

```
In 1 1 from __future__ import print_function, division
2 import os
3 import torch
4 import pandas as pd
5 from skimage import io, transform
6 import numpy as np
7 import matplotlib.pyplot as plt
8 from torch.utils.data import Dataset, DataLoader
9 from torchvision import transforms, utils
10
11 # Ignore warnings
12 import warnings
13 warnings.filterwarnings("ignore")
14
15 plt.ion() # interactive mode
Executed at 2023.12.01 22:50:33 in 97ms
> /Users/adi/opt/anaconda3/lib/python3.10/site-packages/torchvision/io/image.py:13: UserWarning: Failed to load image Pyth
```

Out 1 <contextlib.ExitStack at 0x10657f20>

```
In 2 1 # from my utils notebook
2 def add_to_class(klass):
3     """Register them functions"""
4     def wrapper(obj):
5         # setattr(object, name, value) → sets the value of the attribute
6         setattr(klass, obj.__name__, obj)
7     return wrapper
Executed at 2023.12.01 22:50:33 in 4ms
```

```
In 3 1 class RedWebDataset(Dataset):
2     def __init__(self, root_dir, transform=None):
3         """
4             Root Dir is REDWEB_V1
5                 Imgs → Monocular Images
6                 RDs → Corresponding Response (Heatmap)
7         """
8         self.root_dir = root_dir
9         self.transform = transform
10        self.monocular_folder = os.path.join(root_dir, 'Imgs')
11        self.heatmap_folder = os.path.join(root_dir, 'RDs')
12
13        # Monocular images are in jpgs and the heatmap images are in pngs
14        self.monocular_images = sorted(os.listdir(self.monocular_folder))
15        self.heatmap_images = sorted(os.listdir(self.heatmap_folder))
16
Executed at 2023.12.01 22:50:33 in 2ms
```

now we can get images from our loaders:

```
In 4 1 loader = RedWebDataset("../RedWeb_V1")
2 loader.monocular_images[:2]
Executed at 2023.12.01 22:50:33 in 58ms
```

Out 4 ['1014775965\_3367569158.jpg', '10147764085\_481cd7d72f.jpg']

```
In 5 1 loader.heatmap_images[:2]
Executed at 2023.12.01 22:50:33 in 16ms
```

Out 5 ['1014775965\_3367569158.png', '10147764085\_481cd7d72f.png']

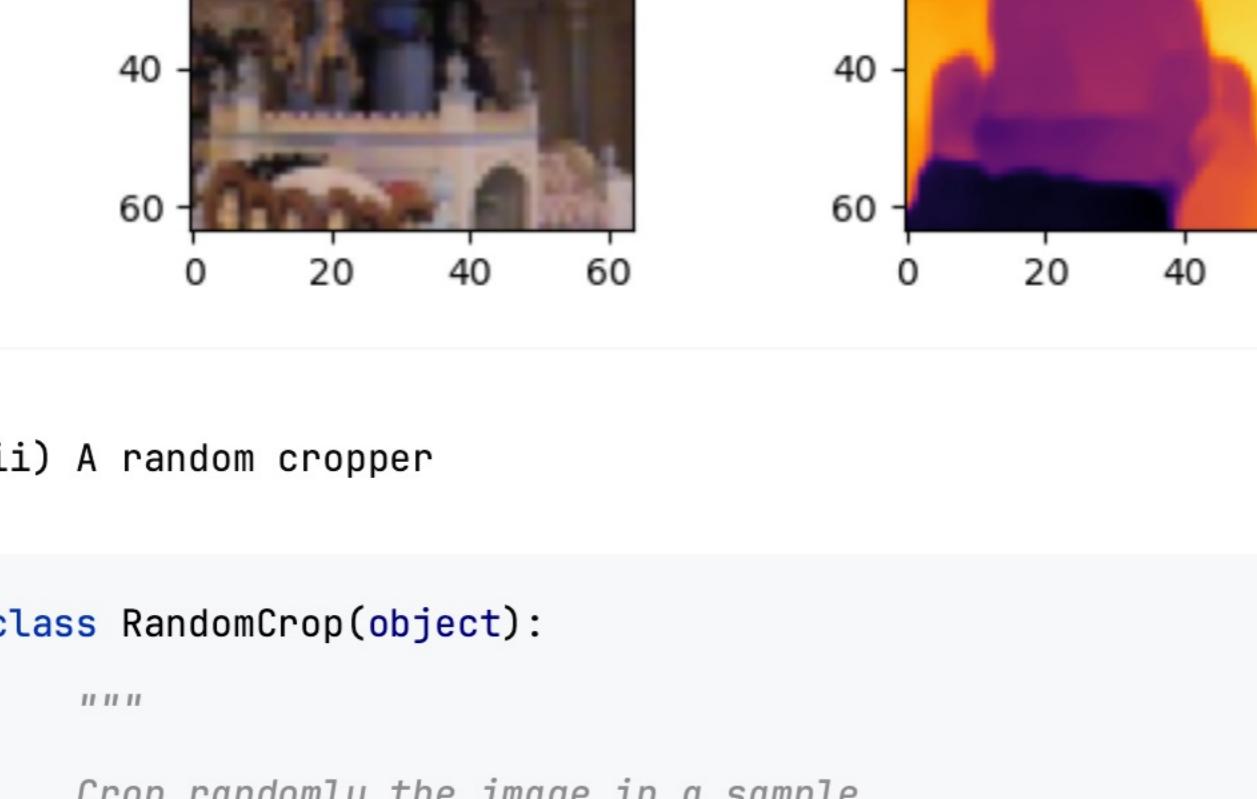
```
In 6 1 # add a simple __len__ method
2 @add_to_class(RedWebDataset)
3 def __len__(self):
4     assert len(self.monocular_images) == len(self.heatmap_images), "Hein?"
5     return len(self.monocular_images)
Executed at 2023.12.01 22:50:33 in 12ms
```

```
In 7 1 # now we can call len on our loader
2 len(loader)
Executed at 2023.12.01 22:50:33 in 10ms
```

