from einops import rearrange

from copy import deepcopy

from nnformer.utilities.nd\_softmax import softmax\_helper

from torch import nn

import torch

import numpy as np

from nnformer.network\_architecture.initialization import InitWeights\_He

from nnformer.network\_architecture.neural\_network import SegmentationNetwork

import torch.nn.functional

import torch.nn.functional as F

import torch.utils.checkpoint as checkpoint

from timm.models.layers import DropPath, to\_3tuple, trunc\_normal\_

class ContiguousGrad(torch.autograd.Function):

@staticmethod

def forward(ctx, x):

return x

@staticmethod

def backward(ctx, grad\_out):

return grad\_out.contiguous()

class Mlp(nn.Module):

""" Multilayer perceptron."""

def \_\_init\_\_(self, in\_features, hidden\_features=None, out\_features=None, act\_layer=nn.GELU, drop=0.):

super().\_\_init\_\_()

out\_features = out\_features or in\_features

hidden\_features = hidden\_features or in\_features

self.fc1 = nn.Linear(in\_features, hidden\_features)

self.act = act\_layer()

self.fc2 = nn.Linear(hidden\_features, out\_features)

self.drop = nn.Dropout(drop)

def forward(self, x):

x = self.fc1(x)

x = self.act(x)

x = self.drop(x)

x = self.fc2(x)

x = self.drop(x)

return x

def window\_partition(x, window\_size):

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

x = x.view(B, S // window\_size, window\_size, H // window\_size, window\_size, W // window\_size, window\_size, C)

windows = x.permute(0, 1, 3, 5, 2, 4, 6, 7).contiguous().view(-1, window\_size, window\_size, window\_size, C)

return windows

def window\_reverse(windows, window\_size, S, H, W):

B = int(windows.shape[0] / (S \* H \* W / window\_size / window\_size / window\_size))

x = windows.view(B, S // window\_size, H // window\_size, W // window\_size, window\_size, window\_size, window\_size, -1)

x = x.permute(0, 1, 4, 2, 5, 3, 6, 7).contiguous().view(B, S, H, W, -1)

return x

class SwinTransformerBlock\_kv(nn.Module):

def \_\_init\_\_(self, dim, input\_resolution, num\_heads, window\_size=7, shift\_size=0,

mlp\_ratio=4., qkv\_bias=True, qk\_scale=None, drop=0., attn\_drop=0., drop\_path=0.,

act\_layer=nn.GELU, norm\_layer=nn.LayerNorm):

super().\_\_init\_\_()

self.dim = dim

self.input\_resolution = input\_resolution

self.num\_heads = num\_heads

self.window\_size = window\_size

self.shift\_size = shift\_size

self.mlp\_ratio = mlp\_ratio

if min(self.input\_resolution) <= self.window\_size:

# if window size is larger than input resolution, we don't partition windows

self.shift\_size = 0

self.window\_size = min(self.input\_resolution)

assert 0 <= self.shift\_size < self.window\_size, "shift\_size must in 0-window\_size"

self.norm1 = norm\_layer(dim)

self.attn = WindowAttention\_kv(

dim, window\_size=to\_3tuple(self.window\_size), num\_heads=num\_heads,

qkv\_bias=qkv\_bias, qk\_scale=qk\_scale, attn\_drop=attn\_drop, proj\_drop=drop)

#self.window\_size=to\_3tuple(self.window\_size)

self.drop\_path = DropPath(drop\_path) if drop\_path > 0. else nn.Identity()

self.norm2 = norm\_layer(dim)

mlp\_hidden\_dim = int(dim \* mlp\_ratio)

self.mlp = Mlp(in\_features=dim, hidden\_features=mlp\_hidden\_dim, act\_layer=act\_layer, drop=drop)

def forward(self, x, mask\_matrix,skip=None,x\_up=None):

B, L, C = x.shape

S, H, W = self.input\_resolution

assert L == S \* H \* W, "input feature has wrong size"

shortcut = x

skip = self.norm1(skip)

x\_up = self.norm1(x\_up)

skip = skip.view(B, S, H, W, C)

x\_up = x\_up.view(B, S, H, W, C)

x = x.view(B, S, H, W, C)

# pad feature maps to multiples of window size

pad\_r = (self.window\_size - W % self.window\_size) % self.window\_size

pad\_b = (self.window\_size - H % self.window\_size) % self.window\_size

pad\_g = (self.window\_size - S % self.window\_size) % self.window\_size

skip = F.pad(skip, (0, 0, 0, pad\_r, 0, pad\_b, 0, pad\_g))

x\_up = F.pad(x\_up, (0, 0, 0, pad\_r, 0, pad\_b, 0, pad\_g))

