```python
import random
import os
import os.path

import torch.nn.functional as F

import numpy as np
import matplotlib
import matplotlib.pyplot as plt

import math

import torch
import torch.nn as nn
import torchvision.datasets as dsets
import torchvision.transforms as transforms
from torch.autograd import Variable
from scipy.stats import multivariate_normal
from PIL import Image


# set the colormap and centre the colorbar
class MidpointNormalize(matplotlib.colors.Normalize):
    """
    Normalise the colorbar so that diverging bars work there way either side from
a prescribed midpoint value)

    e.g. im=ax1.imshow(array, norm=MidpointNormalize(midpoint=0.,vmin=-100,
vmax=100))
    """
    def __init__(self, vmin=None, vmax=None, midpoint=None, clip=False):
        self.midpoint = midpoint
        matplotlib.colors.Normalize.__init__(self, vmin, vmax, clip)

    def __call__(self, value, clip=None):
        # I'm ignoring masked values and all kinds of edge cases to make a
        # simple example...
        x, y = [self.vmin, self.midpoint, self.vmax], [0, 0.5, 1]
        return np.ma.masked_array(np.interp(value, x, y), np.isnan(value))

def plot_BW(batch):
    fig = plt.figure(figsize=(10,10))
    number_images = batch.size(0)
    for i in range(number_images):
        tensor = batch[i].unsqueeze(0)
        img = tensor.squeeze().numpy()
        fig.add_subplot(4, 4, i+1)
        plt.imshow(img, cmap='gray')
    plt.show()

def plot(batch):
    fig = plt.figure(figsize=(30,30))
    number_images = batch.size(0)
    for i in range(number_images):
        tensor = batch[i]
        img = np.swapaxes(tensor.numpy(), 0, 2)
        img = np.swapaxes(img, 0, 1)
        fig.add_subplot(4, 4, i+1)
        plt.imshow(img)
    #plt.subplot_tool()
    plt.show()

def stats(var):
    t = var.data
    M = t.max()
    m = t.min()
```

```
    mu = t.mean()
    st = t.std()
    print("Max: {}, Min: {}, Avg: {}, Std:{}".format(M, m, mu, st))
    print(t.size())
    return M, m, mu, st

def plot_scores(classnr, score, log = True, figsize=(20, 20), threshold=None,
cmap=matplotlib.cm.seismic, only_positive=False, vmin = None, vmax = None,
savepath=None):
    epsilon = 1e-5

    img = score.data[classnr]

    # if vmin and vmax not fixed from outside, use image max and min
    if vmin == None:
        vmin = score.data.min()
    if vmax == None:
        vmax = score.data.max()

    #assert(vmax > 0 and vmin < 0)
    # vmax must be greater than zero, if not, fixed to epsilon
    if vmax <= 0:
        vmax = +epsilon
    # vmin must be lower than zero, if not, fixed to -epsilon
    if vmin >= 0:
        vmin = -epsilon

    if threshold != None:
        img_plus = torch.mul(img, torch.gt(img, abs(threshold)).float())
        img_minus = torch.mul(img, torch.lt(img, -abs(threshold)).float())
    else:
        img_plus = torch.gt(img, 0).float() * img
        img_minus = torch.lt(img, 0).float() * img

    if log:
        img_plus = torch.log(1.0 + img_plus)
        img_minus = torch.log(1.0 - img_minus)
        vmax = math.log(vmax + 1.0)
        vmin = -math.log(-vmin + 1.0) # gives error if vmin > 0, vmin < 0 is
expected

    img = img_plus
    if not only_positive:
        img -= img_minus

    norm = MidpointNormalize(midpoint=0,vmin=vmin, vmax=vmax)

    img = img.cpu().numpy()

    fig = plt.figure(figsize=figsize)

    sz = score.size(1)
    for i in range(sz):