Out 7 3600

```
In 8 1 # lets write for practice that makes a subplot which shows
2 # image and their corresponding heatmap on a single subplot
3
4 @add_to_class(RedWebDataset)
5 def _show_sample(self, name = None):
6     if name:
7         name = name if "." not in name else name.split(".")[0]
8     else:
9         # show random index
10        rand_index = np.random.randint(low=0,high = len(self) , size=1).item()
11        name = self.monocular_images[rand_index].split(".")[0]
12
13        # read images
14        mono_img = plt.imread(loader.monocular_folder +"/"+ name + ".jpg")
15        heat_img = plt.imread(loader.heatmap_folder +"/"+ name + ".png")
16        mono_img = io.imread(self.monocular_folder +"/"+ name + ".jpg")
17        heat_img = io.imread(self.heatmap_folder +"/"+ name + ".png")
18
19        fig, ax = plt.subplot_mosaic("AB")
20        ax["A"].set_title(f"Monocular Image({name})", fontsize=8)
21        ax["A"].imshow(mono_img)
22        ax["B"].set_title(f"Heatmap Image({name})", fontsize=8)
23        ax["B"].imshow(heat_img, cmap="inferno")
24        ax["A"].grid(False)
25        ax["A"].axis("off")
26        ax["B"].grid(False)
27        ax["B"].axis("off")
28        plt.show()
Executed at 2023.12.01 22:50:33 in 7ms
```

```
In 9 1 loader._show_sample()
Executed at 2023.12.01 22:50:33 in 96ms
```



