data_utils.py

```python
import torch.utils.data as data
import torchvision.transforms as tfs
from torchvision.transforms import functional as FF
import os,sys
sys.path.append('.')
sys.path.append('..')
import numpy as np
import torch
import random
from PIL import Image
from torch.utils.data import DataLoader
from matplotlib import pyplot as plt
from torchvision.utils import make_grid
from metrics import *
from option import opt
BS=opt.bs
print(BS)
crop_size='whole_img'
if opt.crop:
    crop_size=opt.crop_size

def tensorShow(tensors,titles=None):
        '''
        t:BCWH
        '''
        fig=plt.figure()
        for tensor,tit,i in zip(tensors,titles,range(len(tensors))):
            img = make_grid(tensor)
            npimg = img.numpy()
            ax = fig.add_subplot(211+i)
            ax.imshow(np.transpose(npimg, (1, 2, 0)))
            ax.set_title(tit)
        plt.show()
```

```
class RESIDE_Dataset(data.Dataset):
    def __init__(self,path,train,size=crop_size,format='.png'):
        super(RESIDE_Dataset,self).__init__()
        self.size=size
        print('crop size',size)
        self.train=train
        self.format=format
        self.haze_imgs_dir=os.listdir(os.path.join(path,'hazy'))
        self.haze_imgs=[os.path.join(path,'hazy',img) for img in self.haze_imgs_dir]
        self.clear_dir=os.path.join(path,'clear')
    def __getitem__(self, index):
        haze=Image.open(self.haze_imgs[index])
        if isinstance(self.size,int):
            while haze.size[0]<self.size or haze.size[1]<self.size :
                index=random.randint(0,20000)
                haze=Image.open(self.haze_imgs[index])
        img=self.haze_imgs[index]
        id=img.split('/')[-1].split('_')[0]
        clear_name=id+self.format
        clear=Image.open(os.path.join(self.clear_dir,clear_name))
        clear=tfs.CenterCrop(haze.size[::-1])(clear)
        if not isinstance(self.size,str):
            i,j,h,w=tfs.RandomCrop.get_params(haze,output_size=(self.size,self.size))
            haze=FF.crop(haze,i,j,h,w)
            clear=FF.crop(clear,i,j,h,w)
        haze,clear=self.augData(haze.convert("RGB") ,clear.convert("RGB") )
        return haze,clear
    def augData(self,data,target):
        if self.train:
            rand_hor=random.randint(0,1)
            rand_rot=random.randint(0,3)
            data=tfs.RandomHorizontalFlip(rand_hor)(data)
            target=tfs.RandomHorizontalFlip(rand_hor)(target)
            if rand_rot:
                data=FF.rotate(data,90*rand_rot)
                target=FF.rotate(target,90*rand_rot)
        data=tfs.ToTensor()(data)
        data=tfs.Normalize(mean=[0.64, 0.6, 0.58],std=[0.14,0.15, 0.152])(data)
        target=tfs.ToTensor()(target)
        return  data ,target
    def __len__(self):
        return len(self.haze_imgs)

```

```
import os
pwd=os.getcwd()
print(pwd)
path='/home/zhilin007/VS/FFA-Net/data'#path to your 'data' folder

ITS_train_loader=DataLoader(dataset=RESIDE_Dataset(path+'/RESIDE/ITS',train=True,size=crop_size),batch_size=BS,shuffle=True)
ITS_test_loader=DataLoader(dataset=RESIDE_Dataset(path+'/RESIDE/SOTS/indoor',train=False,size='whole img'),batch_size=1,shuffle=False)

OTS_train_loader=DataLoader(dataset=RESIDE_Dataset(path+'/RESIDE/OTS',train=True,format='.jpg'),batch_size=BS,shuffle=True)
OTS_test_loader=DataLoader(dataset=RESIDE_Dataset(path+'/RESIDE/SOTS/outdoor',train=False,size='whole img',format='.png'),batch_size=1,shuffle=False)

if __name__ == "__main__":
    pass