        fig.add_subplot(8,8, i+1)
        plt.imshow(img[i], cmap=cmap, clim=(vmin, vmax), norm=norm)
        plt.colorbar()

    if savepath != None:
        plt.savefig(savepath)

    plt.show()

def plot_final(score, path, log = True, threshold=None,
cmap=matplotlib.cm.seismic, only_positive=False, vmin=None, vmax=None,
pathMask=None, binary=False):
    base = os.path.splitext(os.path.basename(path))[0]
```

```
    f1 = base + ".1.png"
    f2 = base + ".2.png"
    fmask = base + ".mask.png"
    f3 = base + ".mul.png"
    f3mask = base + ".mul.mask.png"
    f4 = base + ".animated.gif"

    img = score.data

    # if vmin and vmax not fixed from outside, use image max and min
    if vmin == None:
        vmin = score.data.min()
    if vmax == None:
        vmax = score.data.max()

    #assert(vmax > 0 and vmin < 0)
    # vmax must be greater than zero, if not, fixed to epsilon
    if vmax <= 0:
        vmax = +epsilon
    # vmin must be lower than zero, if not, fixed to -epsilon
    if vmin >= 0:
        vmin = -epsilon

    if threshold != None:
        img_plus = torch.mul(img, torch.gt(img, abs(threshold)).float())
        img_minus = torch.mul(img, torch.lt(img, -abs(threshold)).float())
    else:
        img_plus = torch.gt(img, 0).float() * img
        img_minus = torch.lt(img, 0).float() * img

    if log:
        img_plus = torch.log(1.0 + img_plus)
        img_minus = torch.log(1.0 - img_minus)
        vmax = math.log(vmax + 1.0)
        vmin = -math.log(-vmin + 1.0) # gives error if vmin > 0, vmin < 0 is
expected

    img = img_plus
    if not only_positive:
        img -= img_minus

    if binary:
        img = torch.gt(img,0).float() - torch.lt(img,0).float()
        vmin = -1.0
        vmax = +1.0

    norm = MidpointNormalize(midpoint=0,vmin=vmin, vmax=vmax)

    img = img.cpu().numpy()

    # create first image
    img_background = plt.imread(path)
    fig, ax = plt.subplots(figsize=(15,15))
    #fig.add_subplot(2,1,1)
    plt.imshow(img_background)
    plt.imshow(img, cmap=cmap, clim=(vmin, vmax), norm=norm, alpha=0)
    plt.colorbar()
    #fig.add_subplot(2,1,2)
    #plt.imshow(img_background)
    #plt.show()
    plt.savefig(f1)

    # create second image
    fig, ax = plt.subplots(figsize=(15,15))
    #fig.add_subplot(2,1,1)
    plt.imshow(img_background)
```

```
    plt.imshow(img, cmap=cmap, clim=(vmin, vmax), norm=norm, alpha=1)
    plt.colorbar()
    #fig.add_subplot(2,1,2)
    #plt.imshow(img_background)
    #plt.show()
    plt.savefig(f2)

    if pathMask != None:
        im = Image.open(pathMask)
        p = np.array(im)[:,:,0]
        fig, ax = plt.subplots(figsize=(15,15))
        #fig.add_subplot(2,1,1)
        plt.imshow(img_background)
        plt.imshow(img - img*p + p*vmin, cmap=cmap, clim=(vmin, vmax), norm=norm,
alpha=1)
        plt.colorbar()
        plt.savefig(fmask)

    command_img_multiply = "convert " + f1 + " " + f2 + " -compose Multiply -
composite " + f3
    os.system(command_img_multiply)
    command_img_multiply = "convert " + f3 + " " + fmask + " -compose Multiply -
composite " + f3mask
    os.system(command_img_multiply)
    command_img_animated = "convert -loop 0 -delay 250 " + f1 + " " + f3 + " " +
f3mask + " " + f4
    os.system(command_img_animated)

# soumith clarification over batch normalization pytorch parameters
# mean = self.running_mean
# variance = self.running_var
# gamma = self.weight
# beta = self.bias


def propagate_score_through_conv2d_sz2_pad0(score_output_var, input_var,
