的压缩层
```

```
# 上分支的卷积层，包括分组卷积和点卷积
```

```
self.GWC = nn.Conv2d(up_channel // squeeze_ratio, op_channel, kernel_size=group_kernel_size, stride=1,
```

```
padding=group_kernel_size // 2, groups=group_size)
```

```
self.PWC1 = nn.Conv2d(up_channel // squeeze_ratio, op_channel, kernel_size=1, bias=False)
```

```
# 下分支的卷积层
```

```
self.PWC2 = nn.Conv2d(low_channel // squeeze_ratio, op_channel - low_channel // squeeze_ratio,  
kernel_size=1,
```

```
bias=False)
```

```
self.advavg = nn.AdaptiveAvgPool2d(1) # 自适应平均池化层
```

```
def forward(self, x):
```

```
# 实现 CRU 的前向传播
```

```
# 将输入特征图分为上下两部分
```

```
up, low = torch.split(x, [self.up_channel, self.low_channel], dim=1)
```

```
up, low = self.squeeze1(up), self.squeeze2(low) # 对上下两部分分别应用压缩层
```

```
# 对上分支应用卷积层
```

```
Y1 = self.GWC(up) + self.PWC1(up)
```

```
# 对下分支应用卷积层，并与压缩后的低分支特征图合并
```

```
Y2 = torch.cat([self.PWC2(low), low], dim=1)
```

```
# 合并上下分支的特征图，并应用自适应平均池化和 softmax 函数
```

```
out = torch.cat([Y1, Y2], dim=1)
```

```
out = F.softmax(self.advavg(out), dim=1) * out
```

```
out1, out2 = torch.split(out, out.size(1) // 2, dim=1) # 将合并后的特征图分为两部分
```

```
return out1 + out2 # 将两部分的特征图相加，得到最终的输出
```

```
# ScConv 模块结合了 SRU 和 CRU 两个子模块，用于同时处理空间和通道冗余。
```

```
class ScConv(nn.Module):
```

```
    def __init__(self,
```

```
        op_channel: int,
```

```
        group_num: int = 4,
```

```
        gate_treshold: float = 0.5,
```

```
        alpha: float = 1 / 2,
```

```
        squeeze_radio: int = 2,
```

```
        group_size: int = 2,
```

```
        group_kernel_size: int = 3,
```

```
    ):
```

```
        super().__init__()
```

```
        self.SRU = SRU(op_channel, # 初始化空间重构单元
```

```
            group_num=group_num,
```

```
            gate_treshold=gate_treshold)
```

```
        self.CRU = CRU(op_channel, # 初始化通道重构单
```

```
            alpha=alpha,
```

```
            squeeze_radio=squeeze_radio,
```

```
            group_size=group_size,
```

```
            group_kernel_size=group_kernel_size)
```

```
def forward(self, x):  
  
    x = self.SRU(x) # 通过 SRU 处理空间冗余  
  
    x = self.CRU(x) # 通过 CRU 处理通道冗余  
  
    return x # 返回处理后的特征图  
  
# 测试 ScConv 模块  
  
if __name__ == '__main__':  
  
    x = torch.randn(1, 32, 16, 16) # 创建一个随机的输入张量  
  
    model = ScConv(32) # 创建一个 ScConv 模块实例  
  
    print(model(x).shape) # 打印 ScConv 处理后的输出张量形状
```

## 41、部分卷积模块

论文《Run, Don't Walk: Chasing Higher FLOPS for Faster Neural Networks》

### 1、作用：

FasterNet 旨在设计快速的神经网络，主要关注减少浮点操作(FLOPs)数量。然而，FLOPs 数量的减少并不一定导致相似级别的延迟降低，主要是由于运算的浮点操作每秒(FLOPS)效率低下。为了实现更快的网络，FasterNet 重新审视了流行的操作符，并证明了这种低 FLOPS 主要是由于运算符特别是深度卷积的频繁内存访问导致的。因此，FasterNet 提出了一种新颖的部分卷积(PConv)，通过减少冗余计算和内存访问来更有效地提取空间特征。基于 PConv，FasterNet 是一个新的神经网络家族，提供了比其他网络更高的运行速度，而不会在各种视觉任务的准确性上妥协。

### 2、机制

部分卷积(PConv)的设计利用了特征图内部的冗余，系统地只在输入通道的一部分上应用常规卷积(Conv)，而保留其余通道不变。这种操作有两个主要优势：首先，与常规卷积相比，PConv 具有更低的 FLOPs；其次，与深度卷积/组卷积相比，PConv 拥有更高的 FLOPs，这意味着 PConv 更好地利用了设备上的计算能力。此外，为了充分地利用所有通道的信息，FasterNet 在 PConv 之后进一步附加了逐点卷积(PWConv)。这两者的有效接受场类似于一个 T 形的卷积，更加关注中心位置，与常规卷积在补丁上的均匀处理相比，这提供了一个集中于中心位置的计算视角。

### 3、独特优势

FasterNet 通过 PConv 实现了显著的运行速度提升，同时保持了准确性。例如，在 ImageNet-1k 上，FasterNet 的一个小型版本比 MobileViT-XXS 在 GPU、CPU 和 ARM 处理器上分别快 2.8 倍、3.3 倍和 2.4 倍，同时准确率提高了 2.9%。其大型版本 FasterNet-L 在 GPU 上的推理吞吐量比 Swin-B 高 36%，在 CPU 上节省了 37%的计算时间，同时实现了与 Swin-B 相当的 83.5%的 top-1 准确率。这些成就展示了简单而有效的神经网络设计的可能性，不仅限于学术研究，也有潜力直接影响工业界和社区。

### 4、代码

