Efficient Net

1. 导入库

```python
import math
import copy
from functools import partial
from collections import OrderedDict
from typing import Optional, Callable
import torch
import torch.nn as nn
from torch import Tensor
from torch.nn import functional as F
```

2. 定义辅助函数

定义一个函数来确保所有层的通道数是8的倍数。

```python
def _make_divisible(ch, divisor=8, min_ch=None):
    # 将通道数调整为8的倍数
    if min_ch is None:
        min_ch = divisor  # 设置最小通道数为divisor
    # 计算新的通道数
    new_ch = max(min_ch, int(ch + divisor / 2) // divisor * divisor)
    # 如果新的通道数小于原通道数的90%，则增加divisor
    if new_ch < 0.9 * ch:
        new_ch += divisor
    return new_ch  # 返回调整后的通道数
```

定义 `drop_path` 函数，用于实现随机深度。

```python
def drop_path(x, drop_prob: float = 0., training: bool = False):
    # 如果没有dropout或不是训练状态，直接返回输入
    if drop_prob == 0. or not training:
        return x
    keep_prob = 1 - drop_prob  # 计算保持概率
    shape = (x.shape[0],) + (1,) * (x.ndim - 1)  # 创建随机张量的形状
    # 创建一个随机张量，应用保持概率
```

```
    random_tensor = keep_prob + torch.rand(shape, dtype=x.dtype,
device=x.device)
    random_tensor.floor_()  # 取整
    output = x.div(keep_prob) * random_tensor  # 进行dropout
    return output  # 返回经过dropout的输出
```

3. 定义层类

创建用于卷积、批量归一化和激活函数的类。

```
class DropPath(nn.Module):
    def __init__(self, drop_prob=None):
        # 初始化DropPath层
        super(DropPath, self).__init__()
        self.drop_prob = drop_prob  # 设置dropout概率

    def forward(self, x):
        # 前向传播，应用drop_path
        return drop_path(x, self.drop_prob, self.training)  # 返回经过drop_path处
理的输出
```

定义 `ConvBNActivation` 类，将卷积、批量归一化和激活结合在一起。

```
class ConvBNActivation(nn.Sequential):
    def __init__(self, in_channels: int, out_channels: int, kernel_size: int =
3, stride: int = 1,
                 groups: int = 1, norm_layer: Optional[Callable[...,
nn.Module]] = None,
                 activation_layer: Optional[Callable[..., nn.Module]] = None):
        padding = (kernel_size - 1) // 2  # 计算填充
        if norm_layer is None:
            norm_layer = nn.BatchNorm2d  # 默认使用BatchNorm
        if activation_layer is None:
            activation_layer = nn.SiLU  # 默认使用Swish激活函数

        # 调用父类构造函数，创建卷积、批量归一化和激活层的顺序容器
        super(ConvBNActivation, self).__init__(
            nn.Conv2d(in_channels=in_channels, out_channels=out_channels,
                      kernel_size=kernel_size, stride=stride, padding=padding,
                      groups=groups, bias=False),  # 卷积层
            norm_layer(out_channels),  # 批量归一化层
            activation_layer()  # 激活层
        )
```

该类用于封装卷积、批量归一化和激活函数，便于在后续网络结构中重复使用。该模块设置了卷积层的超参数，以便高效处理输入特征，同时保持输出特征图的尺寸一致。

定义 `SqueezeExcitation` 类，用于压缩和激励机制。

```
class SqueezeExcitation(nn.Module):
    def __init__(self, input_c: int, expand_c: int, squeeze_factor: int = 4):
        super(SqueezeExcitation, self).__init__()
        squeeze_c = input_c // squeeze_factor  # 计算压缩后的通道数
        # 定义两个全连接层
        self.fc1 = nn.Conv2d(expand_c, squeeze_c, 1)  # 第一个全连接层
        self.ac1 = nn.SiLU()  # 激活函数
        self.fc2 = nn.Conv2d(squeeze_c, expand_c, 1)  # 第二个全连接层
        self.ac2 = nn.Sigmoid()  # Sigmoid激活函数

    def forward(self, x: Tensor) -> Tensor:
        # 前向传播，进行Squeeze和Excitation
        scale = F.adaptive_avg_pool2d(x, output_size=(1, 1))  # 池化
        scale = self.fc1(scale)  # 通过第一个全连接层
        scale = self.ac1(scale)  # 激活
        scale = self.fc2(scale)  # 通过第二个全连接层
        scale = self.ac2(scale)  # 激活
        return scale * x  # 返回加权后的输出
```

