In 1
```python
# So we will combine the ideas from the losses we have experimented with
# InverseDepth Loss works fine if there is no feathering in the subject . But if we penalize also on incorrectly detecting edges . we might improve upon the loss . So a edge-guide is needed

# And we will linearly combine with the loss that we experimented with Online sampling
```
Executed at 2023.12.10 18:16:31 in 4ms

In 4
```python
import torch
from torch import nn
import numpy as np
import torch.nn.functional as F
from torchvision import transforms

from data.loaders.DataLoader import RedWebDataset , Rescale , RandomCrop
from torch.utils.data import DataLoader

normal_dataset = RedWebDataset(root_dir="../data/ReDWeb_V1",transform=transforms.Compose([
    Rescale((256,256)),
]))
batcher = DataLoader(normal_dataset,batch_size=1,shuffle=True)
```
Executed at 2023.12.10 18:21:34 in 9ms

In 6
```python
# Lets Device our online sampling first

# Note this is derived completely from the Redweb paper
def onlineSampling(inputs, targets, masks, threshold, sample_num):

    # find A-B point pairs from predictions (mostly random)
    inputs_index = torch.masked_select(inputs, targets.gt(threshold))
    num_effect_pixels = len(inputs_index)
    shuffle_effect_pixels = torch.randperm(num_effect_pixels).cuda()
    rgb_a = inputs_index[shuffle_effect_pixels[0:sample_num*2:2]]
    rgb_b = inputs_index[shuffle_effect_pixels[1:sample_num*2:2]]

    # find corresponding pairs from ground truth
    depth_index = torch.masked_select(targets, targets.gt(threshold))
    depth_a = depth_index[shuffle_effect_pixels[0:sample_num*2:2]]
    depth_b = depth_index[shuffle_effect_pixels[1:sample_num*2:2]]

    # only compute the losses of point pairs with valid ground truth i.e consistent masked
    consistent_masks_index = torch.masked_select(masks, targets.gt(threshold))
    consistent_masks_A = consistent_masks_index[shuffle_effect_pixels[0:sample_num*2:2]]
    consistent_masks_B = consistent_masks_index[shuffle_effect_pixels[1:sample_num*2:2]]

    # The amount of A and B should be the same!!
    if len(depth_a) > len(depth_b):
        depth_a = depth_a[:-1]
        rgb_a = rgb_a[:-1]
        consistent_masks_A = consistent_masks_A[:-1]

    return rgb_a, rgb_b, depth_a, depth_b, consistent_masks_A, consistent_masks_B

```
Executed at 2023.12.10 18:32:20 in 2ms

In 7
```python
# now lets penalize wrong edges
# i.e if the edeges derived from the depth map does not map with the edges derived from the original image. there should be a corresponding penalty foir it.

# convenience wrapper function to get pixels
def ind2sub(idx, cols):
    r = idx / cols
    c = idx - r * cols
    return r, c


def sub2ind(r, c, cols):
    idx = r * cols + c
    return idx

```
Executed at 2023.12.10 19:26:27 in 5ms

In 8
```python
def edgeGuidedSampling(inputs, targets, edges_img, thetas_img, masks, h, w):

    # find edges
    edges_max = edges_img.max()
    edges_mask = edges_img.ge(edges_max*0.1)
    edges_loc = edges_mask.nonzero()

    inputs_edge = torch.masked_select(inputs, edges_mask)
    targets_edge = torch.masked_select(targets, edges_mask)
    thetas_edge = torch.masked_select(thetas_img, edges_mask)
    minlen = inputs_edge.size()[0]

    # find anchor points (i.e, edge points)
    sample_num = minlen
    index_anchors = torch.randint(
        0, minlen, (sample_num,), dtype=torch.long).cuda()
    anchors = torch.gather(inputs_edge, 0, index_anchors)
    theta_anchors = torch.gather(thetas_edge, 0, index_anchors)
    row_anchors, col_anchors = ind2sub(edges_loc[index_anchors].squeeze(1), w)
    # compute the coordinates of 4-points,  distances are from [2, 30]
    distance_matrix = torch.randint(2, 31, (4, sample_num)).cuda()
    pos_or_neg = torch.ones(4, sample_num).cuda()
    pos_or_neg[:2, :] = -pos_or_neg[:2, :]
    distance_matrix = distance_matrix.float() * pos_or_neg
    col = col_anchors.unsqueeze(0).expand(4, sample_num).long(
    ) + torch.round(distance_matrix.double() * torch.cos(theta_anchors).unsqueeze(0)).long()
    row = row_anchors.unsqueeze(0).expand(4, sample_num).long(
    ) + torch.round(distance_matrix.double() * torch.sin(theta_anchors).unsqueeze(0)).long()

    # constrain 0=<c≤w, 0≤r≤h
    # Note: index should minus 1
    col[col < 0] = 0
    col[col > w-1] = w-1
    row[row < 0] = 0
    row[row > h-1] = h-1

    # a-b, b-c, c-d
    a = sub2ind(row[0, :], col[0, :], w)
    b = sub2ind(row[1, :], col[1, :], w)
    c = sub2ind(row[2, :], col[2, :], w)
    d = sub2ind(row[3, :], col[3, :], w)
    A = torch.cat((a, b, c), 0)
    B = torch.cat((b, c, d), 0)

    rgb_a = torch.gather(inputs, 0, A.long())
    rgb_b = torch.gather(inputs, 0, B.long())
    depth_a = torch.gather(targets, 0, A.long())
    depth_b = torch.gather(targets, 0, B.long())
    masks_A = torch.gather(masks, 0, A.long())
    masks_B = torch.gather(masks, 0, B.long())

    return rgb_a, rgb_b, depth_a, depth_b, masks_A, masks_B, sample_num
```
Executed at 2023.12.10 19:26:39 in 13ms