```
import os

import sys

import inspect


from torch import nn

import torch


class Partial_conv3(nn.Module):

    def __init__(self, dim, n_div, forward):

        """

        初始化函数

        :param dim: 输入通道的维度
```

```
:param n_div: 输入维度划分的份数，用于确定哪一部分通道会应用卷积
```

```
:param forward: 指定前向传播的模式，'slicing' 或 'split_cat'
```

```
"""
```

```
super().__init__()
```

```
self.dim_conv3 = dim // n_div # 应用卷积的通道数
```

```
self.dim_untouched = dim - self.dim_conv3 # 保持不变的通道数
```

```
self.partial_conv3 = nn.Conv2d(self.dim_conv3, self.dim_conv3, 3, 1, 1, bias=False) # 部分应用的 3x3  
卷积
```

```
# 根据 forward 参数，选择前向传播的方式
```

```
if forward == 'slicing':
```

```
    self.forward = self.forward_slicing
```

```
elif forward == 'split_cat':
```

```
    self.forward = self.forward_split_cat
```

```
else:
```

```
    raise NotImplementedError
```

```
def forward_slicing(self, x):
```

```
    """
```

```
    利用 slicing 方法的前向传播，主要用于推理
```

```
:param x: 输入特征图
```

```
:return: 输出特征图，部分通道被卷积处理
```

```
    """
```

```
x = x.clone() # 克隆输入以保持原输入不变，用于后续的残差连接
```

```
x[:, :self.dim_conv3, :, :] = self.partial_conv3(x[:, :self.dim_conv3, :, :])
```

```
return x
```

```
def forward_split_cat(self, x):
```

```
    """
```

利用 `split` 和 `cat` 方法的前向传播，可用于训练/推理

:param x: 输入特征图

:return: 输出特征图，部分通道被卷积处理，剩余通道保持不变

```
    """
```

```
x1, x2 = torch.split(x, [self.dim_conv3, self.dim_untouched], dim=1) # 将输入特征图分为两部分
```

```
x1 = self.partial_conv3(x1) # 对第一部分应用卷积
```

```
x = torch.cat((x1, x2), 1) # 将处理后的第一部分和未处理的第二部分拼接
```

```
return x
```

```
if __name__ == '__main__':
```

```
    block = Partial_conv3(64, 2, 'split_cat').cuda() # 实例化模型
```

```
    input = torch.rand(1, 64, 64, 64).cuda() # 创建输入张量
```

```
    output = block(input) # 执行前向传播
```

```
    print(output.shape) # 输出的尺寸
```

## 42、空洞卷积模块



论文《MULTI-SCALE CONTEXT AGGREGATION BY DILATED CONVOLUTIONS》由于它较多用于 ASPP 模块中。所以代码可以借鉴 ASPP

## 1、作用

膨胀卷积旨在通过扩大卷积核的感受野而不增加参数数量或计算量，提高模型对多尺度上下文信息的捕获能力。这在处理图像和语音信号时特别有效，能够在细节和全局信息间建立更好的平衡。

## 2、机制

### 1、膨胀卷积（Dilated Convolution）：

通过在标准卷积核的元素间插入空白（即“膨胀”卷积核），无需增加额外的计算负担即可扩大感受野。

### 2、多尺度上下文聚合：

通过组合不同膨胀率的膨胀卷积，模型能够同时考虑多种尺度的上下文信息，提高特征的表示能力。

### 3、效率与有效性的平衡：

与增加卷积层或扩大卷积核尺寸相比，膨胀卷积在提升感受野的同时，保持了模型的参数数量和计算效率。

## 独特优势

### 1、增强的多尺度信息处理能力：

膨胀卷积通过灵活地调整膨胀率，允许网络在不同层次捕捉到从细粒度到粗粒度的信息，增强了对不同尺度特征的捕获能力。

### 2、保持参数高效性：

在扩大感受野的同时，膨胀卷积不增加额外的参数，使得模型在增加表达能力的同时保持了高效性。

### 3、广泛的应用场景：

膨胀卷积的这些优点使其在图像分割、语音识别、自然语言处理等多个领域中得到了广泛的应用。

## 4、代码

```
from torch import nn
```

```
import torch
```

```
import torch.nn.functional as F
```

```
# 定义一个包含空洞卷积、批量归一化和 ReLU 激活函数的子模块
```

```
class ASPPConv(nn.Sequential):
```

```
    def __init__(self, in_channels, out_channels, dilation):
```

```
        modules = [
```

```
            # 空洞卷积，通过调整 dilation 参数来捕获不同尺度的信息
```

```
            nn.Conv2d(in_channels, out_channels, 3, padding=dilation, dilation=dilation, bias=False),
```

```
            nn.BatchNorm2d(out_channels), # 批量归一化
```

```
            nn.ReLU() # ReLU 激活函数
```

```
        ]
```

```
        super(ASPPConv, self).__init__(*modules)
```

```
# 定义一个全局平均池化后接卷积、批量归一化和 ReLU 的子模块
```

```
class ASPPPooling(nn.Sequential):
```

```
    def __init__(self, in_channels, out_channels):
```

```
        super(ASPPPooling, self).__init__(
```

```
            nn.AdaptiveAvgPool2d(1), # 全局平均池化
```

```
nn.Conv2d(in_channels, out_channels, 1, bias=False), # 1x1 卷积
```

```
nn.BatchNorm2d(out_channels), # 批量归一化
```

```
nn.ReLU()) # ReLU 激活函数
```

```
def forward(self, x):
```

```
    size = x.shape[-2:] # 保存输入特征图的空间维度
```

```
    x = super(ASPPPooling, self).forward(x)
```

```
    # 通过双线性插值将特征图大小调整回原始输入大小
```

```
    return F.interpolate(x, size=size, mode='bilinear', align_corners=False)
```