该模块通过全局平均池化减少空间维度，然后进行缩放以突出重点特征。在 EfficientNet 中，SE模块会在特定位置插入，用于增强网络的特征表示能力。

4. 定义倒残差块

创建倒残差块的配置类和具体实现类。

```
class InvertedResidualConfig:
    def __init__(self, in_channels: int, out_channels: int, kernel_size: int, stride: int,
                 expanded_ratio: int, use_se: bool, drop_rate: float, index: str,
                 width_coefficient: float):
        # 初始化倒残差块配置
        self.in_channels = self.adjust_channels(in_channels, width_coefficient)  # 输入通道数
        self.kernel_size = kernel_size  # 卷积核大小
        self.expanded_channels = self.in_channels * expanded_ratio  # 扩展通道数
```

```
        self.out_channels = self.adjust_channels(out_channels,
width_coefficient)  # 输出通道数
        self.use_se = use_se  # 是否使用SE模块
        self.stride = stride  # 步幅
        self.drop_rate = drop_rate  # dropout率
        self.index = index  # 索引

    @staticmethod
    def adjust_channels(channels: int, width_coefficient: float):
        # 调整通道数为8的倍数
        return _make_divisible(channels * width_coefficient, 8)
```

包含了输入输出通道、卷积核大小、步幅、扩展率等关键信息。这些参数用于构建 EfficientNet 的倒残差块，并提供灵活性来根据输入和输出需求进行调整。

```
class InvertedResidual(nn.Module):
    def __init__(self, cnf: InvertedResidualConfig, norm_layer: Callable[...,
nn.Module]):
        super(InvertedResidual, self).__init__()

        if cnf.stride not in [1, 2]:
            raise ValueError("illegal stride value.")  # 检查步幅是否合法

        # 判断是否使用残差连接
        self.use_res_connect = (cnf.stride == 1 and cnf.in_channels ==
cnf.out_channels)
        layers = OrderedDict()  # 存储层的有序字典
        activation_layer = nn.SiLU  # 使用Swish激活函数

        if cnf.expanded_channels != cnf.in_channels:
            # 如果需要扩展通道，添加扩展卷积层
            layers.update({"expand_conv": ConvBNActivation(cnf.in_channels,
cnf.expanded_channels,
                                                           kernel_size=1,
norm_layer=norm_layer,

activation_layer=activation_layer)})

        # 添加深度卷积层
        layers.update({"dwconv": ConvBNActivation(cnf.expanded_channels,
cnf.expanded_channels,
                                                  kernel_size=cnf.kernel_size,
stride=cnf.stride,

groups=cnf.expanded_channels, norm_layer=norm_layer,
```

```
23
   activation_layer=activation_layer)})
24
25          if cnf.use_se:
26              # 如果使用SE模块，添加SE层
27              layers.update({"se": SqueezeExcitation(cnf.in_channels,
   cnf.expanded_channels)})
28
29          # 添加输出卷积层
30          layers.update({"project_conv": ConvBNActivation(cnf.expanded_channels,
   cnf.out_channels,
31                                                          kernel_size=1,
   norm_layer=norm_layer,
32
   activation_layer=nn.Identity)})
33
34          self.block = nn.Sequential(layers)  # 构建块
35          self.out_channels = cnf.out_channels  # 输出通道数
36          self.is_strided = cnf.stride > 1  # 是否使用步幅
37
38          if self.use_res_connect and cnf.drop_rate > 0:
39              # 如果使用残差连接且需要dropout，添加DropPath
40              self.dropout = DropPath(cnf.drop_rate)
41          else:
42              self.dropout = nn.Identity()  # 否则使用恒等映射
43
44      def forward(self, x: Tensor) -> Tensor:
45          # 前向传播
46          result = self.block(x)  # 通过块
47          result = self.dropout(result)  # 应用dropout
48          if self.use_res_connect:
49              result += x
50          return result
```

使用倒残差结构，通过多个层次的卷积和非线性变化来提取和处理特征。首先检查扩展率是否为 1，以决定是否对通道数进行扩展。然后构建一个深度卷积模块，在高维度的特征空间中对特征进行卷积操作，最后加入 SE 模块（如果启用）来增强该块的特征表达。