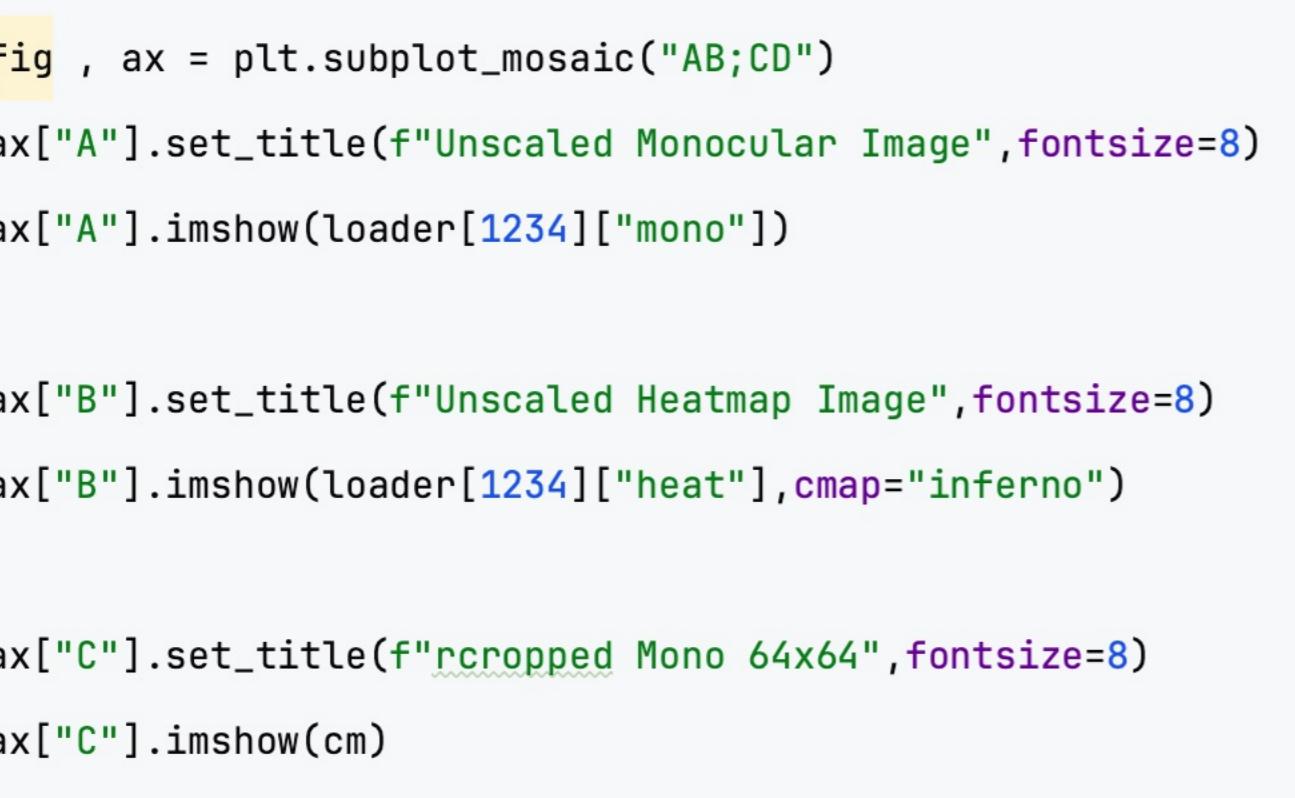
```
In 10 1 loader._show_sample()
Executed at 2023.12.01 22:50:33 in 94ms
```



```
In 11 1 # now lastly just implement __getitem__
2 @add_to_class(RedWebDataset)
3 def __getitem__(self, idx):
4     if torch.is_tensor(idx):
5         idx = idx.tolist()
6
7     monocular_image = io.imread(self.monocular_folder +"/"+ self.monocular_images[idx])
8     heatmap_image = io.imread(self.heatmap_folder +"/"+ self.heatmap_images[idx])
9     sample = {'mono': monocular_image, 'heat': heatmap_image }
10
11     if self.transform:
12         sample = self.transform(sample)
13
14     return sample
Executed at 2023.12.01 22:50:33 in 6ms
```

```
In 12 1 # now we can call index on our loader
2 plt.imshow(loader[1234]["mono"])
Executed at 2023.12.01 22:50:33 in 145ms
```

Out 12 <matplotlib.image.AxesImage at 0x10657f20>



## Now lets Develop some useful data transformation

i) A rescaler

```
In 13 1 class Rescale(object):
2     """
3         simple Rescaler for my monocular and heatmap images
4     """
5     def __init__(self, output_size):
6         assert isinstance(output_size, (int, tuple))
7         self.output_size = output_size
8
9     def __call__(self, sample):
10        mono, heat = sample['mono'], sample['heat']
11        h, w = mono.shape[:2]
12
13        if isinstance(self.output_size, int):
14            if h > w:
15                new_h, new_w = self.output_size * h / w, self.output_size
16            else:
17                new_h, new_w = self.output_size, self.output_size * w / h
18            else:
19                new_h, new_w = self.output_size
20
21        new_h, new_w = int(new_h), int(new_w)
22
23        mono = transform.resize(mono, (new_h, new_w))
24        heat = transform.resize(heat, (new_h, new_w))
25
26
27        return {'mono': mono, 'heat': heat}
Executed at 2023.12.01 22:50:33 in 4ms
```

```
In 14 1 rescaler_64 = Rescale((64,64))
2 scaled = rescaler_64(loader[1234])
3 m64, h64 = scaled["mono"], scaled["heat"]
Executed at 2023.12.01 22:50:33 in 6ms
```