```

main.py

```python
import torch,os,sys,torchvision,argparse
import torchvision.transforms as tfs
from metrics import psnr,ssim
from models import *
import time,math
import numpy as np
from torch.backends import cudnn
from torch import optim
import torch,warnings
from torch import nn
#from tensorboardX import SummaryWriter
import torchvision.utils as vutils
warnings.filterwarnings('ignore')
from option import opt,model_name,log_dir
from data_utils import *
from torchvision.models import vgg16
print('log_dir :',log_dir)
print('model_name:',model_name)

models_={
    'ffa':FFA(gps=opt.gps,blocks=opt.blocks),
}
loaders_={
    'its_train':ITS_train_loader,
    'its_test':ITS_test_loader,
    'ots_train':OTS_train_loader,
    'ots_test':OTS_test_loader
}
start_time=time.time()
T=opt.steps
```

```python
def lr_schedule_cosdecay(t,T,init_lr=opt.lr):
    lr=0.5*(1+math.cos(t*math.pi/T))*init_lr
    return lr

def train(net,loader_train,loader_test,optim,criterion):
    losses=[]
    start_step=0
    max_ssim=0
    max_psnr=0
    ssims=[]
    psnrs=[]
    if opt.resume and os.path.exists(opt.model_dir):
        print(f'resume from {opt.model_dir}')
        ckp=torch.load(opt.model_dir)
        losses=ckp['losses']
        net.load_state_dict(ckp['model'])
        start_step=ckp['step']
        max_ssim=ckp['max_ssim']
        max_psnr=ckp['max_psnr']
        psnrs=ckp['psnrs']
        ssims=ckp['ssims']
        print(f'start_step:{start_step} start training ---')
    else :
        print('train from scratch *** ')
    for step in range(start_step+1,opt.steps+1):
        net.train()
        lr=opt.lr
        if not opt.no_lr_sche:
            lr=lr_schedule_cosdecay(step,T)
            for param_group in optim.param_groups:
                param_group["lr"] = lr
        x,y=next(iter(loader_train))
        x=x.to(opt.device);y=y.to(opt.device)
        out=net(x)
        loss=criterion[0](out,y)
        if opt.perloss:
            loss2=criterion[1](out,y)
            loss=loss+0.04*loss2

        loss.backward()

        optim.step()
        optim.zero_grad()
        losses.append(loss.item())
        print(f'\rtrain loss : {loss.item():.5f}| step :{step}/{opt.steps}|lr :{lr :.7f} |time_used :\
            {(time.time()-start_time)/60 :.1f}',end='',flush=True)

        #with SummaryWriter(logdir=log_dir,comment=log_dir) as writer:
        #   writer.add_scalar('data/loss',loss,step)
```

```python
        #with SummaryWriter(logdir=log_dir,comment=log_dir) as writer:
        #   writer.add_scalar('data/loss',loss,step)

        if step % opt.eval_step ==0 :
            with torch.no_grad():
                ssim_eval,psnr_eval=test(net,loader_test, max_psnr,max_ssim,step)

            print(f'\nstep :{step} |ssim:{ssim_eval:.4f}| psnr:{psnr_eval:.4f}')

            # with SummaryWriter(logdir=log_dir,comment=log_dir) as writer:
            #   writer.add_scalar('data/ssim',ssim_eval,step)
            #   writer.add_scalar('data/psnr',psnr_eval,step)
            #   writer.add_scalars('group',{
            #       'ssim':ssim_eval,
            #       'psnr':psnr_eval,
            #       'loss':loss
            #   },step)
            ssims.append(ssim_eval)
            psnrs.append(psnr_eval)
            if ssim_eval > max_ssim and psnr_eval > max_psnr :
                max_ssim=max(max_ssim,ssim_eval)
                max_psnr=max(max_psnr,psnr_eval)
                torch.save({
                            'step':step,
                            'max_psnr':max_psnr,
                            'max_ssim':max_ssim,
                            'ssims':ssims,
                            'psnrs':psnrs,
                            'losses':losses,
                            'model':net.state_dict()
                },opt.model_dir)
                print(f'\n model saved at step :{step}| max_psnr:{max_psnr:.4f}|max_ssim:{max_ssim:.4f}')

    np.save(f'./numpy_files/{model_name}_{opt.steps}_losses.npy',losses)
    np.save(f'./numpy_files/{model_name}_{opt.steps}_ssims.npy',ssims)
    np.save(f'./numpy_files/{model_name}_{opt.steps}_psnrs.npy',psnrs)
```

```python
def test(net,loader_test,max_psnr,max_ssim,step):
    net.eval()
    torch.cuda.empty_cache()
    ssims=[]
    psnrs=[]
    #s=True
    for i ,(inputs,targets) in enumerate(loader_test):
        inputs=inputs.to(opt.device);targets=targets.to(opt.device)
        pred=net(inputs)
        # # print(pred)
        # tfs.ToPILImage()(torch.squeeze(targets.cpu())).save('111.png')
        # vutils.save_image(targets.cpu(),'target.png')
        # vutils.save_image(pred.cpu(),'pred.png')
        ssim1=ssim(pred,targets).item()
        psnr1=psnr(pred,targets)
        ssims.append(ssim1)
        psnrs.append(psnr1)
        #if (psnr1>max_psnr or ssim1 > max_ssim) and s :
        #       ts=vutils.make_grid([torch.squeeze(inputs.cpu()),torch.squeeze(targets.cpu()),torch.squeeze(pred.clamp(0,1).cpu())])
        #       vutils.save_image(ts,f'samples/{model_name}/{step}_{psnr1:.4}_{ssim1:.4}.png')
        #       s=False
    return np.mean(ssims) ,np.mean(psnrs)


if __name__ == "__main__":
    loader_train=loaders_[opt.trainset]
    loader_test=loaders_[opt.testset]
    net=models_[opt.net]
    net=net.to(opt.device)
    if opt.device=='cuda':
        net=torch.nn.DataParallel(net)
        cudnn.benchmark=True
    criterion = []
    criterion.append(nn.L1Loss().to(opt.device))
    if opt.perloss:
            vgg_model = vgg16(pretrained=True).features[:16]
            vgg_model = vgg_model.to(opt.device)
            for param in vgg_model.parameters():
                param.requires_grad = False
            criterion.append(PerLoss(vgg_model).to(opt.device))
    optimizer = optim.Adam(params=filter(lambda x: x.requires_grad, net.parameters()),lr=opt.lr, betas = (0.9, 0.999), eps=1e-08)
    optimizer.zero_grad()
    train(net,loader_train,loader_test,optimizer,criterion)
```

