import torch.nn as nn  
import torch  
from torchvision import models  
import torch.nn.functional as F  
  
  
class Conv2d(nn.Module):  
 def \_\_init\_\_(self, in\_channels, out\_channels, kernel\_size, stride=1, dilation=1, bn=False, se='relu', pad=True):  
 super(Conv2d, self).\_\_init\_\_()  
 # print('I am in network conv2d.init')  
 padding = int(dilation \* (kernel\_size - 1) / 2) if pad == True else 0  
 self.conv = nn.Conv2d(in\_channels, out\_channels, kernel\_size, stride, dilation=dilation, padding=padding)  
 self.bn = nn.BatchNorm2d(out\_channels) if bn else None  
 if se == "relu":  
 self.relu = nn.ReLU(inplace=True)  
 elif se == "sigmoid":  
 self.relu = nn.Sigmoid()  
 else:  
 self.relu = None  
 # self.sigmoid = nn.ReLU(inplace=True) if sigmoid else None  
  
 def forward(self, x):  
 # print("iam in work conv2d forward")  
 x = self.conv(x)  
 if self.bn is not None:  
 x = self.bn(x)  
 if self.relu is not None:  
 x = self.relu(x)  
  
 return x  
  
  
def make\_layers(cfg, in\_channels=3, batch\_norm=False, dilation=False):  
 if dilation:  
 d\_rate = 2  
 else:  
 d\_rate = 1  
 layers = []  
 for v in cfg:  
 if v == 'M':  
 layers += [nn.MaxPool2d(kernel\_size=2, stride=2)]  
 else:  
 conv2d = nn.Conv2d(in\_channels, v, kernel\_size=3, padding=d\_rate, dilation=d\_rate)  
 if batch\_norm:  
 layers += [conv2d, nn.BatchNorm2d(v), nn.ReLU(inplace=True)]  
 else:  
 layers += [conv2d, nn.ReLU(inplace=True)]  
 in\_channels = v  
 return nn.Sequential(\*layers)  
  
  
class zt3(nn.Module):  
 def \_\_init\_\_(self, load\_weights=False):  
 super(zt3, self).\_\_init\_\_()  
  
 self.Conv3\_3f = [64, 64, 'M', 128, 128, 'M', 256, 256, 256]  
 self.Conv4\_3f = ['M', 512, 512, 512]  
 self.Conv5\_3f = ['M', 512, 512, 512]  
  
 # self.Conv2\_2 = make\_layers(self.Conv2\_2f, in\_channels=3, batch\_norm=True, dilation=False)  
 self.Conv3\_3 = make\_layers(self.Conv3\_3f, in\_channels=3, batch\_norm=True, dilation=False)  
 self.Conv4\_3 = make\_layers(self.Conv4\_3f, in\_channels=256, batch\_norm=True, dilation=False)  
 self.Conv5\_3 = make\_layers(self.Conv5\_3f, in\_channels=512, batch\_norm=True, dilation=False)  
 # self.back = make\_layers(self.Conv5\_3f, in\_channels=512, batch\_norm=True, dilation=False)  
  
 self.T1 = nn.Sequential(  
 Conv2d(1025, 256, 1, bn=True),  
 Conv2d(256, 256, 3, bn=True),  
 )  
  
 self.T2 = nn.Sequential(  
 Conv2d(513, 128, 1, bn=True),  
 Conv2d(128, 128, 3, bn=True),  
 )  
  
 self.T3 = nn.Sequential(  
 Conv2d(128, 64, 3, bn=True),  
 Conv2d(64, 64, 3, bn=True),  
 Conv2d(64, 1, 1, bn=True),  
 )  
  
 self.dmnT3 = Conv2d(128, 1, 1, bn=True, se='sigmoid')  
 self.d1024b = Conv2d(1024, 1, 1, bn=True)  
 self.d512b = Conv2d(512, 1, 1, bn=True)  
 self.enhance\_pos = multi\_att(dim=[512], top=6)  
  
 if not load\_weights:  
 mod = models.vgg16\_bn(pretrained=True)  
 self.\_initialize\_weights()  
  
 for j in range(len(self.Conv3\_3)):  
 self.Conv3\_3[j].load\_state\_dict(mod.features[j].state\_dict())  
 for p in range(len(self.Conv4\_3)):  
 self.Conv4\_3[p].load\_state\_dict(mod.features[j + p + 1].state\_dict())  
 for q in range(len(self.Conv5\_3)):  
 # self.Conv5\_3[q].load\_state\_dict(mod.features[i + j + p + q + 3+ 1].state\_dict())  
 self.Conv5\_3[q].load\_state\_dict(mod.features[j + p + q + 2].state\_dict())  
  