```
# ASPP 模块主体，结合不同膨胀率的空洞卷积和全局平均池化
```

```
class ASPP(nn.Module):
```

```
    def __init__(self, in_channels, atrous_rates):
```

```
        super(ASPP, self).__init__()
```

```
        out_channels = 256 # 输出通道数
```

```
        modules = []
```

```
        modules.append(nn.Sequential(
```

```
            nn.Conv2d(in_channels, out_channels, 1, bias=False), # 1x1 卷积用于降维
```

```
            nn.BatchNorm2d(out_channels),
```

```
            nn.ReLU()))
```

```
# 根据不同的膨胀率添加空洞卷积模块
```

```
for rate in atrous_rates:
```

```
    modules.append(ASPPConv(in_channels, out_channels, rate))
```

```
# 添加全局平均池化模块
```

```
modules.append(ASPPPooling(in_channels, out_channels))
```

```
self.convs = nn.ModuleList(modules)
```

```
# 将所有模块的输出融合后的投影层
```

```
self.project = nn.Sequential(  
    nn.Conv2d(5 * out_channels, out_channels, 1, bias=False), # 融合特征后降维  
    nn.BatchNorm2d(out_channels),  
    nn.ReLU(),  
    nn.Dropout(0.5)) # 防止过拟合的 Dropout 层
```

```
def forward(self, x):
```

```
    res = []
```

```
    # 对每个模块的输出进行收集
```

```
    for conv in self.convs:
```

```
        res.append(conv(x))
```

```
    # 将收集到的特征在通道维度上拼接
```

```
res = torch.cat(res, dim=1)
```

```
# 对拼接后的特征进行处理
```

```
return self.project(res)
```

```
# 示例使用 ASPP 模块
```

```
aspp = ASPP(256, [6, 12, 18])
```

```
x = torch.rand(2, 256, 13, 13)
```

```
print(aspp(x).shape) # 输出处理后的特征图维度
```

## 43、可变形卷积模块

论文《InternImage: Exploring Large-Scale Vision Foundation Models with Deformable Convolutions》

### 1、作用

该文档介绍了一种基于卷积神经网络（CNNs）的大规模视觉基础模型，名为 InternImage。与近年来取得巨大进展的大规模视觉变换器（ViTs）不同，基于 CNN 的大规模模型仍处于早期阶段。InternImage 通过采用可变形卷积作为核心运算符，不仅具有执行下游任务（如检测和分割）所需的大有效接收场，而且还能够根据输入和任务信息进行自适应空间聚合。

### 2、机制

InternImage 利用可变形卷积（DCN），与传统 CNN 采用的大密集核心运算符不同，它是一种动态稀疏卷积，采用通常的 3x3 窗口大小。这使得模型能够从给定数据中动态学习合适的接收字段（可为长程或短程），并根据输入数据自适应调整采样偏移和调制标量，类似于 ViTs，减少了常规卷积的过度归纳偏置。

### 3、独特优势

InternImage 通过上述设计，可以高效地扩展到大规模参数，并从大规模训练数据中学习更强大的表示，从而在包括 ImageNet、COCO 和 ADE20K 在内的具有挑战性的基准测试中证明了模型的有效性。值得一提的是，InternImage-H 在 COCO test-dev 上创造了新纪录，达到了 65.4 mAP，在 ADE20K 上达到了 62.9 mIoU，超越了当前领先的 CNN 和 ViTs。

## 4、代码

```
import torch

from torch import nn

class DeformConv2d(nn.Module):

    def __init__(self, inc, outc, kernel_size=3, padding=1, stride=1, bias=None, modulation=False):

        """

        Args:

            modulation(bool, optional): If True, Modulated Deformable Convolution(Deformable ConvNets v2).

        """

        super(DeformConv2d, self).__init__()

        self.kernel_size = kernel_size

        self.padding = padding

        self.stride = stride

        self.zero_padding = nn.ZeroPad2d(padding)

        self.conv = nn.Conv2d(inc, outc, kernel_size=kernel_size, stride=kernel_size, bias=bias)

        # self.p_conv 偏置层，学习公式（2）中的偏移量。
```

# 2\*kernel\_size\*kernel\_size: 代表了卷积核中所有元素的偏移坐标，因为同时存在 x 和 y 的偏移，故要乘以 2。

```
self.p_conv = nn.Conv2d(inc, 2 * kernel_size * kernel_size, kernel_size=3, padding=1, stride=stride)
```

```
nn.init.constant_(self.p_conv.weight, 0)
```

# register\_backward\_hook 是为了方便查看这几层学出来的结果，对网络结构无影响。

```
self.p_conv.register_backward_hook(self._set_lr)
```

```
self.modulation = modulation
```

```
if modulation:
```

# self.m\_conv 权重学习层，是后来提出的第二个版本的卷积也就是公式（3）描述的卷积。

# kernel\_size\*kernel\_size: 代表了卷积核中每个元素的权重。

```
self.m_conv = nn.Conv2d(inc, kernel_size * kernel_size, kernel_size=3, padding=1, stride=stride)
```

```
nn.init.constant_(self.m_conv.weight, 0)
```

# register\_backward\_hook 是为了方便查看这几层学出来的结果，对网络结构无影响。