metrics.py

```python
from math import exp
import math
import numpy as np

import torch
import torch.nn.functional as F
from torch.autograd import Variable

from math import exp
import math
import numpy as np

import torch
import torch.nn.functional as F
from torch.autograd import Variable
from  torchvision.transforms import ToPILImage

def gaussian(window_size, sigma):
    gauss = torch.Tensor([exp(-(x - window_size // 2) ** 2 / float(2 * sigma ** 2)) for x in range(window_size)])
    return gauss / gauss.sum()

def create_window(window_size, channel):
    _1D_window = gaussian(window_size, 1.5).unsqueeze(1)
    _2D_window = _1D_window.mm(_1D_window.t()).float().unsqueeze(0).unsqueeze(0)
    window = Variable(_2D_window.expand(channel, 1, window_size, window_size).contiguous())
    return window

def _ssim(img1, img2, window, window_size, channel, size_average=True):
    mu1 = F.conv2d(img1, window, padding=window_size // 2, groups=channel)
    mu2 = F.conv2d(img2, window, padding=window_size // 2, groups=channel)
    mu1_sq = mu1.pow(2)
    mu2_sq = mu2.pow(2)
    mu1_mu2 = mu1 * mu2
    sigma1_sq = F.conv2d(img1 * img1, window, padding=window_size // 2, groups=channel) - mu1_sq
    sigma2_sq = F.conv2d(img2 * img2, window, padding=window_size // 2, groups=channel) - mu2_sq
    sigma12 = F.conv2d(img1 * img2, window, padding=window_size // 2, groups=channel) - mu1_mu2
    C1 = 0.01 ** 2
    C2 = 0.03 ** 2
    ssim_map = ((2 * mu1_mu2 + C1) * (2 * sigma12 + C2)) / ((mu1_sq + mu2_sq + C1) * (sigma1_sq + sigma2_sq + C2))

    if size_average:
        return ssim_map.mean()
    else:
        return ssim_map.mean(1).mean(1).mean(1)
```

```python
def ssim(img1, img2, window_size=11, size_average=True):
    img1=torch.clamp(img1,min=0,max=1)
    img2=torch.clamp(img2,min=0,max=1)
    (_, channel, _, _) = img1.size()
    window = create_window(window_size, channel)
    if img1.is_cuda:
        window = window.cuda(img1.get_device())
    window = window.type_as(img1)
    return _ssim(img1, img2, window, window_size, channel, size_average)
def psnr(pred, gt):
    pred=pred.clamp(0,1).cpu().numpy()
    gt=gt.clamp(0,1).cpu().numpy()
    imdff = pred - gt
    rmse = math.sqrt(np.mean(imdff ** 2))
    if rmse == 0:
        return 100
    return 20 * math.log10( 1.0 / rmse)

if __name__ == "__main__":
    pass
```