 def forward(self, img): # the shape of x is 3,3,368,640  
 B, C, H, W = img.shape  
 c3 = self.Conv3\_3(img)  
 c4 = self.Conv4\_3(c3)  
 c5 = self.Conv5\_3(c4)  
 c5, r0 = self.enhance\_pos([c5], H, W)  
 s1 = F.interpolate(c5, scale\_factor=2, mode='bilinear')  
  
 s1 = torch.cat((s1, c4), 1)  
 del c4  
 r1 = self.d1024b(s1)  
  
 s1 = self.T1(torch.cat((s1, r1), 1))  
  
 s2 = F.interpolate(s1, scale\_factor=2, mode='bilinear')  
 s2 = torch.cat((s2, c3), 1)  
 del c3  
 r2 = self.d512b(s2)  
 s2 = self.T2(torch.cat((s2, r2), 1))  
 mask = self.dmnT3(s2)  
 s2 = s2 \* mask  
 r4 = self.T3(s2)  
  
 return r4, [r2, r1, r0], mask  
  
 def \_initialize\_weights(self):  
 for m in self.modules():  
 if isinstance(m, nn.Conv2d):  
 nn.init.xavier\_uniform\_(m.weight, gain=nn.init.calculate\_gain('relu'))  
 if m.bias is not None:  
 nn.init.constant\_(m.bias.data, 0)  
 elif isinstance(m, nn.Linear):  
 nn.init.xavier\_uniform\_(m.weight)  
 if m.bias is not None:  
 nn.init.constant\_(m.bias, 0)  
  
  
class ADConv(nn.Module):  
 def \_\_init\_\_(self, p\_num):  
 super(ADConv, self).\_\_init\_\_()  
 self.conv = nn.Conv2d(1, 1, 1)  
 self.act = nn.Softmax(dim=-1)  
 self.pos\_mask = nn.Sequential(  
 nn.Linear(p\_num, 2 \* p\_num, bias=False),  
 nn.ReLU(inplace=True),  
 nn.Linear(2 \* p\_num, p\_num, bias=False),  
 nn.Sigmoid()  
 )  
  
 def forward(self, x):  
 *'''  
 forward方法中，对输入的图像进行一系列的卷积操作。  
 '''* '''  
 input:  
 weight: b,hw,n0um  
 '''  
 # print("begin", torch.max(x), torch.min(x))  
 B, L, num = x.shape  
 x = x.view(B, 1, L, num)  
 x = self.conv(x).squeeze(dim=1)  
 device = x.get\_device()  
 # one = torch.ones\_like(x, device=device) \* (-100000)  
 # zero = torch.zeros\_like(x, device=device)  
 one = torch.ones\_like(x) \* (-100000)  
 zero = torch.zeros\_like(x)  
 mask = self.pos\_mask(torch.where(x > 0, x, zero))  
 x = self.act(torch.where(x > 0, x, one))  
  
 return x \* mask  
  
  
class multi\_att(nn.Module):  
 def \_\_init\_\_(self, dim, top=9, c\_ratio=8):  
 super(multi\_att, self).\_\_init\_\_()  
 self.p\_num = top  
 cim = int(dim[0] / c\_ratio)  
  
 self.proj\_q = nn.Linear(dim[0], cim, bias=False)  
 self.proj\_k = nn.Linear(dim[0], cim, bias=False)  
 self.proj\_v = nn.Linear(dim[0], cim, bias=False)  
 self.adptive = ADConv(top)  
 self.catt = nn.Parameter(torch.ones([1, dim[0], 1, 1]), requires\_grad=True)  
 self.catt1 = nn.Parameter(torch.ones([1, dim[0], 1, 1]), requires\_grad=True)  
 self.dsn = Conv2d(dim[0], 1, 1, bn=True)  
 self.back = nn.Linear(cim, dim[0], bias=False)  
 self.mlp = Conv2d(dim[0] \* 2, dim[0], 1, bn=True)  
  
 def forward(self, feat, H, W):  
  
 *'''* ***:param*** *feat: list with different scale feature* ***:param*** *size:* ***:return****:  
 '''* # print ("Train",Train)  
  
 if H < 768 and W < 768:  
 return self.get\_forward(feat)  
 else:  
 return self.get\_split\_forward(feat)  
  
 def get\_forward(self, feat):  
  