\_, Sp, Hp, Wp, \_ = skip.shape

# cyclic shift

if self.shift\_size > 0:

skip = torch.roll(skip, shifts=(-self.shift\_size, -self.shift\_size,-self.shift\_size), dims=(1, 2,3))

x\_up = torch.roll(x\_up, shifts=(-self.shift\_size, -self.shift\_size,-self.shift\_size), dims=(1, 2,3))

attn\_mask = mask\_matrix

else:

skip = skip

x\_up=x\_up

attn\_mask = None

# partition windows

skip = window\_partition(skip, self.window\_size)

skip = skip.view(-1, self.window\_size \* self.window\_size \* self.window\_size,

C)

x\_up = window\_partition(x\_up, self.window\_size)

x\_up = x\_up.view(-1, self.window\_size \* self.window\_size \* self.window\_size,

C)

attn\_windows=self.attn(skip,x\_up,mask=attn\_mask,pos\_embed=None)

# merge windows

attn\_windows = attn\_windows.view(-1, self.window\_size, self.window\_size, self.window\_size, C)

shifted\_x = window\_reverse(attn\_windows, self.window\_size, Sp, Hp, Wp) # B H' W' C

# reverse cyclic shift

if self.shift\_size > 0:

x = torch.roll(shifted\_x, shifts=(self.shift\_size, self.shift\_size, self.shift\_size), dims=(1, 2, 3))

else:

x = shifted\_x

if pad\_r > 0 or pad\_b > 0 or pad\_g > 0:

x = x[:, :S, :H, :W, :].contiguous()

x = x.view(B, S \* H \* W, C)

# FFN

x = shortcut + self.drop\_path(x)

x = x + self.drop\_path(self.mlp(self.norm2(x)))

return x

class WindowAttention\_kv(nn.Module):

def \_\_init\_\_(self, dim, window\_size, num\_heads, qkv\_bias=True, qk\_scale=None, attn\_drop=0., proj\_drop=0.):

super().\_\_init\_\_()

self.dim = dim

self.window\_size = window\_size

self.num\_heads = num\_heads

head\_dim = dim // num\_heads

self.scale = qk\_scale or head\_dim \*\* -0.5

# define a parameter table of relative position bias

self.relative\_position\_bias\_table = nn.Parameter(

torch.zeros((2 \* window\_size[0] - 1) \* (2 \* window\_size[1] - 1) \* (2 \* window\_size[2] - 1),

num\_heads))

# get pair-wise relative position index for each token inside the window

coords\_s = torch.arange(self.window\_size[0])

coords\_h = torch.arange(self.window\_size[1])

coords\_w = torch.arange(self.window\_size[2])

coords = torch.stack(torch.meshgrid([coords\_s, coords\_h, coords\_w]))

coords\_flatten = torch.flatten(coords, 1)

relative\_coords = coords\_flatten[:, :, None] - coords\_flatten[:, None, :]

relative\_coords = relative\_coords.permute(1, 2, 0).contiguous()

relative\_coords[:, :, 0] += self.window\_size[0] - 1 # shift to start from 0

relative\_coords[:, :, 1] += self.window\_size[1] - 1

relative\_coords[:, :, 2] += self.window\_size[2] - 1

relative\_coords[:, :, 0] \*= 3 \* self.window\_size[1] - 1

relative\_coords[:, :, 1] \*= 2 \* self.window\_size[1] - 1

relative\_position\_index = relative\_coords.sum(-1)

self.register\_buffer("relative\_position\_index", relative\_position\_index)

self.kv = nn.Linear(dim, dim \* 2, bias=qkv\_bias)

self.attn\_drop = nn.Dropout(attn\_drop)

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

self.proj\_drop = nn.Dropout(proj\_drop)

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

trunc\_normal\_(self.relative\_position\_bias\_table, std=.02)

def forward(self, skip,x\_up,pos\_embed=None, mask=None):