```
self.m_conv.register_backward_hook(self._set_lr)
```

```
@staticmethod
```

```
def _set_lr(module, grad_input, grad_output):
```

```
    grad_input = (grad_input[i] * 0.1 for i in range(len(grad_input)))
```

```
    grad_output = (grad_output[i] * 0.1 for i in range(len(grad_output)))
```

# 生成卷积核的邻域坐标

```
def _get_p_n(self, N, dtype):
```

```
"""
```

`torch.meshgrid()`:Creates grids of coordinates specified by the 1D inputs in attr:tensors.

功能是生成网格，可以用于生成坐标。

函数输入两个数据类型相同的一维张量，两个输出张量的行数为第一个输入张量的元素个数，

列数为第二个输入张量的元素个数，当两个输入张量数据类型不同或维度不是一维时会报错。

其中第一个输出张量填充第一个输入张量中的元素，各行元素相同；

第二个输出张量填充第二个输入张量中的元素各列元素相同。

```
"""
```

```
p_n_x, p_n_y = torch.meshgrid(

    torch.arange(-(self.kernel_size - 1) // 2, (self.kernel_size - 1) // 2 + 1),

    torch.arange(-(self.kernel_size - 1) // 2, (self.kernel_size - 1) // 2 + 1))

# p_n ==> offsets_x(kernel_size*kernel_size,) concat offsets_y(kernel_size*kernel_size,)

#      ==> (2*kernel_size*kernel_size,)

p_n = torch.cat([torch.flatten(p_n_x), torch.flatten(p_n_y)], 0)

# (1, 2*kernel_size*kernel_size, 1, 1)

p_n = p_n.view(1, 2 * N, 1, 1).type(dtype)

return p_n
```

# 获取卷积核在 feature map 上所有对应的中心坐标，也就是 p0

```
def _get_p_0(self, h, w, N, dtype):
```



```

p_0_x, p_0_y = torch.meshgrid(

    torch.arange(1, h * self.stride + 1, self.stride),

    torch.arange(1, w * self.stride + 1, self.stride))

p_0_x = torch.flatten(p_0_x).view(1, 1, h, w).repeat(1, N, 1, 1)

p_0_y = torch.flatten(p_0_y).view(1, 1, h, w).repeat(1, N, 1, 1)

# (b, 2*kernel_size, h, w)

p_0 = torch.cat([p_0_x, p_0_y], 1).type(dtype)

return p_0

```

# 将获取的相对坐标信息与中心坐标相加就获得了卷积核的所有坐标。

# 再加上之前学习得到的 `offset` 后，就是加上了偏移量后的坐标信息。

# 即对应论文中公式(2)中的 $(p_0 + p_n + \Delta p_n)$

```

def _get_p(self, offset, dtype):

    N, h, w = offset.size(1) // 2, offset.size(2), offset.size(3)

    # p_n ==> (1, 2*kernel_size*kernel_size, 1, 1)

    p_n = self._get_p_n(N, dtype)

    # p_0 ==> (1, 2*kernel_size*kernel_size, h, w)

    p_0 = self._get_p_0(h, w, N, dtype)

    # (1, 2*kernel_size*kernel_size, h, w)

    p = p_0 + p_n + offset

    return p

```

```

def _get_x_q(self, x, q, N):

    # b, h, w, 2*kernel_size*kernel_size

    b, h, w, _ = q.size()

    padded_w = x.size(3)

    c = x.size(1)

    # x ==> (b, c, h*w)

    x = x.contiguous().view(b, c, -1)

    # 因为 x 是与 h 轴方向平行, y 是与 w 轴方向平行。故将 2D 卷积核投影到 1D 上, 位移公式如下:

    # 各个卷积核中心坐标及邻域坐标的索引 offsets_x * w + offsets_y

    # (b, h, w, kernel_size*kernel_size)

    index = q[..., :N] * padded_w + q[..., N:]

    # (b, c, h, w, kernel_size*kernel_size) ==> (b, c, h*w*kernel_size*kernel_size)

    index = index.contiguous().unsqueeze(dim=1).expand(-1, c, -1, -1, -1).contiguous().view(b, c, -1)

    # (b, c, h*w)

    # x_offset[0][0][0] = x[0][0][index[0][0][0]]

    # index[i][j][k]的值应该是一一对应着输入 x 的(h*w)的坐标, 且在之前将 index[i][j][k]的值 clamp 在[0, h]及[0, w]范围里。

    # (b, c, h, w, kernel_size*kernel_size)

    x_offset = x.gather(dim=-1, index=index).contiguous().view(b, c, h, w, N)

    return x_offset

@staticmethod

def _reshape_x_offset(x_offset, ks):

```

```

# (b, c, h, w, kernel_size*kernel_size)

b, c, h, w, N = x_offset.size()

# (b, c, h, w*kernel_size)

x_offset = torch.cat([x_offset[..., s:s + ks].contiguous().view(b, c, h, w * ks) for s in range(0, N,
ks)],

                        dim=-1)

# (b, c, h*kernel_size, w*kernel_size)

x_offset = x_offset.contiguous().view(b, c, h * ks, w * ks)


return x_offset


def forward(self, x):

# (b, c, h, w) ==> (b, 2*kernel_size*kernel_size, h, w)

offset = self.p_conv(x)

if self.modulation:

# (b, c, h, w) ==> (b, kernel_size*kernel_size, h, w)

m = torch.sigmoid(self.m_conv(x))


dtype = offset.data.type()

ks = self.kernel_size

# kernel_size*kernel_size

N = offset.size(1) // 2

```

```

if self.padding:

    x = self.zero_padding(x)