output_var, block_params):
    # score_output size: nclasses x filter_out x height_out x width_out
    # input size: 1 x filter_in x height_in x width_in

    epsilon = 1e-5 # batch normalization stability constant

    input = input_var.data
    output = output_var.data
    score_output = score_output_var.data
    conv_weights = block_params[0].weight.data   # filter_out x filter_in x 2 x
2
    conv_bias = block_params[0].bias.data        # filter_out
    bn_gamma = block_params[1].weight.data       # filter_out
    bn_beta = block_params[1].bias.data          # filter_out
    bn_mean = block_params[1].running_mean       # filter_out
    bn_var = block_params[1].running_var         # filter_out

    nclasses = score_output.size(0)
    fout_sz = conv_weights.size(0)
    fin_sz = conv_weights.size(1)
    h = input.size(2)
    w = input.size(3)
    ch = 2 # convolution height
    cw = 2 # convolution width

    # In order to split the score proportionally we have to split the calculation
of the convolution
    a = input.squeeze().unsqueeze(0).unsqueeze(2).unsqueeze(2).expand(fout_sz,
fin_sz, ch, cw, h, w)
    b = conv_weights.unsqueeze(4).unsqueeze(4).expand(fout_sz, fin_sz, ch, cw, h,
```

```
w)

    splittedconv2d = torch.mul(a,b)

    # vertices only contain one value, remove other
    # top-row
    splittedconv2d[:,:,1,:,0,:] = 0.0
    # bottom-row
    splittedconv2d[:,:,0,:,-1,:] = 0.0
    # left-col
    splittedconv2d[:,:,:,1,:,0] = 0.0
    # right-col
    splittedconv2d[:,:,:,0,:,-1] = 0.0

    # Now
    gv = torch.div(bn_gamma, torch.sqrt(bn_var + epsilon))       # filter_out
    k = (gv*(conv_bias - bn_mean) + bn_beta) # afterwards multiply by lambda sum

    gv = gv.unsqueeze(1).unsqueeze(2).unsqueeze(3).unsqueeze(4).unsqueeze(5).expand
(fout_sz, fin_sz, ch, cw, h, w)
    splittedconv2d.mul_(gv)

    lamb = torch.div(score_output, output.expand(nclasses,fout_sz,h-1,w-1))
    lamb[output.abs().lt(epsilon).expand(nclasses,fout_sz,h-1,w-1)] = 0.0
    score_k = torch.mul(k.unsqueeze(0).unsqueeze(2).unsqueeze(3).expand(nclasses,
fout_sz, h-1, w-1), lamb)
    lamb = lamb.unsqueeze(2).expand(nclasses, fout_sz, fin_sz, h-1, w-1)

    splittedconv2d = splittedconv2d.unsqueeze(0).expand(nclasses, fout_sz, fin_sz,
ch, cw, h, w).clone()
    # normalize
    for i in range(ch):
        for j in range(cw):
            splittedconv2d[:,:,:,i,j,0+i:h-1+i,0+j:w-1+j].mul_(lamb)
    # sum partial scores
    score_in = splittedconv2d.sum(1).sum(2).sum(2) # sum collapsing all the output
filters

    score_in = torch.autograd.Variable(score_in, volatile=True)
    score_k = torch.autograd.Variable(score_k, volatile=True)

    return score_in, score_k

def propagate_score_through_conv2d_sz3_pad1(score_output_var, input_var,
output_var, block_params):
    # score_in size: nclasses x filter_out x height_out x width_out
    # conv_input size: 1 x filter_in x height x width
    # conv_weights size: filter_in x filter_out x 3 x 3

    epsilon = 1e-5 # batch normalization stability constant

    input = input_var.data
    output = output_var.data
    score_output = score_output_var.data
    conv_weights = block_params[0].weight.data   # filter_out x filter_in x 3 x 3
    conv_bias = block_params[0].bias.data        # filter_out
    bn_gamma = block_params[1].weight.data       # filter_out