B\_, N, C = skip.shape

kv = self.kv(skip)

q = x\_up

kv=kv.reshape(B\_, N, 2, self.num\_heads, C // self.num\_heads).permute(2, 0, 3, 1, 4).contiguous()

q = q.reshape(B\_,N,self.num\_heads,C//self.num\_heads).permute(0,2,1,3).contiguous()

k,v = kv[0], kv[1]

q = q \* self.scale

attn = (q @ k.transpose(-2, -1).contiguous())

relative\_position\_bias = self.relative\_position\_bias\_table[self.relative\_position\_index.view(-1)].view(

self.window\_size[0] \* self.window\_size[1] \* self.window\_size[2],

self.window\_size[0] \* self.window\_size[1] \* self.window\_size[2], -1)

relative\_position\_bias = relative\_position\_bias.permute(2, 0, 1).contiguous()

attn = attn + relative\_position\_bias.unsqueeze(0)

if mask is not None:

nW = mask.shape[0]

attn = attn.view(B\_ // nW, nW, self.num\_heads, N, N) + mask.unsqueeze(1).unsqueeze(0)

attn = attn.view(-1, self.num\_heads, N, N)

attn = self.softmax(attn)

else:

attn = self.softmax(attn)

attn = self.attn\_drop(attn)

x = (attn @ v).transpose(1, 2).reshape(B\_, N, C).contiguous()

if pos\_embed is not None:

x = x + pos\_embed

x = self.proj(x)

x = self.proj\_drop(x)

return x

class WindowAttention(nn.Module):

def \_\_init\_\_(self, dim, window\_size, num\_heads, qkv\_bias=True, qk\_scale=None, attn\_drop=0., proj\_drop=0.):

super().\_\_init\_\_()

self.dim = dim

self.window\_size = window\_size

self.num\_heads = num\_heads

head\_dim = dim // num\_heads

self.scale = qk\_scale or head\_dim \*\* -0.5

# define a parameter table of relative position bias

self.relative\_position\_bias\_table = nn.Parameter(

torch.zeros((2 \* window\_size[0] - 1) \* (2 \* window\_size[1] - 1) \* (2 \* window\_size[2] - 1),

num\_heads))

# get pair-wise relative position index for each token inside the window

coords\_s = torch.arange(self.window\_size[0])

coords\_h = torch.arange(self.window\_size[1])

coords\_w = torch.arange(self.window\_size[2])

coords = torch.stack(torch.meshgrid([coords\_s, coords\_h, coords\_w]))

coords\_flatten = torch.flatten(coords, 1)

relative\_coords = coords\_flatten[:, :, None] - coords\_flatten[:, None, :]

relative\_coords = relative\_coords.permute(1, 2, 0).contiguous()

relative\_coords[:, :, 0] += self.window\_size[0] - 1 # shift to start from 0

relative\_coords[:, :, 1] += self.window\_size[1] - 1

relative\_coords[:, :, 2] += self.window\_size[2] - 1

relative\_coords[:, :, 0] \*= 3 \* self.window\_size[1] - 1

relative\_coords[:, :, 1] \*= 2 \* self.window\_size[1] - 1

relative\_position\_index = relative\_coords.sum(-1)

self.register\_buffer("relative\_position\_index", relative\_position\_index)

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)

trunc\_normal\_(self.relative\_position\_bias\_table, std=.02)

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

def forward(self, x, mask=None,pos\_embed=None):

B\_, N, C = x.shape

qkv = self.qkv(x)

qkv=qkv.reshape(B\_, N, 3, self.num\_heads, C // self.num\_heads).permute(2, 0, 3, 1, 4).contiguous()

q, k, v = qkv[0], qkv[1], qkv[2] # make torchscript happy (cannot use tensor as tuple)

q = q \* self.scale

attn = (q @ k.transpose(-2, -1).contiguous())

relative\_position\_bias = self.relative\_position\_bias\_table[self.relative\_position\_index.view(-1)].view(

self.window\_size[0] \* self.window\_size[1] \* self.window\_size[2],

self.window\_size[0] \* self.window\_size[1] \* self.window\_size[2], -1)

relative\_position\_bias = relative\_position\_bias.permute(2, 0, 1).contiguous()

attn = attn + relative\_position\_bias.unsqueeze(0)

if mask is not None:

nW = mask.shape[0]

attn = attn.view(B\_ // nW, nW, self.num\_heads, N, N) + mask.unsqueeze(1).unsqueeze(0)

attn = attn.view(-1, self.num\_heads, N, N)

attn = self.softmax(attn)

else:

attn = self.softmax(attn)

attn = self.attn\_drop(attn)

x = (attn @ v).transpose(1, 2).reshape(B\_, N, C).contiguous()

if pos\_embed is not None:

x = x+pos\_embed

x = self.proj(x)

x = self.proj\_drop(x)

return x

class SwinTransformerBlock(nn.Module):

def \_\_init\_\_(self, dim, input\_resolution, num\_heads, window\_size=7, shift\_size=0,

mlp\_ratio=4., qkv\_bias=True, qk\_scale=None, drop=0., attn\_drop=0., drop\_path=0.,

act\_layer=nn.GELU, norm\_layer=nn.LayerNorm):

super().\_\_init\_\_()

self.dim = dim

self.input\_resolution = input\_resolution

self.num\_heads = num\_heads

self.window\_size = window\_size

self.shift\_size = shift\_size

self.mlp\_ratio = mlp\_ratio

if min(self.input\_resolution) <= self.window\_size:

# if window size is larger than input resolution, we don't partition windows

self.shift\_size = 0

self.window\_size = min(self.input\_resolution)

assert 0 <= self.shift\_size < self.window\_size, "shift\_size must in 0-window\_size"

self.norm1 = norm\_layer(dim)

self.attn = WindowAttention(

dim, window\_size=to\_3tuple(self.window\_size), num\_heads=num\_heads,

qkv\_bias=qkv\_bias, qk\_scale=qk\_scale, attn\_drop=attn\_drop, proj\_drop=drop)

self.drop\_path = DropPath(drop\_path) if drop\_path > 0. else nn.Identity()

self.norm2 = norm\_layer(dim)

mlp\_hidden\_dim = int(dim \* mlp\_ratio)

self.mlp = Mlp(in\_features=dim, hidden\_features=mlp\_hidden\_dim, act\_layer=act\_layer, drop=drop)

def forward(self, x, mask\_matrix):

B, L, C = x.shape

S, H, W = self.input\_resolution

assert L == S \* H \* W, "input feature has wrong size"

shortcut = x

x = self.norm1(x)

x = x.view(B, S, H, W, C)

# pad feature maps to multiples of window size

pad\_r = (self.window\_size - W % self.window\_size) % self.window\_size

pad\_b = (self.window\_size - H % self.window\_size) % self.window\_size

pad\_g = (self.window\_size - S % self.window\_size) % self.window\_size

x = F.pad(x, (0, 0, 0, pad\_r, 0, pad\_b, 0, pad\_g))

\_, Sp, Hp, Wp, \_ = x.shape

# cyclic shift

if self.shift\_size > 0:

shifted\_x = torch.roll(x, shifts=(-self.shift\_size, -self.shift\_size,-self.shift\_size), dims=(1, 2,3))

attn\_mask = mask\_matrix

else:

shifted\_x = x

attn\_mask = None

# partition windows

x\_windows = window\_partition(shifted\_x, self.window\_size) # nW\*B, window\_size, window\_size, C

x\_windows = x\_windows.view(-1, self.window\_size \* self.window\_size \* self.window\_size,

C)

# W-MSA/SW-MSA

attn\_windows = self.attn(x\_windows, mask=attn\_mask,pos\_embed=None)

# merge windows

attn\_windows = attn\_windows.view(-1, self.window\_size, self.window\_size, self.window\_size, C)

shifted\_x = window\_reverse(attn\_windows, self.window\_size, Sp, Hp, Wp)

# reverse cyclic shift

if self.shift\_size > 0:

x = torch.roll(shifted\_x, shifts=(self.shift\_size, self.shift\_size, self.shift\_size), dims=(1, 2, 3))

else:

x = shifted\_x

if pad\_r > 0 or pad\_b > 0 or pad\_g > 0:

x = x[:, :S, :H, :W, :].contiguous()

x = x.view(B, S \* H \* W, C)

# FFN

x = shortcut + self.drop\_path(x)

x = x + self.drop\_path(self.mlp(self.norm2(x)))

return x

class PatchMerging(nn.Module):

def \_\_init\_\_(self, dim, norm\_layer=nn.LayerNorm):

super().\_\_init\_\_()

self.dim = dim

self.reduction = nn.Conv3d(dim,dim\*2,kernel\_size=3,stride=2,padding=1)

self.norm = norm\_layer(dim)

def forward(self, x, S, H, W):

B, L, C = x.shape

assert L == H \* W \* S, "input feature has wrong size"

x = x.view(B, S, H, W, C)

x = F.gelu(x)

x = self.norm(x)

x=x.permute(0,4,1,2,3).contiguous()

x=self.reduction(x)

x=x.permute(0,2,3,4,1).contiguous().view(B,-1,2\*C)

return x

class Patch\_Expanding(nn.Module):

def \_\_init\_\_(self, dim, norm\_layer=nn.LayerNorm):

super().\_\_init\_\_()

self.dim = dim

self.norm = norm\_layer(dim)

self.up=nn.ConvTranspose3d(dim,dim//2,2,2)

def forward(self, x, S, H, W):