Out 13 <matplotlib.image.AxesImage at 0x10657f20>



ii) A random cropper

```
In 16 1 class RandomCrop(object):
2     """
3         Crop randomly the image in a sample.
4     """
5     def __init__(self, output_size):
6         assert isinstance(output_size, (int, tuple))
7         if isinstance(output_size, int):
8             self.output_size = (output_size, output_size)
9         else:
10            assert len(output_size) == 2
11            self.output_size = output_size
12
13     def __call__(self, sample):
14        mono, heat = sample['mono'], sample['heat']
15
16        h, w = mono.shape[:2]
17        new_h, new_w = self.output_size
18
19        top = np.random.randint(0, h - new_h)
20        left = np.random.randint(0, w - new_w)
21
22        mono = mono[top:top + new_h, left:left + new_w]
23        heat = heat[top:top + new_h, left:left + new_w]
24
25
26        return {'mono': mono, 'heat': heat}
Executed at 2023.12.01 22:50:34 in 9ms
```

```
In 17 1 # a random cropper which randomly crops and returns back an image of 64x64
2 r_cropper = RandomCrop((64,64))
3 cropped = r_cropper(loader[1234])
4 cm, ch = cropped["mono"], cropped["heat"]
Executed at 2023.12.01 22:50:34 in 4ms
```

Out 14 <matplotlib.image.AxesImage at 0x10657f20>



# You can composite transforms composed = (rescale o cropped)(img)

```
In 19 1 scale = Rescale(256)
2 crop = RandomCrop(128)
3 composed = transforms.Compose([Rescale(256),
4                               RandomCrop(128)])
5
6 # Apply each of the above transforms on sample.
7 fig = plt.figure()
8 sample = loader[65] # lets start using a different example now
9 for tsfrm in enumerate([scale, crop, composed]):
10    transformed_sample = tsfrm(sample)
11
12    ax = plt.subplot(1, 3, i + 1)
13    plt.tight_layout()
14    ax.set_title(type(tsfrm).__name__)
15    plt.imshow(transformed_sample["mono"])
16
17    plt.show()
Executed at 2023.12.01 22:50:34 in 239ms
```

Out 15 <matplotlib.image.AxesImage at 0x10657f20>



iii) A random cropper

```
In 16 1 class RandomCrop(object):
2     """
3         Crop randomly the image in a sample.
4     """
5     def __init__(self, output_size):
6         assert isinstance(output_size, (int, tuple))
7         if isinstance(output_size, int):
8             self.output_size = (output_size, output_size)
9         else:
10            assert len(output_size) == 2
11            self.output_size = output_size
12
13     def __call__(self, sample):
14        mono, heat = sample['mono'], sample['heat']
15
16        h, w = mono.shape[:2]
17        new_h, new_w = self.output_size
18
19        top = np.random.randint(0, h - new_h)
20        left = np.random.randint(0, w - new_w)
21
22        mono = mono[top:top + new_h, left:left + new_w]
23        heat = heat[top:top + new_h, left:left + new_w]
24
25
26        return {'mono': mono, 'heat': heat}
Executed at 2023.12.01 22:50:34 in 9ms
```

```
In 17 1 # a random cropper which randomly crops and returns back an image of 64x64
2 r_cropper = RandomCrop((64,64))
3 cropped = r_cropper(loader[1234])
4 cm, ch = cropped["mono"], cropped["heat"]
Executed at 2023.12.01 22:50:34 in 4ms
```

Out 16 <matplotlib.image.AxesImage at 0x10657f20>



iv) A random cropper

```
In 17 1 class RandomCrop(object):
2     """
3         Crop randomly the image in a sample.
4     """
5     def __init__(self, output_size):
6         assert isinstance(output_size, (int, tuple))
7         if isinstance(output_size, int):
8             self.output_size = (output_size, output_size)
9         else:
10            assert len(output_size) == 2
11            self.output_size = output_size
12
13     def __call__(self, sample):
14        mono, heat = sample['mono'], sample['heat']
15
16        h, w = mono.shape[:2]
17        new_h, new_w = self.output_size
18
19        top = np.random.randint(0, h - new_h)
20        left = np.random.randint(0, w - new_w)
21
22        mono = mono[top:top + new_h, left:left + new_w]
23        heat = heat[top:top + new_h, left:left + new_w]
24
25
26        return {'mono': mono, 'heat': heat}
Executed at 2023.12.01 22:50:34 in 9ms
```

Out 17 <matplotlib.image.AxesImage at 0x10657f20>



v) A random cropper

```
In 18 1 class RandomCrop(object):
2     """
3         Crop randomly the image in a sample.
4     """
5     def __init__(self, output_size):
6         assert isinstance(output_size, (int, tuple))
7         if isinstance(output_size, int):
8             self.output_size = (output_size, output_size)
9         else:
10            assert len(output_size) == 2
11            self.output_size = output_size
12
13     def __call__(self, sample):
14        mono, heat = sample['mono'], sample['heat']
15
16        h, w = mono.shape[:2]
17        new_h, new_w = self.output_size
18
19        top = np.random.randint(0, h - new_h)
20        left = np.random.randint(0, w - new_w)
21
22        mono = mono[top:top + new_h, left:left + new_w]
23        heat = heat[top:top + new_h, left:left + new_w]
24
25
26        return {'mono': mono, 'heat': heat}
Executed at 2023.12.01 22:50:34 in 24ms
```