 B, C, H, W = feat[0].shape  
 q = self.proj\_q(feat[0].flatten(2).transpose(1, 2)) # b,h'w',c/4  
 k = self.proj\_k(feat[0].flatten(2).transpose(1, 2))  
 v = self.proj\_v(feat[0].flatten(2).transpose(1, 2))  
  
 w = cos\_dot(q, k) # (b,hw,c/4) (b,hw1\*4,c/4) b,hw,hw1\*4  
 del q, k  
 w, index = torch.topk(w, self.p\_num, dim=-1) # b,hw,n0um  
 v = get\_top\_value(v, index) # (b,hw1\*3,c) (b,hw,num) -> b,hw,num,c  
 w = self.adptive(w)  
 w = w.view(B, H \* W, 1, self.p\_num) # b,HW,1,num  
 v = torch.matmul(w, v).squeeze(dim=2) # b,hw,c  
 v = self.back(v)  
 v = self.catt \* v.transpose(1, 2).reshape(B, C, H, W)  
 x = self.mlp(torch.cat([v, self.catt1 \* feat[0]], dim=1))  
 r0 = self.dsn(x)  
 return x, r0  
  
 def get\_split\_forward(self, feat):  
 *'''* ***:param*** *feat: list with different scale feature* ***:param*** *size:* ***:return****:  
 '''* # print ("Train",Train)  
 # s\_feat=feat[0]  
 B, C, H, W = feat[0].shape  
 if H >= 384:  
 hn = 4  
 elif H >= 192:  
 hn = 2  
 else:  
 hn = 1  
 if W >= 384:  
 wn = 4  
 elif W >= 192:  
 wn = 2  
 else:  
 wn = 1  
  
 f = get\_chunck(feat[0], [hn, wn])  
 q = self.proj\_q(f.flatten(2).transpose(1, 2)) # b,hw1,c/4  
 k = self.proj\_k(f.flatten(2).transpose(1, 2))  
 v = self.proj\_v(f.flatten(2).transpose(1, 2))  
 w = cos\_dot(q, k)  
 del q, k  
 w, index = torch.topk(w, self.p\_num, dim=-1) # b,hw,n0um  
 v = get\_top\_value(v, index) # (b,hw1\*3,c) (b,hw,num) -> b,hw,num,c  
 w = self.adptive(w)  
 w = w.view(-1, int(H / hn) \* int(W / wn), 1, self.p\_num) # b,HW,1,num  
 v = torch.matmul(w, v).squeeze(dim=2) # b,hw,c  
 v = self.back(v).transpose(1, 2).reshape(-1, C, int(H / hn), int(W / wn))  
 v = get\_back(v, [hn, wn])  
 v = self.catt \* v  
 x = self.mlp(torch.cat([v, self.catt1 \* feat[0]], dim=1))  
 r3 = self.dsn(x)  
  
 return x, r3  
  
  
def get\_back(img, split\_r):  
 B, C, H, W = img.shape  
 hn, wn = split\_r  
 img = torch.cat(torch.split(img, wn, dim=0), dim=2)  
 img = torch.cat(torch.split(img, 1, dim=0), dim=-1)  
 return img  
  
  
def get\_chunck(input, size):  
 # input = torch.randn(1,3,8,6)  
 # print ('ori',input)  
 B, C, H, W = input.shape  
 img = []  
 row = torch.split(input, int(H / size[0]), dim=2)  
 for r in row:  
 img += [torch.cat(torch.split(r, int(W / size[1]), dim=-1), dim=0)]  
 img = torch.cat(img, dim=0)  
 return img  
  
  
def get\_top\_value(value, index):  
 *'''  
 value: b, pri1, other. Where, pri1 reprensets the dim need be selceted, and other is the other dim  
 index:b,pri,num. Where num is response to pri1 in value. here index means the i-th feat in pri need nums in pri1  
 '''* b, pri1, other = value.shape  
 b, pri, num = index.shape  
 indext = torch.stack([index[i] + i \* pri1 for i in range(b)], dim=0)  
 value = value.view(-1, other) # Bc1,hw  
 indext = indext.view(-1, num)  
 indext = indext.transpose(0, 1).contiguous() # num,bpri  
 value = torch.stack([torch.index\_select(value, dim=0, index=n) for n in indext], dim=1)  
 value = value.view(b, pri, num, -1)  
  
 return value  
  
  
def cos\_dot(x, y):  
 *'''  
 x:b,c,hw / b,hw,c  
 y:b,c1,hw / b,hw,c  
 output:  
 weight:b,c,c1 cos\_smilarity  
 '''* x = F.normalize(x, p=2, dim=-1)  
 y = F.normalize(y, p=2, dim=-1)  
 y = y.transpose(1, 2).contiguous()  
 weight = torch.matmul(x, y) # b,c,c1  
 return weight  
  
  
def get\_mask(feat, mask):  
 B, C, H, W = feat.shape  
 mask = F.interpolate(mask, size=[H, W], mode='bilinear', align\_corners=None)  
 return feat \* mask  
  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 model = zt3()  
 # x = torch.ones(1, 3, 256, 256)  
 x = torch.ones(1, 3, 256, 256)  
 mu, mu\_norm = model(x)  
 print(mu.size(), mu\_norm.size())