B, L, C = x.shape

assert L == H \* W \* S, "input feature has wrong size"

x = x.view(B, S, H, W, C)

x = self.norm(x)

x=x.permute(0,4,1,2,3).contiguous()

x = self.up(x)

x = ContiguousGrad.apply(x)

x=x.permute(0,2,3,4,1).contiguous().view(B,-1,C//2)

return x

class BasicLayer(nn.Module):

def \_\_init\_\_(self,

dim,

input\_resolution,

depth,

num\_heads,

window\_size=7,

mlp\_ratio=4.,

qkv\_bias=True,

qk\_scale=None,

drop=0.,

attn\_drop=0.,

drop\_path=0.,

norm\_layer=nn.LayerNorm,

downsample=True

):

super().\_\_init\_\_()

self.window\_size = window\_size

self.shift\_size = window\_size // 2

self.depth = depth

# build blocks

self.blocks = nn.ModuleList([

SwinTransformerBlock(

dim=dim,

input\_resolution=input\_resolution,

num\_heads=num\_heads,

window\_size=window\_size,

shift\_size=0 if (i % 2 == 0) else window\_size // 2,

mlp\_ratio=mlp\_ratio,

qkv\_bias=qkv\_bias,

qk\_scale=qk\_scale,

drop=drop,

attn\_drop=attn\_drop,

drop\_path=drop\_path[i] if isinstance(drop\_path, list) else drop\_path, norm\_layer=norm\_layer)

for i in range(depth)])

# patch merging layer

if downsample is not None:

self.downsample = downsample(dim=dim, norm\_layer=norm\_layer)

else:

self.downsample = None

def forward(self, x, S, H, W):

# calculate attention mask for SW-MSA

Sp = int(np.ceil(S / self.window\_size)) \* self.window\_size

Hp = int(np.ceil(H / self.window\_size)) \* self.window\_size

Wp = int(np.ceil(W / self.window\_size)) \* self.window\_size

img\_mask = torch.zeros((1, Sp, Hp, Wp, 1), device=x.device) # 1 Hp Wp 1

s\_slices = (slice(0, -self.window\_size),

slice(-self.window\_size, -self.shift\_size),

slice(-self.shift\_size, None))

h\_slices = (slice(0, -self.window\_size),

slice(-self.window\_size, -self.shift\_size),

slice(-self.shift\_size, None))

w\_slices = (slice(0, -self.window\_size),

slice(-self.window\_size, -self.shift\_size),

slice(-self.shift\_size, None))

cnt = 0

for s in s\_slices:

for h in h\_slices:

for w in w\_slices:

img\_mask[:, s, h, w, :] = cnt

cnt += 1

mask\_windows = window\_partition(img\_mask, self.window\_size)

mask\_windows = mask\_windows.view(-1,

self.window\_size \* self.window\_size \* self.window\_size)

attn\_mask = mask\_windows.unsqueeze(1) - mask\_windows.unsqueeze(2)

attn\_mask = attn\_mask.masked\_fill(attn\_mask != 0, float(-100.0)).masked\_fill(attn\_mask == 0, float(0.0))

for blk in self.blocks:

x = blk(x, attn\_mask)

if self.downsample is not None:

x\_down = self.downsample(x, S, H, W)

Ws, Wh, Ww = (S + 1) // 2, (H + 1) // 2, (W + 1) // 2

return x, S, H, W, x\_down, Ws, Wh, Ww

else:

return x, S, H, W, x, S, H, W

class BasicLayer\_up(nn.Module):

def \_\_init\_\_(self,

dim,

input\_resolution,

depth,

num\_heads,

window\_size=7,

mlp\_ratio=4.,

qkv\_bias=True,

qk\_scale=None,

drop=0.,

attn\_drop=0.,

drop\_path=0.,

norm\_layer=nn.LayerNorm,

upsample=True

):

super().\_\_init\_\_()

self.window\_size = window\_size

self.shift\_size = window\_size // 2

self.depth = depth

# build blocks

self.blocks = nn.ModuleList()

self.blocks.append(

SwinTransformerBlock\_kv(

dim=dim,

input\_resolution=input\_resolution,

num\_heads=num\_heads,

window\_size=window\_size,

shift\_size=0 ,

mlp\_ratio=mlp\_ratio,

qkv\_bias=qkv\_bias,

qk\_scale=qk\_scale,

drop=drop,

attn\_drop=attn\_drop,

drop\_path=drop\_path[0] if isinstance(drop\_path, list) else drop\_path, norm\_layer=norm\_layer)

)

for i in range(depth-1):

self.blocks.append(

SwinTransformerBlock(

dim=dim,

input\_resolution=input\_resolution,

num\_heads=num\_heads,

window\_size=window\_size,

shift\_size=window\_size // 2 ,

mlp\_ratio=mlp\_ratio,

qkv\_bias=qkv\_bias,

qk\_scale=qk\_scale,

drop=drop,

attn\_drop=attn\_drop,

drop\_path=drop\_path[i+1] if isinstance(drop\_path, list) else drop\_path, norm\_layer=norm\_layer)