    bn_beta = block_params[1].bias.data          # filter_out
    bn_mean = block_params[1].running_mean       # filter_out
    bn_var = block_params[1].running_var         # filter_out

    nclasses = score_output.size(0)
    fout_sz = conv_weights.size(0)
    fin_sz = conv_weights.size(1)
    h = input.size(2)
    w = input.size(3)
```

```
    ch = 3
    cw = 3

    # create input layer that is the original one with a padding of 1 (1 + h + 1)
x (1 + w + 1)
    # we set input=0 in the borders, so we don't have to control weights==0 there
because W·I = 0 due to I = 0
    data = torch.zeros(1,fin_sz,h+2,w+2)
    if input.is_cuda:
        data = data.cuda()
    data[:,:,1:h+1,1:w+1].copy_(input)
    #print(data.size())
    # In order to split the score proportionally we have to split the calculation
of the convolution
    a = data.unsqueeze(2).unsqueeze(2).expand(fout_sz, fin_sz, ch, cw, h+2, w+2)
    b = conv_weights.unsqueeze(4).unsqueeze(4).expand(fout_sz, fin_sz, ch, cw, h
+2, w+2)
    splittedconv2d = torch.mul(a, b)
    del data

    # vertices (due to padding are one inside) only contain some values, remove
other
    # top-row
    splittedconv2d[:,:,2,:,1,:] = 0.0
    # bottom-row
    splittedconv2d[:,:,0,:,-2,:] = 0.0
    # left-col
    splittedconv2d[:,:,:,2,:,1] = 0.0
    # right-col
    splittedconv2d[:,:,:,0,:,-2] = 0.0

    # Now
    gv = bn_gamma/torch.sqrt(bn_var + epsilon)       # filter_out
    k = (gv*(conv_bias - bn_mean) + bn_beta) # afterwards multiply by lambda sum
    #elconst = (bn_alpha + betavar*(conv_bias - bn_mean)) / (fin_sz * ch * cw)  #
filter_out

    gv = gv.unsqueeze(1).unsqueeze(2).unsqueeze(3).unsqueeze(4).unsqueeze(5).expand
(fout_sz, fin_sz, ch, cw, h+2, w+2)
    splittedconv2d.mul_(gv)

    lamb = torch.div(score_output, output.expand(nclasses,fout_sz,h,w))
    lamb[output.abs().lt(epsilon).expand(nclasses,fout_sz,h,w)] = 0.0
    score_k = torch.mul(k.unsqueeze(0).unsqueeze(2).unsqueeze(3).expand(nclasses,
fout_sz, h, w), lamb)
    lamb = lamb.unsqueeze(2).expand(nclasses, fout_sz, fin_sz, h, w)

    # For memory concerns in big layers we calculate per class instead of a big
matrix
    #splittedconv2d = splittedconv2d.unsqueeze(0).expand(nclasses,fout_sz,fin_sz,
ch, cw, h+2, w+2).clone()
    #for i in range(ch):
    #    for j in range(cw):
    #        splittedconv2d[:,:,:,i,j,0+i:h+i,0+j:w+j].mul_(lamb)
    ## sum partial scores
    #score_in = splittedconv2d.sum(1).sum(2).sum(2)[:,:,1:h+1,1:w+1] # sum
collapsing all the output filters

    score_in = torch.zeros(nclasses, fin_sz, h, w)
    if splittedconv2d.is_cuda:
        score_in = score_in.cuda()
    for c in range(nclasses):
        sc = splittedconv2d.expand(fout_sz,fin_sz, ch, cw, h+2, w+2).clone()
        for i in range(ch):
            for j in range(cw):
                sc[:,:,i,j,0+i:h+i,0+j:w+j].mul_(lamb[c])
```

```
        score_in[c].copy_(sc.sum(0).sum(1).sum(1)[:,1:h+1,1:w+1])
        del sc

    score_in = torch.autograd.Variable(score_in, volatile=True)
    score_k = torch.autograd.Variable(score_k, volatile=True)

    return score_in, score_k

def propagate_score_through_maxpool_sz2x2_st2x2(score_in, max_indexes):
    # score_in expected size: nclasses x filter_sz x height x width
    # max_indexes expected size: 1 x filter_sz x height x width
    sz0 = score_in.size(0)
    sz1 = score_in.size(1)