Out 18 <matplotlib.image.AxesImage at 0x10657f20>



vi) A random cropper

```
In 19 1 class RandomCrop(object):
2     """
3         Crop randomly the image in a sample.
4     """
5     def __init__(self, output_size):
6         assert isinstance(output_size, (int, tuple))
7         if isinstance(output_size, int):
8             self.output_size = (output_size, output_size)
9         else:
10            assert len(output_size) == 2
11            self.output_size = output_size
12
13     def __call__(self, sample):
14        mono, heat = sample['mono'], sample['heat']
15
16        h, w = mono.shape[:2]
17        new_h, new_w = self.output_size
18
19        top = np.random.randint(0, h - new_h)
20        left = np.random.randint(0, w - new_w)
21
22        mono = mono[top:top + new_h, left:left + new_w]
23        heat = heat[top:top + new_h, left:left + new_w]
24
25
26        return {'mono': mono, 'heat': heat}
Executed at 2023.12.01 22:50:34 in 24ms
```

Out 19 <matplotlib.image.AxesImage at 0x10657f20>



vii) A random cropper

```
In 20 1 class RandomCrop(object):
2     """
3         Crop randomly the image in a sample.
4     """
5     def __init__(self, output_size):
6         assert isinstance(output_size, (int, tuple))
7         if isinstance(output_size, int):
8             self.output_size = (output_size, output_size)
9         else:
10            assert len(output_size) == 2
11            self.output_size = output_size
12
13     def __call__(self, sample):
14        mono, heat = sample['mono'], sample['heat']
15
16        h, w = mono.shape[:2]
17        new_h, new_w = self.output_size
18
19        top = np.random.randint(0, h - new_h)
20        left = np.random.randint(0, w - new_w)
21
22        mono = mono[top:top + new_h, left:left + new_w]
23        heat = heat[top:top + new_h, left:left + new_w]
24
25
26        return {'mono': mono, 'heat': heat}
Executed at 2023.12.01 22:50:34 in 24ms
```

Out 20 <matplotlib.image.AxesImage at 0x10657f20>



viii) A random cropper

```
In 21 1 class RandomCrop(object):
2     """
3         Crop randomly the image in a sample.
4     """
5     def __init__(self, output_size):
6         assert isinstance(output_size, (int, tuple))
7         if isinstance(output_size, int):
8             self.output_size = (output_size, output_size)
9         else:
10            assert len(output_size) == 2
11            self.output_size = output_size
12
13     def __call__(self, sample):
14        mono, heat = sample['mono'], sample['heat']
15
16        h, w = mono.shape[:2]
17        new_h, new_w = self.output_size
18
19        top = np.random.randint(0, h - new_h)
20        left = np.random.randint(0, w - new_w)
21
22        mono = mono[top:top + new_h, left:left + new_w]
23        heat = heat[top:top + new_h, left:left + new_w]
24
25
26        return {'mono': mono, 'heat': heat}
Executed at 2023.12.01 22:50:34 in 24ms
```

Out 21 <matplotlib.image.AxesImage at 0x10657f20>



vix) A random cropper

```
In 22 1 class RandomCrop(object):
2     """
3         Crop randomly the image in a sample.
4     """
5     def __init__(self, output_size):
6         assert isinstance(output_size, (int, tuple))
7         if isinstance(output_size, int):
8             self.output_size = (output_size, output_size)
9         else:
10            assert len(output_size) == 2
11            self.output_size = output_size
12
13     def __call__(self, sample):
14        mono, heat = sample['mono'], sample['heat']
15
16        h, w = mono.shape[:2]
17        new_h, new_w = self.output_size
18
19        top = np.random.randint(0, h - new_h)
20        left = np.random.randint(0, w - new_w)
21
22        mono = mono[top:top + new_h, left:left + new_w]
23        heat = heat[top:top + new_h, left:left + new_w]
24
25
26        return {'mono': mono, 'heat': heat}
Executed at 2023.12.01 22:50:34 in 24ms
```