)

self.Upsample = upsample(dim=2\*dim, norm\_layer=norm\_layer)

def forward(self, x,skip, S, H, W):

x\_up = self.Upsample(x, S, H, W)

x = x\_up + skip

S, H, W = S \* 2, H \* 2, W \* 2

# calculate attention mask for SW-MSA

Sp = int(np.ceil(S / self.window\_size)) \* self.window\_size

Hp = int(np.ceil(H / self.window\_size)) \* self.window\_size

Wp = int(np.ceil(W / self.window\_size)) \* self.window\_size

img\_mask = torch.zeros((1, Sp, Hp, Wp, 1), device=x.device) # 1 Hp Wp 1

s\_slices = (slice(0, -self.window\_size),

slice(-self.window\_size, -self.shift\_size),

slice(-self.shift\_size, None))

h\_slices = (slice(0, -self.window\_size),

slice(-self.window\_size, -self.shift\_size),

slice(-self.shift\_size, None))

w\_slices = (slice(0, -self.window\_size),

slice(-self.window\_size, -self.shift\_size),

slice(-self.shift\_size, None))

cnt = 0

for s in s\_slices:

for h in h\_slices:

for w in w\_slices:

img\_mask[:, s, h, w, :] = cnt

cnt += 1

mask\_windows = window\_partition(img\_mask, self.window\_size) # nW, window\_size, window\_size, 1

mask\_windows = mask\_windows.view(-1,

self.window\_size \* self.window\_size \* self.window\_size) # 3d��3��winds�˻�����Ŀ�Ǻܴ�ģ�����winds����̫��

attn\_mask = mask\_windows.unsqueeze(1) - mask\_windows.unsqueeze(2)

attn\_mask = attn\_mask.masked\_fill(attn\_mask != 0, float(-100.0)).masked\_fill(attn\_mask == 0, float(0.0))

x = self.blocks[0](x, attn\_mask,skip=skip,x\_up=x\_up)

for i in range(self.depth-1):

x = self.blocks[i+1](x,attn\_mask)

return x, S, H, W

class project(nn.Module):

def \_\_init\_\_(self,in\_dim,out\_dim,stride,padding,activate,norm,last=False):

super().\_\_init\_\_()

self.out\_dim=out\_dim

self.conv1=nn.Conv3d(in\_dim,out\_dim,kernel\_size=3,stride=stride,padding=padding)

self.conv2=nn.Conv3d(out\_dim,out\_dim,kernel\_size=3,stride=1,padding=1)

self.activate=activate()

self.norm1=norm(out\_dim)

self.last=last

if not last:

self.norm2=norm(out\_dim)

def forward(self,x):

x=self.conv1(x)

x=self.activate(x)

#norm1

Ws, Wh, Ww = x.size(2), x.size(3), x.size(4)

x = x.flatten(2).transpose(1, 2).contiguous()

x = self.norm1(x)

x = x.transpose(1, 2).contiguous().view(-1, self.out\_dim, Ws, Wh, Ww)

x=self.conv2(x)

if not self.last:

x=self.activate(x)

#norm2

Ws, Wh, Ww = x.size(2), x.size(3), x.size(4)

x = x.flatten(2).transpose(1, 2).contiguous()

x = self.norm2(x)

x = x.transpose(1, 2).contiguous().view(-1, self.out\_dim, Ws, Wh, Ww)

return x

class PatchEmbed(nn.Module):

def \_\_init\_\_(self, patch\_size=4, in\_chans=4, embed\_dim=96, norm\_layer=None):

super().\_\_init\_\_()

patch\_size = to\_3tuple(patch\_size)

self.patch\_size = patch\_size

self.in\_chans = in\_chans

self.embed\_dim = embed\_dim

stride1=[patch\_size[0],patch\_size[1]//2,patch\_size[2]//2]

stride2=[patch\_size[0]//2,patch\_size[1]//2,patch\_size[2]//2]

self.proj1 = project(in\_chans,embed\_dim//2,stride1,1,nn.GELU,nn.LayerNorm,False)

self.proj2 = project(embed\_dim//2,embed\_dim,stride2,1,nn.GELU,nn.LayerNorm,True)

if norm\_layer is not None:

self.norm = norm\_layer(embed\_dim)

else:

self.norm = None

def forward(self, x):

"""Forward function."""