option.py

```python
import torch,os,sys,torchvision,argparse
import torchvision.transforms as tfs
import time,math
import numpy as np
from torch.backends import cudnn
from torch import optim
import torch,warnings
from torch import nn
import torchvision.utils as vutils
warnings.filterwarnings('ignore')

parser=argparse.ArgumentParser()
parser.add_argument('--steps',type=int,default=100000)
parser.add_argument('--device',type=str,default='Automatic detection')
parser.add_argument('--resume',type=bool,default=True)
parser.add_argument('--eval_step',type=int,default=5000)
parser.add_argument('--lr', default=0.0001, type=float, help='learning rate')
parser.add_argument('--model_dir',type=str,default='./trained_models/')
parser.add_argument('--trainset',type=str,default='its_train')
parser.add_argument('--testset',type=str,default='its_test')
parser.add_argument('--net',type=str,default='ffa')
parser.add_argument('--gps',type=int,default=3,help='residual_groups')
parser.add_argument('--blocks',type=int,default=20,help='residual_blocks')
parser.add_argument('--bs',type=int,default=16,help='batch size')
parser.add_argument('--crop',action='store_true')
parser.add_argument('--crop_size',type=int,default=240,help='Takes effect when using --crop ')
parser.add_argument('--no_lr_sche',action='store_true',help='no lr cos schedule')
parser.add_argument('--perloss',action='store_true',help='perceptual loss')

opt=parser.parse_args()
opt.device='cuda' if torch.cuda.is_available() else 'cpu'
model_name=opt.trainset+'_'+opt.net.split('.')[0]+'_'+str(opt.gps)+'_'+str(opt.blocks)
opt.model_dir=opt.model_dir+model_name+'.pk'
log_dir='logs/'+model_name

print(opt)
print('model_dir:',opt.model_dir)


if not os.path.exists('trained_models'):
    os.mkdir('trained_models')
if not os.path.exists('numpy_files'):
    os.mkdir('numpy_files')
if not os.path.exists('logs'):
    os.mkdir('logs')
if not os.path.exists('samples'):
    os.mkdir('samples')
if not os.path.exists(f"samples/{model_name}"):
    os.mkdir(f'samples/{model_name}')
if not os.path.exists(log_dir):
    os.mkdir(log_dir)

```

test.py

```python
import os,argparse
import numpy as np
from PIL import Image
from models import *
import torch
import torch.nn as nn
import torchvision.transforms as tfs
import torchvision.utils as vutils
import matplotlib.pyplot as plt
from torchvision.utils import make_grid
abs=os.getcwd()+'/'
def tensorShow(tensors,titles=['haze']):
        fig=plt.figure()
        for tensor,tit,i in zip(tensors,titles,range(len(tensors))):
            img = make_grid(tensor)
            npimg = img.numpy()
            ax = fig.add_subplot(221+i)
            ax.imshow(np.transpose(npimg, (1, 2, 0)))
            ax.set_title(tit)
        plt.show()

parser=argparse.ArgumentParser()
parser.add_argument('--task',type=str,default='its',help='its or ots')
parser.add_argument('--test_imgs',type=str,default='test_imgs',help='Test imgs folder')
opt=parser.parse_args()
dataset=opt.task
gps=3
blocks=19
img_dir=abs+opt.test_imgs+'/'
output_dir=abs+f'pred_FFA_{dataset}/'
print("pred_dir:",output_dir)
if not os.path.exists(output_dir):
    os.mkdir(output_dir)
model_dir=abs+f'trained_models/{dataset}_train_ffa_{gps}_{blocks}.pk'
device='cuda' if torch.cuda.is_available() else 'cpu'
ckp=torch.load(model_dir,map_location=device)
net=FFA(gps=gps,blocks=blocks)
net=nn.DataParallel(net)
net.load_state_dict(ckp['model'])
net.eval()
for im in os.listdir(img_dir):
    print(f'\r {im}',end='',flush=True)
    haze = Image.open(img_dir+im)
    haze1= tfs.Compose([
        tfs.ToTensor(),
        tfs.Normalize(mean=[0.64, 0.6, 0.58],std=[0.14,0.15, 0.152])
    ])(haze)[None,::]
    haze_no=tfs.ToTensor()(haze)[None,::]
    with torch.no_grad():
        pred = net(haze1)
    ts=torch.squeeze(pred.clamp(0,1).cpu())
    tensorShow([haze_no,pred.clamp(0,1).cpu()],['haze','pred'])
    vutils.save_image(ts,output_dir+im.split('.')[0]+'_FFA.png')