Out 22 <matplotlib.image.AxesImage at 0x10657f20>



vixi) A random cropper

```
In 23 1 class RandomCrop(object):
2     """
3         Crop randomly the image in a sample.
4     """
5     def __init__(self, output_size):
6         assert isinstance(output_size, (int, tuple))
7         if isinstance(output_size, int):
8             self.output_size = (output_size, output_size)
9         else:
10            assert len(output_size) == 2
11            self.output_size = output_size
12
13     def __call__(self, sample):
14        mono, heat = sample['mono'], sample['heat']
15
16        h, w = mono.shape[:2]
17        new_h, new_w = self.output_size
18
19        top = np.random.randint(0, h - new_h)
20        left = np.random.randint(0, w - new_w)
21
22        mono = mono[top:top + new_h, left:left + new_w]
23        heat = heat[top:top + new_h, left:left + new_w]
24
25
26        return {'mono': mono, 'heat': heat}
Executed at 2023.12.01 22:50:34 in 24ms
```

```
8 sample = loader[65] # lets start using a different example now
9 for i, tsfrm in enumerate([scale, crop, composed]):
10     transformed_sample = tsfrm(sample)
11
12     ax = plt.subplot(1, 3, i + 1)
13     plt.tight_layout()
14     ax.set_title(type(tsfrm).__name__)
15     plt.imshow(transformed_sample["mono"])
16
17 plt.show()
```

Executed at 2023.12.01 22:50:34 in 235ms



```
In 20 1 # a simple image to tensor class
2 class ToTensor(object):
3     """Convert ndarrays in sample to Tensors."""
4
5     def __call__(self, sample):
6         mono, heat = sample['mono'], sample['heat']
7
8         # for monocular image :
9         # swap color axis because
10        # numpy image: H x W x C
11        # torch image: C X H X W
12        mono = mono.transpose((2, 0, 1))
13
14        # dont have to do anything for single channel image
15
16        return {'mono': torch.from_numpy(mono),
17                'heat': torch.from_numpy(heat)}
```

Executed at 2023.12.01 22:50:34 in 4ms

```
In 21 1 to_tensor = ToTensor()
2 tensored= to_tensor(loader[65])
3 print(type(tensored["mono"]) , type(tensored["heat"]))
4
5 Executed at 2023.12.01 22:50:34 in 7ms
```

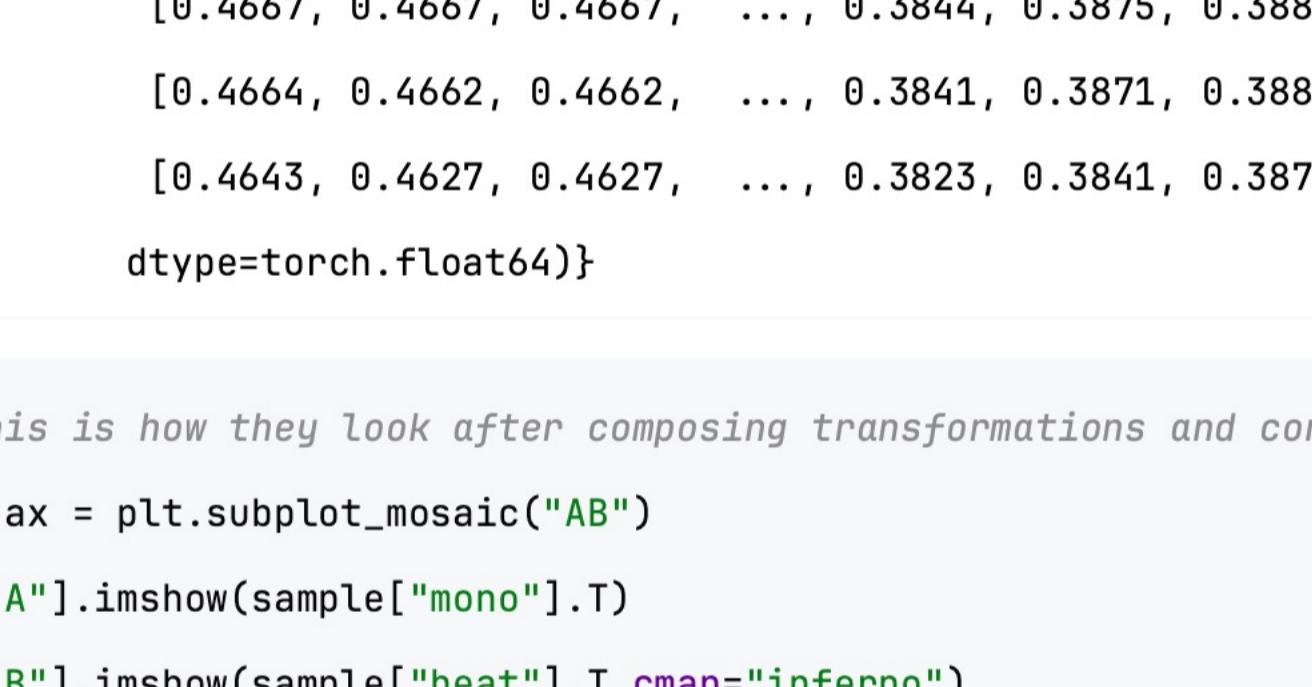
<class 'torch.Tensor'> <class 'torch.Tensor'>

```
In 22 1 # and their shape also must be the same in height and width .
2 # (channel , height , width) ← torch.img
3 print(tensored["mono"].shape , tensored["heat"].shape)
4
5 Executed at 2023.12.01 22:50:34 in 12ms
```