    # (b, 2*kernel_size*kernel_size, h, w)

    p = self._get_p(offset, dtype)

    # (b, h, w, 2*kernel_size*kernel_size)

    p = p.contiguous().permute(0, 2, 3, 1)

    # 将 p 从 tensor 的前向计算中取出来，并向下取整得到左上角坐标 q_lt。

    q_lt = p.detach().floor()

    # 将 p 向上再取整，得到右下角坐标 q_rb。

    q_rb = q_lt + 1

    # 学习的偏移量是 float 类型，需要用双线性插值的方法去推算相应的值。

    # 同时防止偏移量太大，超出 feature map，故需要 torch.clamp 来约束。

    # Clamps all elements in input into the range [ min, max ].

    # torch.clamp(a, min=-0.5, max=0.5)

    # p 左上角 x 方向的偏移量不超过 h,y 方向的偏移量不超过 w。

    q_lt = torch.cat([torch.clamp(q_lt[..., :N], 0, x.size(2) - 1), torch.clamp(q_lt[..., N:], 0, x.size(3)
- 1)],

                    dim=-1).long()

    # p 右下角 x 方向的偏移量不超过 h,y 方向的偏移量不超过 w。

    q_rb = torch.cat([torch.clamp(q_rb[..., :N], 0, x.size(2) - 1), torch.clamp(q_rb[..., N:], 0, x.size(3)
- 1)],

```

```
dim=-1).long()
```

# p 左上角的 x 方向的偏移量和右下角 y 方向的偏移量组合起来，得到 p 左下角的值。

```
q_lb = torch.cat([q_lt[..., :N], q_rb[..., N:]], dim=-1)
```

# p 右下角的 x 方向的偏移量和左上角 y 方向的偏移量组合起来，得到 p 右上角的值。

```
q_rt = torch.cat([q_rb[..., :N], q_lt[..., N:]], dim=-1)
```

# clip p。

```
p = torch.cat([torch.clamp(p[..., :N], 0, x.size(2) - 1), torch.clamp(p[..., N:], 0, x.size(3) - 1)],  
dim=-1)
```

# 双线性插值公式里的四个系数。即 **bilinear kernel**。

# 作者代码为了保持整齐，每行的变量计算形式一样，所以计算需要做一点对应变量的对应变化。

```
g_lt = (1 + (q_lt[..., :N].type_as(p) - p[..., :N])) * (1 + (q_lt[..., N:].type_as(p) - p[..., N:]))
```

```
g_rb = (1 - (q_rb[..., :N].type_as(p) - p[..., :N])) * (1 - (q_rb[..., N:].type_as(p) - p[..., N:]))
```

```
g_lb = (1 + (q_lb[..., :N].type_as(p) - p[..., :N])) * (1 - (q_lb[..., N:].type_as(p) - p[..., N:]))
```

```
g_rt = (1 - (q_rt[..., :N].type_as(p) - p[..., :N])) * (1 + (q_rt[..., N:].type_as(p) - p[..., N:]))
```

# 计算双线性插值的四个坐标对应的像素值。

# (b, c, h, w, kernel\_size\*kernel\_size)

```
x_q_lt = self._get_x_q(x, q_lt, N)
```

```
x_q_rb = self._get_x_q(x, q_rb, N)
```

```
x_q_lb = self._get_x_q(x, q_lb, N)
```

```
x_q_rt = self._get_x_q(x, q_rt, N)
```

```

# 双线性插值的最后计算

# (b, c, h, w, kernel_size*kernel_size)

x_offset = g_lt.unsqueeze(dim=1) * x_q_lt + \

            g_rb.unsqueeze(dim=1) * x_q_rb + \

            g_lb.unsqueeze(dim=1) * x_q_lb + \

            g_rt.unsqueeze(dim=1) * x_q_rt


# modulation

if self.modulation:

    # (b, kernel_size*kernel_size, h, w) ==> (b, h, w, kernel_size*kernel_size)

    m = m.contiguous().permute(0, 2, 3, 1)

    # (b, h, w, kernel_size*kernel_size) ==> (b, 1, h, w, kernel_size*kernel_size)

    m = m.unsqueeze(dim=1)

    # (b, c, h, w, kernel_size*kernel_size)

    m = torch.cat([m for _ in range(x_offset.size(1))], dim=1)

    x_offset *= m


# x_offset: (b, c, h, w, kernel_size*kernel_size)

# x_offset: (b, c, h*kernel_size, w*kernel_size)

x_offset = self._reshape_x_offset(x_offset, ks)

# out: (b, c, h, w)

out = self.conv(x_offset)

```

```
return out
```

## 44、蛇形卷积模块

论文《Dynamic Snake Convolution based on Topological Geometric Constraints for Tubular Structure Segmentation》

### 1、作用

本文提出了一种新的框架 DSCNet，旨在精确分割拓扑管状结构，如血管和道路。这些结构在临床应用和遥感应用中至关重要，准确的分割对于下游任务的精确性和效率至关重要。

### 2、机制

#### 1、动态蛇形卷积（DSConv）：

通过适应性聚焦于细长和曲折的局部结构，精确捕获管状结构的特征。不同于可变形卷积，DSConv 考虑管状结构的蛇形形态，并通过约束补充自由学习过程，专注于管状结构的感知。

#### 2、多视角特征融合策略：

基于 DSConv 生成多种形态的内核模板，从多个角度观察目标的结构特征，并通过总结典型的关键特征实现高效的特征融合。

#### 3、拓扑连续性约束损失函数（TCLoss）：

基于持续同调（Persistent Homology, PH）提出一种连续性约束损失函数，以约束分割的拓扑连续性，更好地维持管状结构的完整性。

### 独特优势

#### 1、精确的局部特征捕获：

DSConv 能够适应性地聚焦于细长和曲折的局部特征，与可变形卷积相比，在保持目标形态的同时增强了对管状结构的感知能力。

## 2、有效的特征融合：

通过多视角特征融合策略，模型能够从多个角度综合考虑管状结构的特征，保留了来自不同全局形态的重要信息。

## 3、拓扑连续性的保持：

利用 TCLoss 在拓扑角度对分割连续性进行约束，有效地引导模型关注于可能断裂的区域，提升了管状结构分割的连续性和完整性。

## 4、代码