    assert(score_in.size(2) == max_indexes.size(2) and score_in.size(3) ==
max_indexes.size(3))
    szin = score_in.size(2) * score_in.size(3)
    szout = szin * 4
    score_out = torch.autograd.Variable(torch.zeros(sz0,sz1,szout), volatile=True)
    if score_in.is_cuda:
        score_out.data = score_out.data.cuda()
    tmp = score_in.clone().view(sz0,sz1,szin)
    score_out.scatter_(2, max_indexes.view(1,sz1,szin).expand(sz0,sz1,szin), tmp)
    score_out = score_out.view(sz0, sz1, 2*score_in.size(2), 2*score_in.size(3))
    return score_out

# Generates the probability density function of a 2d normal distribution of
# stddev_width with nr_points
def generate_pdf(nr_points, stdev_width = 2., remove_center = False):
    step = float(stdev_width)*2./float(nr_points-1)
    x,y = np.mgrid[-stdev_width:stdev_width+step:step, -stdev_width:stdev_width
+step:step]
    pos = np.empty(x.shape + (2,))
    pos[:,:,0] = x; pos[:,:,1] = y
    rv = multivariate_normal([0,0], [[1,0], [0,1]])
    pdf = rv.pdf(pos)
    pdf = torch.from_numpy(pdf).float()
    if remove_center:
        center = int(nr_points/2) + 1
        pdf1 = torch.cat((pdf[0:center-1, 0:center-1], pdf[0:center-1, center:]),
1)
        pdf2 = torch.cat((pdf[center:, 0:center-1], pdf[center:, center:]), 1)
        pdf = torch.cat((pdf1, pdf2), 0)
    # normalize
    pdf.div_(pdf.sum())
    return pdf

# Converts a hidden layer map to input space mapping every activation as the
# mean of a 2d gaussian of rf_sz equal to rf_stddev. out_sz is the size of the
input space
# rf_sz the size of the receptive field, score_from the hidden space values
treated as
# means
def map_scores_to_input_sz(score_from, rf_sz, out_sz = 640, rf_stddev = 2.):
# Mapping only valid for the hidden layers where a stride 2x2 pad 2x2 has been
previously been applied
# Not valid for first and second layer because no stride have been still applied
    # score_from size Cx1xHxW
    threshold = 1e-6
    # score_from is nclasses x 1 x height_score x width_score
    C = score_from.size(0)
    N = score_from.size(2)

    def get_coordinates(x, y):
        delta = int(out_sz/N)
        half = int(delta/2)
        xout = half + delta*x - 1
```

```
        yout = half + delta*y - 1
        return xout, yout

    out = torch.zeros(C, 1, out_sz, out_sz).float()
    pdf = generate_pdf(rf_sz, rf_stddev, remove_center=True)
    mrf = int(rf_sz/2)

    for c in range(C):
        for i in range(N):
            for j in range(N):
                val = score_from[c][0][i][j].data[0]
                if abs(val) > threshold:
                    xc, yc = get_coordinates(i,j)
                    frx = max(0,xc-mrf+1); tox = min(xc+mrf+1,out_sz)
                    fry = max(0,yc-mrf+1); toy = min(yc+mrf+1,out_sz)
                    #print("frx, tox: {}, {} fry,toy: {},{}".format
(frx,tox,fry,toy))
                    outact = out[c,0,frx:tox,fry:toy]
                    frnx = (mrf - 1) - (xc - frx)
                    tonx = (mrf) + (tox - (xc + 1))
                    frny = (mrf - 1) - (yc - fry)
                    tony = (mrf) + (toy - (yc + 1))
                    #print("frnx, tonx: {}, {} frny,tony: {},{}".format
(frnx,tonx,frny,tony))
                    sel_pdf = pdf[frnx:tonx,frny:tony]
                    outact.add_(val*sel_pdf/sel_pdf.sum(1).sum(0)) # multiply and
normalize by zone inside image
    return torch.autograd.Variable(out, volatile=True)
```