torch.Size([3, 305, 222]) torch.Size([305, 222])

```
In 23 1 # but that shouldnt change anything .pairwise
2 # need to plot its transpose as plt.subplots does not take channel as its first dimension
3 fig,ax = plt.subplot_mosaic("AB")
4 ax["A"].imshow(tensored["mono"].T)
5 ax["B"].imshow(tensored["heat"].T,cmap="inferno")
6
7 Executed at 2023.12.01 22:50:34 in 109ms
```

Out 23 <matplotlib.image.AxesImage at 0x16b570d30>



```
In 24 1 # and now we can iterate through the dataset combining all the operations we have made thus far
2 transformed_dataset = RedWebDataset(root_dir="../ReDWeb_V1",transform=transforms.Compose([
3     Rescale(256),
4     RandomCrop(225),
5     ToTensor()
6 ]))
7
8 sample = transformed_dataset[99]
9 sample
10
11 Executed at 2023.12.01 22:50:34 in 36ms
```

Out 24 'heat': tensor([[0.8392, 0.8392, 0.8392, ..., 0.9804, 0.9804, 0.9804],

[0.8392, 0.8392, 0.8392, ..., 0.9784, 0.9800, 0.9804],

[0.8392, 0.8392, 0.8392, ..., 0.9765, 0.9776, 0.9779],

...,

[0.4667, 0.4667, 0.4667, ..., 0.3844, 0.3875, 0.3883],

[0.4664, 0.4662, 0.4662, ..., 0.3841, 0.3871, 0.3882],

[0.4643, 0.4627, 0.4627, ..., 0.3823, 0.3841, 0.3870]],

dtype=torch.float64)}

In 25 1 # this is how they look after composing transformations and converting them into tensor objects

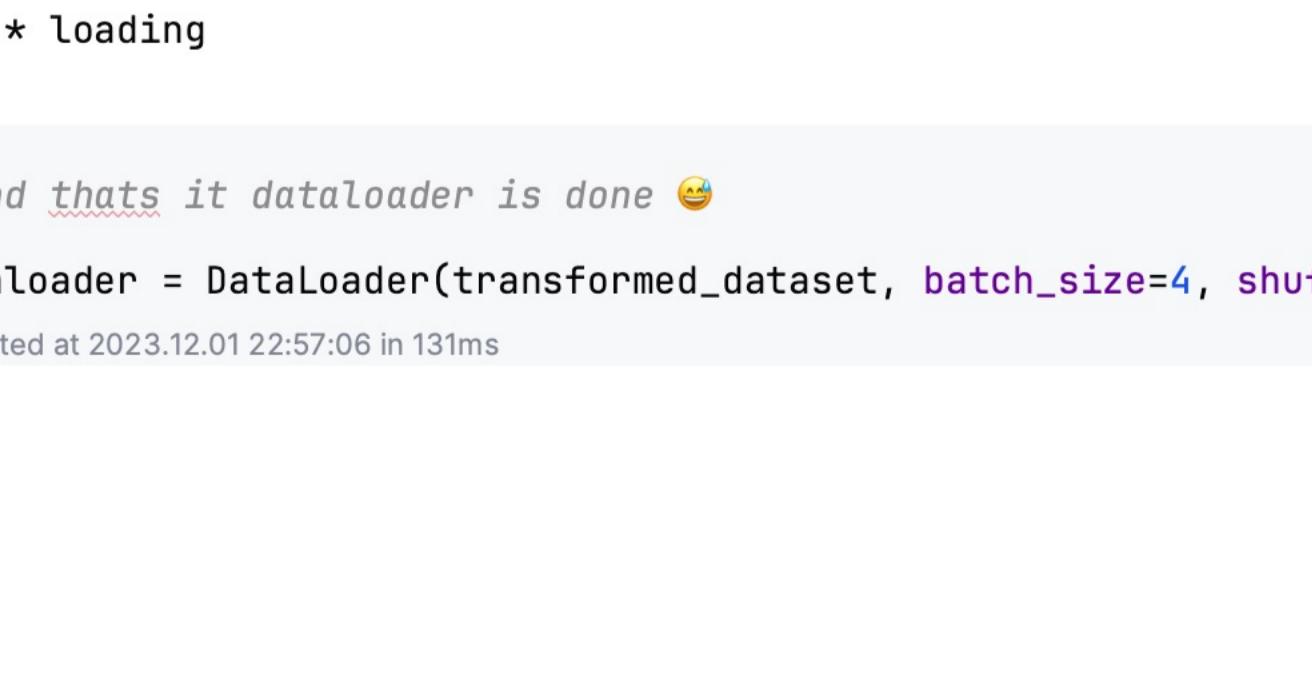
2 fig,ax = plt.subplot\_mosaic("AB")