```
# -*- coding: utf-8 -*-

import os

import torch

import numpy as np

from torch import nn

import warnings

warnings.filterwarnings("ignore")

"""

This code is mainly the deformation process of our DSConv

"""

class DSConv(nn.Module):
```



```

def __init__(self, in_ch, out_ch, kernel_size, extend_scope, morph,
             if_offset, device):

    """

    The Dynamic Snake Convolution

    :param in_ch: input channel

    :param out_ch: output channel

    :param kernel_size: the size of kernel

    :param extend_scope: the range to expand (default 1 for this method)

    :param morph: the morphology of the convolution kernel is mainly divided into two types

                    along the x-axis (0) and the y-axis (1) (see the paper for details)

    :param if_offset: whether deformation is required, if it is False, it is the standard convolution kernel

    :param device: set on gpu

    """

    super(DSConv, self).__init__()

    # use the <offset_conv> to learn the deformable offset

    self.offset_conv = nn.Conv2d(in_ch, 2 * kernel_size, 3, padding=1)

    self.bn = nn.BatchNorm2d(2 * kernel_size)

    self.kernel_size = kernel_size

    # two types of the DSConv (along x-axis and y-axis)

    self.dsc_conv_x = nn.Conv2d(

```

```
        in_ch,

        out_ch,

        kernel_size=(kernel_size, 1),

        stride=(kernel_size, 1),

        padding=0,

    )

    self.dsc_conv_y = nn.Conv2d(

        in_ch,

        out_ch,

        kernel_size=(1, kernel_size),

        stride=(1, kernel_size),

        padding=0,

    )

    self.gn = nn.GroupNorm(out_ch // 4, out_ch)

    self.relu = nn.ReLU(inplace=True)

    self.extend_scope = extend_scope

    self.morph = morph

    self.if_offset = if_offset

    self.device = device
```

```

def forward(self, f):

    offset = self.offset_conv(f)

    offset = self.bn(offset)

    # We need a range of deformation between -1 and 1 to mimic the snake's swing

    offset = torch.tanh(offset)

    input_shape = f.shape

    dsc = DSC(input_shape, self.kernel_size, self.extend_scope, self.morph,

               self.device)

    deformed_feature = dsc.deform_conv(f, offset, self.if_offset)

    if self.morph == 0:

        x = self.dsc_conv_x(deformed_feature)

        x = self.gn(x)

        x = self.relu(x)

        return x

    else:

        x = self.dsc_conv_y(deformed_feature)

        x = self.gn(x)

        x = self.relu(x)

        return x

```

# Core code, for ease of understanding, we mark the dimensions of input and output next to the code

```

class DSC(object):

    def __init__(self, input_shape, kernel_size, extend_scope, morph, device):

        self.num_points = kernel_size

        self.width = input_shape[2]

        self.height = input_shape[3]

        self.morph = morph

        self.device = device

        self.extend_scope = extend_scope # offset (-1 ~ 1) * extend_scope

        # define feature map shape

        """

        B: Batch size  C: Channel  W: Width  H: Height

        """

        self.num_batch = input_shape[0]

        self.num_channels = input_shape[1]


        """

        input: offset [B,2*K,W,H]  K: Kernel size (2*K: 2D image, deformation contains <x_offset> and <y_offset>)

        output_x: [B,1,W,K*H]    coordinate map

        output_y: [B,1,K*W,H]    coordinate map

        """

```

```

def _coordinate_map_3D(self, offset, if_offset):

    # offset

    y_offset, x_offset = torch.split(offset, self.num_points, dim=1)

    y_center = torch.arange(0, self.width).repeat([self.height])

    y_center = y_center.reshape(self.height, self.width)

    y_center = y_center.permute(1, 0)

    y_center = y_center.reshape([-1, self.width, self.height])

    y_center = y_center.repeat([self.num_points, 1, 1]).float()

    y_center = y_center.unsqueeze(0)

    x_center = torch.arange(0, self.height).repeat([self.width])

    x_center = x_center.reshape(self.width, self.height)

    x_center = x_center.permute(0, 1)

    x_center = x_center.reshape([-1, self.width, self.height])

    x_center = x_center.repeat([self.num_points, 1, 1]).float()

    x_center = x_center.unsqueeze(0)

    if self.morph == 0:

        """

        Initialize the kernel and flatten the kernel

```

y: only need 0

x:  $-\text{num\_points} // 2 \sim \text{num\_points} // 2$  (Determined by the kernel size)

step !!! The related PPT will be submitted later, and the PPT will contain the whole changes of each

```
"""
```

```
y = torch.linspace(0, 0, 1)
```

```
x = torch.linspace(  
    -int(self.num_points // 2),  
    int(self.num_points // 2),  
    int(self.num_points),  
)
```