# padding

\_, \_, S, H, W = x.size()

if W % self.patch\_size[2] != 0:

x = F.pad(x, (0, self.patch\_size[2] - W % self.patch\_size[2]))

if H % self.patch\_size[1] != 0:

x = F.pad(x, (0, 0, 0, self.patch\_size[1] - H % self.patch\_size[1]))

if S % self.patch\_size[0] != 0:

x = F.pad(x, (0, 0, 0, 0, 0, self.patch\_size[0] - S % self.patch\_size[0]))

x = self.proj1(x) # B C Ws Wh Ww

x = self.proj2(x) # B C Ws Wh Ww

if self.norm is not None:

Ws, Wh, Ww = x.size(2), x.size(3), x.size(4)

x = x.flatten(2).transpose(1, 2).contiguous()

x = self.norm(x)

x = x.transpose(1, 2).contiguous().view(-1, self.embed\_dim, Ws, Wh, Ww)

return x

class Encoder(nn.Module):

def \_\_init\_\_(self,

pretrain\_img\_size=224,

patch\_size=4,

in\_chans=1 ,

embed\_dim=96,

depths=[2, 2, 2, 2],

num\_heads=[4, 8, 16, 32],

window\_size=7,

mlp\_ratio=4.,

qkv\_bias=True,

qk\_scale=None,

drop\_rate=0.,

attn\_drop\_rate=0.,

drop\_path\_rate=0.2,

norm\_layer=nn.LayerNorm,

patch\_norm=True,

out\_indices=(0, 1, 2, 3)

):

super().\_\_init\_\_()

self.pretrain\_img\_size = pretrain\_img\_size

self.num\_layers = len(depths)

self.embed\_dim = embed\_dim

self.patch\_norm = patch\_norm

self.out\_indices = out\_indices

# split image into non-overlapping patches

self.patch\_embed = PatchEmbed(

patch\_size=patch\_size, in\_chans=in\_chans, embed\_dim=embed\_dim,

norm\_layer=norm\_layer if self.patch\_norm else None)

self.pos\_drop = nn.Dropout(p=drop\_rate)

# stochastic depth

dpr = [x.item() for x in torch.linspace(0, drop\_path\_rate, sum(depths))] # stochastic depth decay rule

# build layers

self.layers = nn.ModuleList()

for i\_layer in range(self.num\_layers):

layer = BasicLayer(

dim=int(embed\_dim \* 2 \*\* i\_layer),

input\_resolution=(

pretrain\_img\_size[0] // patch\_size[0] // 2 \*\* i\_layer, pretrain\_img\_size[1] // patch\_size[1] // 2 \*\* i\_layer,

pretrain\_img\_size[2] // patch\_size[2] // 2 \*\* i\_layer),

depth=depths[i\_layer],

num\_heads=num\_heads[i\_layer],

window\_size=window\_size[i\_layer],

mlp\_ratio=mlp\_ratio,

qkv\_bias=qkv\_bias,

qk\_scale=qk\_scale,

drop=drop\_rate,

attn\_drop=attn\_drop\_rate,

drop\_path=dpr[sum(

depths[:i\_layer]):sum(depths[:i\_layer + 1])],

norm\_layer=norm\_layer,

downsample=PatchMerging

if (i\_layer < self.num\_layers - 1) else None

)

self.layers.append(layer)

num\_features = [int(embed\_dim \* 2 \*\* i) for i in range(self.num\_layers)]

self.num\_features = num\_features

# add a norm layer for each output

for i\_layer in out\_indices:

layer = norm\_layer(num\_features[i\_layer])

layer\_name = f'norm{i\_layer}'

self.add\_module(layer\_name, layer)

def forward(self, x):

"""Forward function."""

x = self.patch\_embed(x)

down=[]

Ws, Wh, Ww = x.size(2), x.size(3), x.size(4)

x = x.flatten(2).transpose(1, 2).contiguous()

x = self.pos\_drop(x)

for i in range(self.num\_layers):

layer = self.layers[i]

x\_out, S, H, W, x, Ws, Wh, Ww = layer(x, Ws, Wh, Ww)

if i in self.out\_indices:

norm\_layer = getattr(self, f'norm{i}')

x\_out = norm\_layer(x\_out)

out = x\_out.view(-1, S, H, W, self.num\_features[i]).permute(0, 4, 1, 2, 3).contiguous()

down.append(out)

return down

class Decoder(nn.Module):

def \_\_init\_\_(self,

pretrain\_img\_size,

embed\_dim,

patch\_size=4,

depths=[2,2,2],

num\_heads=[24,12,6],

window\_size=4,

mlp\_ratio=4.,

qkv\_bias=True,

qk\_scale=None,

drop\_rate=0.,

attn\_drop\_rate=0.,

drop\_path\_rate=0.2,

norm\_layer=nn.LayerNorm

):

super().\_\_init\_\_()

self.num\_layers = len(depths)

self.pos\_drop = nn.Dropout(p=drop\_rate)