```

FFA.py

```python
import torch.nn as nn
import torch

def default_conv(in_channels, out_channels, kernel_size, bias=True):
    return nn.Conv2d(in_channels, out_channels, kernel_size,padding=(kernel_size//2), bias=bias)

class PALayer(nn.Module):
    def __init__(self, channel):
        super(PALayer, self).__init__()
        self.pa = nn.Sequential(
                nn.Conv2d(channel, channel // 8, 1, padding=0, bias=True),
                nn.ReLU(inplace=True),
                nn.Conv2d(channel // 8, 1, 1, padding=0, bias=True),
                nn.Sigmoid()
        )
    def forward(self, x):
        y = self.pa(x)
        return x * y

class CALayer(nn.Module):
    def __init__(self, channel):
        super(CALayer, self).__init__()
        self.avg_pool = nn.AdaptiveAvgPool2d(1)
        self.ca = nn.Sequential(
                nn.Conv2d(channel, channel // 8, 1, padding=0, bias=True),
                nn.ReLU(inplace=True),
                nn.Conv2d(channel // 8, channel, 1, padding=0, bias=True),
                nn.Sigmoid()
        )

    def forward(self, x):
        y = self.avg_pool(x)
        y = self.ca(y)
        return x * y
```

```python
class Block(nn.Module):
    def __init__(self, conv, dim, kernel_size,):
        super(Block, self).__init__()
        self.conv1=conv(dim, dim, kernel_size, bias=True)
        self.act1=nn.ReLU(inplace=True)
        self.conv2=conv(dim,dim,kernel_size,bias=True)
        self.calayer=CALayer(dim)
        self.palayer=PALayer(dim)
    def forward(self, x):
        res=self.act1(self.conv1(x))
        res=res+x
        res=self.conv2(res)
        res=self.calayer(res)
        res=self.palayer(res)
        res += x
        return res
class Group(nn.Module):
    def __init__(self, conv, dim, kernel_size, blocks):
        super(Group, self).__init__()
        modules = [ Block(conv, dim, kernel_size)  for _ in range(blocks)]
        modules.append(conv(dim, dim, kernel_size))
        self.gp = nn.Sequential(*modules)
    def forward(self, x):
        res = self.gp(x)
        res += x
        return res
```

```python
class FFA(nn.Module):
    def __init__(self,gps,blocks,conv=default_conv):
        super(FFA, self).__init__()
        self.gps=gps
        self.dim=64
        kernel_size=3
        pre_process = [conv(3, self.dim, kernel_size)]
        assert self.gps==3
        self.g1= Group(conv, self.dim, kernel_size,blocks=blocks)
        self.g2= Group(conv, self.dim, kernel_size,blocks=blocks)
        self.g3= Group(conv, self.dim, kernel_size,blocks=blocks)
        self.ca=nn.Sequential(*[
            nn.AdaptiveAvgPool2d(1),
            nn.Conv2d(self.dim*self.gps,self.dim//16,1,padding=0),
            nn.ReLU(inplace=True),
            nn.Conv2d(self.dim//16, self.dim*self.gps, 1, padding=0, bias=True),
            nn.Sigmoid()
            ])
        self.palayer=PALayer(self.dim)

        post_precess = [
            conv(self.dim, self.dim, kernel_size),
            conv(self.dim, 3, kernel_size)]

        self.pre = nn.Sequential(*pre_process)
        self.post = nn.Sequential(*post_precess)

    def forward(self, x1):
        x = self.pre(x1)
        res1=self.g1(x)
        res2=self.g2(res1)
        res3=self.g3(res2)
        w=self.ca(torch.cat([res1,res2,res3],dim=1))
        w=w.view(-1,self.gps,self.dim)[:,:,:,None,None]
        out=w[:,0,::]*res1+w[:,1,::]*res2+w[:,2,::]*res3
        out=self.palayer(out)
        x=self.post(out)
        return x + x1
if __name__ == "__main__":
    net=FFA(gps=3,blocks=19)
    print(net)
```

PerceptuaLoss.py

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


# --- Perceptual loss network  --- #
class LossNetwork(torch.nn.Module):
    def __init__(self, vgg_model):
        super(LossNetwork, self).__init__()
        self.vgg_layers = vgg_model
        self.layer_name_mapping = {
            '3': "relu1_2",
            '8': "relu2_2",
            '15': "relu3_3"
        }

    def output_features(self, x):
        output = {}
        for name, module in self.vgg_layers._modules.items():
            x = module(x)
            if name in self.layer_name_mapping:
                output[self.layer_name_mapping[name]] = x
        return list(output.values())

    def forward(self, dehaze, gt):
        loss = []
        dehaze_features = self.output_features(dehaze)
        gt_features = self.output_features(gt)
        for dehaze_feature, gt_feature in zip(dehaze_features, gt_features):
            loss.append(F.mse_loss(dehaze_feature, gt_feature))

        return sum(loss)/len(loss)
```