```
y, x = torch.meshgrid(y, x)
```

```
y_spread = y.reshape(-1, 1)
```

```
x_spread = x.reshape(-1, 1)
```

```
y_grid = y_spread.repeat([1, self.width * self.height])
```

```
y_grid = y_grid.reshape([self.num_points, self.width, self.height])
```

```
y_grid = y_grid.unsqueeze(0) # [B*K*K, W,H]
```

```
x_grid = x_spread.repeat([1, self.width * self.height])
```

```
x_grid = x_grid.reshape([self.num_points, self.width, self.height])
```

```
x_grid = x_grid.unsqueeze(0) # [B*K*K, W,H]
```

```
y_new = y_center + y_grid
```

```
x_new = x_center + x_grid
```

```
y_new = y_new.repeat(self.num_batch, 1, 1, 1).to(self.device)
```

```
x_new = x_new.repeat(self.num_batch, 1, 1, 1).to(self.device)
```

```
y_offset_new = y_offset.detach().clone()
```

```
if if_offset:
```

```
    y_offset = y_offset.permute(1, 0, 2, 3)
```

```
    y_offset_new = y_offset_new.permute(1, 0, 2, 3)
```

```
    center = int(self.num_points // 2)
```

```
    # The center position remains unchanged and the rest of the positions begin to swing
```

```
    # This part is quite simple. The main idea is that "offset is an iterative process"
```

```
    y_offset_new[center] = 0
```

```
    for index in range(1, center):
```

```
        y_offset_new[center + index] = (y_offset_new[center + index - 1] + y_offset[center + index])
```

```
        y_offset_new[center - index] = (y_offset_new[center - index + 1] + y_offset[center - index])
```

```
    y_offset_new = y_offset_new.permute(1, 0, 2, 3).to(self.device)
```

```
    y_new = y_new.add(y_offset_new.mul(self.extend_scope))
```

```

y_new = y_new.reshape(

    [self.num_batch, self.num_points, 1, self.width, self.height])

y_new = y_new.permute(0, 3, 1, 4, 2)

y_new = y_new.reshape([

    self.num_batch, self.num_points * self.width, 1 * self.height

])

x_new = x_new.reshape(

    [self.num_batch, self.num_points, 1, self.width, self.height])

x_new = x_new.permute(0, 3, 1, 4, 2)

x_new = x_new.reshape([

    self.num_batch, self.num_points * self.width, 1 * self.height

])

return y_new, x_new

else:

    """

    Initialize the kernel and flatten the kernel

    y: -num_points//2 ~ num_points//2 (Determined by the kernel size)

    x: only need 0

    """

    y = torch.linspace(

```



```

        -int(self.num_points // 2),

        int(self.num_points // 2),

        int(self.num_points),

    )

    x = torch.linspace(0, 1)

    y, x = torch.meshgrid(y, x)

    y_spread = y.reshape(-1, 1)

    x_spread = x.reshape(-1, 1)

    y_grid = y_spread.repeat([1, self.width * self.height])

    y_grid = y_grid.reshape([self.num_points, self.width, self.height])

    y_grid = y_grid.unsqueeze(0)

    x_grid = x_spread.repeat([1, self.width * self.height])

    x_grid = x_grid.reshape([self.num_points, self.width, self.height])

    x_grid = x_grid.unsqueeze(0)

    y_new = y_center + y_grid

    x_new = x_center + x_grid

    y_new = y_new.repeat(self.num_batch, 1, 1, 1)

```

```

x_new = x_new.repeat(self.num_batch, 1, 1, 1)

y_new = y_new.to(self.device)

x_new = x_new.to(self.device)

x_offset_new = x_offset.detach().clone()

if if_offset:

    x_offset = x_offset.permute(1, 0, 2, 3)

    x_offset_new = x_offset_new.permute(1, 0, 2, 3)

    center = int(self.num_points // 2)

    x_offset_new[center] = 0

    for index in range(1, center):

        x_offset_new[center + index] = (x_offset_new[center + index - 1] + x_offset[center + index])

        x_offset_new[center - index] = (x_offset_new[center - index + 1] + x_offset[center - index])

    x_offset_new = x_offset_new.permute(1, 0, 2, 3).to(self.device)

    x_new = x_new.add(x_offset_new.mul(self.extend_scope))

y_new = y_new.reshape(

    [self.num_batch, 1, self.num_points, self.width, self.height])

y_new = y_new.permute(0, 3, 1, 4, 2)

y_new = y_new.reshape([

    self.num_batch, 1 * self.width, self.num_points * self.height

```

```

    ])

    x_new = x_new.reshape(

        [self.num_batch, 1, self.num_points, self.width, self.height])

    x_new = x_new.permute(0, 3, 1, 4, 2)

    x_new = x_new.reshape([

        self.num_batch, 1 * self.width, self.num_points * self.height

    ])

    return y_new, x_new

```

"""

input: input feature map [N,C,D,W,H]; coordinate map [N,K\*D,K\*W,K\*H]

output: [N,1,K\*D,K\*W,K\*H] deformed feature map

"""