5. 定义EfficientNet架构

创建EfficientNet类，整合之前定义的模块。用输入 `width_coefficient` 和 `depth_coefficient` 来动态调整网络的宽度（通道数）和深度（层数），以实现效率和性能的平衡。

- 定义了 `EfficientNet` 类，继承自 `nn.Module`。

- 初始化网络结构和参数，包括宽度和深度系数，分类数量，以及 Dropout 和 Drop Connect 的概率。
- 使用默认的配置参数，构建倒残差块。

特征提取层

- 整合所有倒残差块，构建特征提取模块。
- 通过 `nn.Sequential` 将这些层组合在一起。
- 添加自适应平均池化层，以确保输出的特征图统一。

```
class EfficientNet(nn.Module):
    def __init__(self, width_coefficient: float, depth_coefficient: float, num_classes: int = 2,
                 dropout_rate: float = 0.2, drop_connect_rate: float = 0.2,
                 block: Optional[Callable[..., nn.Module]] = None,
                 norm_layer: Optional[Callable[..., nn.Module]] = None):
        # 初始化EfficientNet模型
        super(EfficientNet, self).__init__()

        # 定义默认的网络配置，每个元素对应一个倒残差块的参数
        default_cnf = [
            [32, 16, 3, 1, 1, True, drop_connect_rate, 1],    # 32输入，16输出，3x3卷积，步幅1，扩展率1，使用SE
            [16, 24, 3, 2, 6, True, drop_connect_rate, 2],    # 16输入，24输出，3x3卷积，步幅2，扩展率6，使用SE
            [24, 40, 5, 2, 6, True, drop_connect_rate, 2],    # 24输入，40输出，5x5卷积，步幅2
            [40, 80, 3, 2, 6, True, drop_connect_rate, 3],    # 40输入，80输出，3x3卷积，步幅2
            [80, 112, 5, 1, 6, True, drop_connect_rate, 3],   # 80输入，112输出，5x5卷积，步幅1
            [112, 192, 5, 2, 6, True, drop_connect_rate, 4], # 112输入，192输出，5x5卷积，步幅2
            [192, 320, 3, 1, 6, True, drop_connect_rate, 1]    # 192输入，320输出，3x3卷积，步幅1
        ]

        # 定义一个函数来根据深度系数调整重复次数
        def round_repeats(repeats):
            return int(math.ceil(depth_coefficient * repeats))  # 向上取整

        if block is None:
            block = InvertedResidual  # 默认使用InvertedResidual块

        if norm_layer is None:
```

```
            norm_layer = partial(nn.BatchNorm2d, eps=1e-3, momentum=0.1)  # 默认的批量归一化层

        # 使用部分函数应用，调整通道数
        adjust_channels = partial(InvertedResidualConfig.adjust_channels, width_coefficient=width_coefficient)

        # 用于构建倒残差块配置的部分函数
        bneck_conf = partial(InvertedResidualConfig, width_coefficient=width_coefficient)

        b = 0  # 用于计数块的数量
        num_blocks = float(sum(round_repeats(i[-1]) for i in default_cnf))  # 计算总块数
        inverted_residual_setting = []  # 存储每个倒残差块的配置

        # 遍历每个阶段的配置
        for stage, args in enumerate(default_cnf):
            cnf = copy.copy(args)  # 复制当前配置
            for i in range(round_repeats(cnf.pop(-1))):  # 根据重复次数添加多个块
                if i > 0:
                    cnf[3] = 1  # 如果是重复的块，步幅设置为1
                    cnf[0] = cnf[1]  # 输入通道数等于输出通道数

                # 更新drop_connect_rate
                cnf[-1] = args[-2] * b / num_blocks
                index = str(stage + 1) + chr(i + 97)  # 创建块的索引
                inverted_residual_setting.append(bneck_conf(*cnf, index))  # 添加配置
                b += 1  # 增加块计数

        layers = OrderedDict()  # 创建有序字典以存储层

        # 添加初始卷积层
        layers.update({
            "stem_conv": ConvBNActivation(in_channels=3,  # 输入通道为3（RGB图像）
                                          out_channels=adjust_channels(32),  # 输出通道根据宽度系数调整
                                          kernel_size=3,  # 卷积核大小为3
                                          stride=2,  # 步幅为2
                                          norm_layer=norm_layer)  # 使用的归一化层
        })

        self.features = nn.Sequential(layers)  # 整合特征提取层
        self.avgpool = nn.AdaptiveAvgPool2d(1)  # 自适应平均池化层
```