In _
```python
class EdgeguidedRankingLoss(nn.Module):
    def __init__(self, point_pairs=10000, sigma=0.03, alpha=1.0, mask_value=-1e-8):
        super(EdgeguidedRankingLoss, self).__init__()
        self.point_pairs = point_pairs  # number of point pairs
        self.sigma = sigma  # used for determining the ordinal relationship between a selected pair
        self.alpha = alpha  # used for balancing the effect of = and (<,>)
        self.mask_value = mask_value
        # self.regularization_loss = GradientLoss(scales=4)

    def getEdge(self, images):
        n, c, h, w = images.size()
        a = torch.Tensor([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]]
                         ).cuda().view((1, 1, 3, 3)).repeat(1, 1, 1, 1)
        b = torch.Tensor([[1, 2, 1], [0, 0, 0], [-1, -2, -1]]
                         ).cuda().view((1, 1, 3, 3)).repeat(1, 1, 1, 1)
        if c == 3:
            gradient_x = F.conv2d(images[:, 0, :, :].unsqueeze(1), a)
            gradient_y = F.conv2d(images[:, 0, :, :].unsqueeze(1), b)
        else:
            gradient_x = F.conv2d(images, a)
            gradient_y = F.conv2d(images, b)
        edges = torch.sqrt(torch.pow(gradient_x, 2) + torch.pow(gradient_y, 2))
        edges = F.pad(edges, (1, 1, 1, 1), "constant", 0)
        thetas = torch.atan2(gradient_y, gradient_x)
        thetas = F.pad(thetas, (1, 1, 1, 1), "constant", 0)

        return edges, thetas

    def forward(self, inputs, targets, images, masks=None):
        if masks == None:
            masks = targets > self.mask_value
        # Comment this line if you don't want to use the multi-scale gradient matching term !!!
        # regularization_loss = self.regularization_loss(inputs.squeeze(1), targets.squeeze(1), masks.squeeze(1))
        # find edges from RGB
        edges_img, thetas_img = self.getEdge(images)

        # ============================
        n, c, h, w = targets.size()
        if n != 1:
            inputs = inputs.view(n, -1).double()
            targets = targets.view(n, -1).double()
            masks = masks.view(n, -1).double()
            edges_img = edges_img.view(n, -1).double()
            thetas_img = thetas_img.view(n, -1).double()

        else:
            inputs = inputs.contiguous().view(1, -1).double()
            targets = targets.contiguous().view(1, -1).double()
            masks = masks.contiguous().view(1, -1).double()
            edges_img = edges_img.contiguous().view(1, -1).double()
            thetas_img = thetas_img.contiguous().view(1, -1).double()

        # initialization
        loss = torch.DoubleTensor([0.0]).cuda()

        for i in range(n):
            # Edge-Guided sampling
            rgb_a, rgb_b, depth_a, depth_b, masks_A, masks_B, sample_num = edgeGuidedSampling(
                inputs[i, :], targets[i, :], edges_img[i], thetas_img[i], masks[i, :], h, w)
            # Random Sampling
            random_sample_num = sample_num
            random_rgb_a, random_rgb_b, random_depth_a, random_depth_b, random_masks_A, random_masks_B = onlineSampling(
                inputs[i, :], targets[i, :], masks[i, :], self.mask_value, random_sample_num)

            # Combine EGS + RS
            rgb_a = torch.cat((rgb_a, random_rgb_a), 0)
            rgb_b = torch.cat((rgb_b, random_rgb_b), 0)
            depth_a = torch.cat((depth_a, random_depth_a), 0)
            depth_b = torch.cat((depth_b, random_depth_b), 0)
            masks_A = torch.cat((masks_A, random_masks_A), 0)
            masks_B = torch.cat((masks_B, random_masks_B), 0)

            # GT ordinal relationship
            target_ratio = torch.div(depth_a+1e-6, depth_b+1e-6)
            mask_eq = target_ratio.lt(
                1.0 + self.sigma) * target_ratio.gt(1.0/(1.0+self.sigma))
            labels = torch.zeros_like(target_ratio)
            labels[target_ratio.ge(1.0 + self.sigma)] = 1
            labels[target_ratio.le(1.0/(1.0+self.sigma))] = -1

            # consider forward-backward consistency checking, i.e, only compute losses of point pairs with valid GT
            consistency_mask = masks_A * masks_B

            equal_loss = (rgb_a - rgb_b).pow(2) * \
                         mask_eq.double() * consistency_mask
            unequal_loss = torch.log(
                1 + torch.exp((-rgb_a + rgb_b) * labels)) * (~mask_eq).double() * consistency_mask

            # Please comment the regularization term if you don't want to use the multi-scale gradient matching loss !!!
            # + 0.2 * regularization_loss.double()
            loss = loss + self.alpha * equal_loss.mean() + 1.0 * unequal_loss.mean()

        return loss[0].float()/n
```