# stochastic depth

dpr = [x.item() for x in torch.linspace(0, drop\_path\_rate, sum(depths))] # stochastic depth decay rule

# build layers

self.layers = nn.ModuleList()

for i\_layer in range(self.num\_layers)[::-1]:

layer = BasicLayer\_up(

dim=int(embed\_dim \* 2 \*\* (len(depths)-i\_layer-1)),

input\_resolution=(

pretrain\_img\_size[0] // patch\_size[0] // 2 \*\* (len(depths)-i\_layer-1), pretrain\_img\_size[1] // patch\_size[1] // 2 \*\* (len(depths)-i\_layer-1),

pretrain\_img\_size[2] // patch\_size[2] // 2 \*\* (len(depths)-i\_layer-1)),

depth=depths[i\_layer],

num\_heads=num\_heads[i\_layer],

window\_size=window\_size[i\_layer],

mlp\_ratio=mlp\_ratio,

qkv\_bias=qkv\_bias,

qk\_scale=qk\_scale,

drop=drop\_rate,

attn\_drop=attn\_drop\_rate,

drop\_path=dpr[sum(

depths[:i\_layer]):sum(depths[:i\_layer + 1])],

norm\_layer=norm\_layer,

upsample=Patch\_Expanding

)

self.layers.append(layer)

self.num\_features = [int(embed\_dim \* 2 \*\* i) for i in range(self.num\_layers)]

def forward(self,x,skips):

outs=[]

S, H, W = x.size(2), x.size(3), x.size(4)

x = x.flatten(2).transpose(1, 2).contiguous()

for index,i in enumerate(skips):

i = i.flatten(2).transpose(1, 2).contiguous()

skips[index]=i

x = self.pos\_drop(x)

for i in range(self.num\_layers)[::-1]:

layer = self.layers[i]

x, S, H, W, = layer(x,skips[i], S, H, W)

out = x.view(-1, S, H, W, self.num\_features[i])

outs.append(out)

return outs

class final\_patch\_expanding(nn.Module):

def \_\_init\_\_(self,dim,num\_class,patch\_size):

super().\_\_init\_\_()

self.up=nn.ConvTranspose3d(dim,num\_class,patch\_size,patch\_size)

def forward(self,x):

x=x.permute(0,4,1,2,3).contiguous()

x=self.up(x)

return x

class nnFormer(SegmentationNetwork):

def \_\_init\_\_(self, crop\_size=[64,128,128],

embedding\_dim=192,

input\_channels=1,

num\_classes=14,

conv\_op=nn.Conv3d,

depths=[2,2,2,2],

num\_heads=[6, 12, 24, 48],

patch\_size=[2,4,4],

window\_size=[4,4,8,4],

deep\_supervision=True):

super(nnFormer, self).\_\_init\_\_()

self.\_deep\_supervision = deep\_supervision

self.do\_ds = deep\_supervision

self.num\_classes=num\_classes

self.conv\_op=conv\_op

self.upscale\_logits\_ops = []

self.upscale\_logits\_ops.append(lambda x: x)

embed\_dim=embedding\_dim

depths=depths

num\_heads=num\_heads

patch\_size=patch\_size

window\_size=window\_size

self.model\_down=Encoder(pretrain\_img\_size=crop\_size,window\_size=window\_size,embed\_dim=embed\_dim,patch\_size=patch\_size,depths=depths,num\_heads=num\_heads,in\_chans=input\_channels)

self.decoder=Decoder(pretrain\_img\_size=crop\_size,embed\_dim=embed\_dim,window\_size=window\_size[::-1][1:],patch\_size=patch\_size,num\_heads=num\_heads[::-1][1:],depths=depths[::-1][1:])

self.final=[]

if self.do\_ds:

for i in range(len(depths)-1):

self.final.append(final\_patch\_expanding(embed\_dim\*2\*\*i,num\_classes,patch\_size=patch\_size))

else:

self.final.append(final\_patch\_expanding(embed\_dim,num\_classes,patch\_size=patch\_size))

self.final=nn.ModuleList(self.final)

def forward(self, x):

seg\_outputs=[]

skips = self.model\_down(x)

neck=skips[-1]

out=self.decoder(neck,skips)

if self.do\_ds:

for i in range(len(out)):

seg\_outputs.append(self.final[-(i+1)](out[i]))

return seg\_outputs[::-1]

else:

seg\_outputs.append(self.final[0](out[-1]))

return seg\_outputs[-1]