6. 分类器和权重初始化

创建分类器，添加 Dropout 和全连接层。

通过遍历所有模块进行权重初始化，确保网络的稳定性和训练的有效性。

```
    # 定义分类器
    classifier = []
    if dropout_rate > 0:
        classifier.append(nn.Dropout(p=dropout_rate, inplace=True))  # 添加Dropout层以减少过拟合
    classifier.append(nn.Linear(last_conv_output_c, num_classes))  # 添加线性分类层
    self.classifier = nn.Sequential(*classifier)  # 整合分类器

    # 初始化权重
    for m in self.modules():
        if isinstance(m, nn.Conv2d):
            nn.init.kaiming_normal_(m.weight, mode="fan_out")  # 卷积层权重初始化
            if m.bias is not None:
                nn.init.zeros_(m.bias)  # 偏置初始化为0
        elif isinstance(m, nn.BatchNorm2d):
            nn.init.ones_(m.weight)  # 归一化层权重初始化为1
            nn.init.zeros_(m.bias)  # 偏置初始化为0
        elif isinstance(m, nn.Linear):
            nn.init.normal_(m.weight, 0, 0.01)  # 线性层权重初始化
            nn.init.zeros_(m.bias)  # 偏置初始化为0
```

7. 前向传播方法

定义前向传播方法，依次通过特征提取层、池化层、展平层和分类器。

```
    def forward(self, x: Tensor) -> Tensor:
        x = self.features(x)  # 通过特征提取层
        x = self.avgpool(x)  # 进行自适应平均池化
        x = torch.flatten(x, 1)  # 展平为一维向量
        x = self.classifier(x)  # 通过分类器
        return x  # 返回输出
```

8. EfficientNet 的不同版本函数

通过多个函数创建不同版本的 EfficientNet，每个函数根据需求调整网络的宽度、深度。

```python
def efficientnet_b0(num_classes=2):
    # 创建 EfficientNet B0 版本，输入图像大小为 224x224
    return EfficientNet(width_coefficient=1.0,
                        depth_coefficient=1.0,
                        dropout_rate=0.2,
                        num_classes=num_classes)

def efficientnet_b1(num_classes=2):
    # 创建 EfficientNet B1 版本，输入图像大小为 240x240
    return EfficientNet(width_coefficient=1.0,
                        depth_coefficient=1.1,
                        dropout_rate=0.2,
                        num_classes=num_classes)

def efficientnet_b2(num_classes=2):
    # 创建 EfficientNet B2 版本，输入图像大小为 260x260
    return EfficientNet(width_coefficient=1.1,
                        depth_coefficient=1.2,
                        dropout_rate=0.3,
                        num_classes=num_classes)

def efficientnet_b3(num_classes=2):
    # 创建 EfficientNet B3 版本，输入图像大小为 300x300
    return EfficientNet(width_coefficient=1.2,
                        depth_coefficient=1.4,
                        dropout_rate=0.3,
                        num_classes=num_classes)

def efficientnet_b4(num_classes=2):
    # 创建 EfficientNet B4 版本，输入图像大小为 380x380
    return EfficientNet(width_coefficient=1.4,
                        depth_coefficient=1.8,
                        dropout_rate=0.4,
                        num_classes=num_classes)

def efficientnet_b5(num_classes=2):
    # 创建 EfficientNet B5 版本，输入图像大小为 456x456
    return EfficientNet(width_coefficient=1.6,
                        depth_coefficient=2.2,
                        dropout_rate=0.4,
                        num_classes=num_classes)

def efficientnet_b6(num_classes=2):
    # 创建 EfficientNet B6 版本，输入图像大小为 528x528
    return EfficientNet(width_coefficient=1.8,
                        depth_coefficient=2.6,
                        dropout_rate=0.5,
```

```
                        num_classes=num_classes)

def efficientnet_b7(num_classes=2):
    # 创建 EfficientNet B7 版本，输入图像大小为 600x600
    return EfficientNet(width_coefficient=2.0,
                        depth_coefficient=3.1,
                        dropout_rate=0.5,
                        num_classes=num_classes)
```