```
def _bilinear_interpolate_3D(self, input_feature, y, x):
```

```
    y = y.reshape([-1]).float()
```

```
    x = x.reshape([-1]).float()
```

```
    zero = torch.zeros([1]).int()
```

```
    max_y = self.width - 1
```

```
    max_x = self.height - 1
```

```
# find 8 grid locations

y0 = torch.floor(y).int()

y1 = y0 + 1

x0 = torch.floor(x).int()

x1 = x0 + 1


# clip out coordinates exceeding feature map volume

y0 = torch.clamp(y0, zero, max_y)

y1 = torch.clamp(y1, zero, max_y)

x0 = torch.clamp(x0, zero, max_x)

x1 = torch.clamp(x1, zero, max_x)


input_feature_flat = input_feature.flatten()

input_feature_flat = input_feature_flat.reshape(

    self.num_batch, self.num_channels, self.width, self.height)

input_feature_flat = input_feature_flat.permute(0, 2, 3, 1)

input_feature_flat = input_feature_flat.reshape(-1, self.num_channels)

dimension = self.height * self.width


base = torch.arange(self.num_batch) * dimension

base = base.reshape([-1, 1]).float()
```

```

repeat = torch.ones([self.num_points * self.width * self.height

                        ]).unsqueeze(0)

repeat = repeat.float()

base = torch.matmul(base, repeat)

base = base.reshape([-1])

base = base.to(self.device)

base_y0 = base + y0 * self.height

base_y1 = base + y1 * self.height

# top rectangle of the neighbourhood volume

index_a0 = base_y0 - base + x0

index_c0 = base_y0 - base + x1

# bottom rectangle of the neighbourhood volume

index_a1 = base_y1 - base + x0

index_c1 = base_y1 - base + x1

# get 8 grid values

value_a0 = input_feature_flat[index_a0.type(torch.int64)].to(self.device)

```

```
value_c0 = input_feature_flat[index_c0.type(torch.int64)].to(self.device)

value_a1 = input_feature_flat[index_a1.type(torch.int64)].to(self.device)

value_c1 = input_feature_flat[index_c1.type(torch.int64)].to(self.device)


# find 8 grid locations

y0 = torch.floor(y).int()

y1 = y0 + 1

x0 = torch.floor(x).int()

x1 = x0 + 1


# clip out coordinates exceeding feature map volume

y0 = torch.clamp(y0, zero, max_y + 1)

y1 = torch.clamp(y1, zero, max_y + 1)

x0 = torch.clamp(x0, zero, max_x + 1)

x1 = torch.clamp(x1, zero, max_x + 1)


x0_float = x0.float()

x1_float = x1.float()

y0_float = y0.float()

y1_float = y1.float()


vol_a0 = ((y1_float - y) * (x1_float - x)).unsqueeze(-1).to(self.device)
```

```

vol_c0 = ((y1_float - y) * (x - x0_float)).unsqueeze(-1).to(self.device)

vol_a1 = ((y - y0_float) * (x1_float - x)).unsqueeze(-1).to(self.device)

vol_c1 = ((y - y0_float) * (x - x0_float)).unsqueeze(-1).to(self.device)


outputs = (value_a0 * vol_a0 + value_c0 * vol_c0 + value_a1 * vol_a1 +

           value_c1 * vol_c1)


if self.morph == 0:

    outputs = outputs.reshape([

        self.num_batch,

        self.num_points * self.width,

        1 * self.height,

        self.num_channels,

    ])

    outputs = outputs.permute(0, 3, 1, 2)

else:

    outputs = outputs.reshape([

        self.num_batch,

        1 * self.width,

        self.num_points * self.height,

        self.num_channels,

    ])

```

```

        outputs = outputs.permute(0, 3, 1, 2)

    return outputs

def deform_conv(self, input, offset, if_offset):

    y, x = self._coordinate_map_3D(offset, if_offset)

    deformed_feature = self._bilinear_interpolate_3D(input, y, x)

    return deformed_feature

if __name__ == '__main__':

    os.environ["CUDA_VISIBLE_DEVICES"] = '0'

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    A = np.random.rand(4, 5, 6, 7)

    # A = np.ones(shape=(3, 2, 2, 3), dtype=np.float32)

    # print(A)

    A = A.astype(dtype=np.float32)

    A = torch.from_numpy(A)

    # print(A.shape)

    conv0 = DSConv(

        in_ch=5,

        out_ch=10,

        kernel_size=15,

        extend_scope=1,

```



```
morph=0,  
  
if_offset=True,  
  
device=device)  
  
if torch.cuda.is_available():  
  
    A = A.to(device)  
  
    conv0 = conv0.to(device)  
  
out = conv0(A)  
  
print(out.shape)
```

## 45、深度可分离卷积模块

论文《DeepLab V3》

### 1、作用

DepthwiseSeparableConv 模块主要用于执行深度可分离卷积操作，它是一种高效的卷积方法，广泛应用于减少模型参数数量、计算成本以及提高运行效率等场景，特别是在移动和嵌入式设备上的深度学习应用中。

### 2、机制

#### 1、深度卷积层（Depthwise Convolution）：

对输入的每个通道分别应用卷积操作。这个层使用的是 `nn.Conv2d`，其中 `groups` 参数等于输入通道数，实现了深度卷积。这一层之后紧接着一个批归一化层（`nn.BatchNorm2d`）和一个 `LeakyReLU` 激活函数。

#### 2、逐点卷积层（Pointwise Convolution）：

逐点卷积（也称作 `1x1` 卷积）的目的是组合由深度卷积产生的特征，将它们映射到新的空间中（更改特征图的深度）。与深度卷积层类似，逐点卷积层也包括批归一化和 `LeakyReLU` 激活函数。



```
self.pointwise = nn.Sequential(nn.Conv2d(in_channels, out_channels, 1),

                                nn.BatchNorm2d(out_channels),

                                # activation_layer

                                nn.LeakyReLU(0.1, inplace=True)

                                )

def forward(self, x):

    x = self.depthwise(x)

    x = self.pointwise(x)

    return x
```