3 ax["A"].imshow(sample["mono"].T)

4 ax["B"].imshow(sample["heat"].T,cmap="inferno")

5
6 Executed at 2023.12.01 22:50:34 in 119ms

Out 25 <matplotlib.image.AxesImage at 0x16b7b6e00>



# Now lets get to the crux of the dataloader

- \* Batching
- \* shuffling
- \* loading

In 26 1 # and thats it dataloader is done 😊

2 dataloader = DataLoader(transformed\_dataset, batch\_size=4, shuffle=True, num\_workers=4) # but an iterable . cant call getitem on this obviously

3
4 Executed at 2023.12.01 22:57:06 in 131ms

We would be overwriting the `getitem` method of our original Dataloader to incorporate Online Batch Sampling  
We go on detail of its working in a seperate notebook in DepthSense/online\_sampling  
Check out that notebook/pdf before reading this

```
In 2 1 from DataLoader import RedWebDataset , Rescale , RandomCrop , ToTensor
2 from __future__ import print_function, division
3 import os
4 import torch
5 from skimage import io, transform
6 import random
7 import matplotlib.pyplot as plt
8 from torch.utils.data import Dataset, DataLoader
9 from torchvision import transforms, utils
Executed at 2023.12.02 17:20:51 in 8ms
```

```
In 4 1 # from my utils notebook
2 def add_to_class(Class):
3     """Register them functions"""
4     def wrapper(obj):
5         # setattr(object, name, value) → sets the value of the attribute
6         setattr(Class, obj.__name__, obj)
7     return wrapper
Executed at 2023.12.02 17:22:46 in 10ms
```

```
In 80 1 class OnlineRedWeb(RedWebDataset):
2     def __init__(self, root_dir ,transform=None,N:int=10,sigma:int=0.02):
3         super().__init__(root_dir,transform)
4         self.N =N
5         self.sigma = 0.02
6
7     def __getitem__(self, index):
8         # Call the __getitem__ method of the original dataset
9         original_item = super(OnlineRedWeb, self).__getitem__(index)
10
11         # Extract necessary information from the original_item
12         mono = original_item['mono']
13         heat = original_item['heat']
14
15         # Implement online sampling logic here
16         height, width = heat.shape
17         point_a = []
18         point_b = []
19         labels = []
20
21         for _ in range(self.N):
22             i, j = random.randint(0, height-1), random.randint(0, width-1)
23             k,l = random.randint(0, height-1), random.randint(0, width-1)
24
25             ga, gb = heat[i, j], heat[k, l] # Assuming heat is a 2D tensor
26
27             if ga/ gb > 1 + self.sigma:
28                 label = 1
29             elif ga/ gb < 1 - self.sigma:
30                 label = -1
31             else:
32                 label = 0
33
34             point_a.append((i,j))
35             point_b.append((k,l))
36             labels.append(label)
37
38         # Update the original_item or create a new dictionary to return
39         online_item = {'mono': mono,
40                         'heat': heat,
41                         'point_a':torch.tensor(point_a),"point_b": torch.tensor(point_b),
42                         'labels': torch.tensor(labels)}
43
44     return online_item
Executed at 2023.12.02 18:25:00 in 4ms
```

```
In 81 1 @add_to_class(OnlineRedWeb)
```

```
2 def online_collater(batch):
```

```
3     mono = torch.stack([item['mono'] for item in batch])
```

```
4     heat = torch.stack([item['heat'] for item in batch])
```

```
5     point_a = [item['point_a'] for item in batch]
```

```
6     point_b = [item['point_b'] for item in batch]
```

```
7     labels = [item['labels'] for item in batch]
```

```
8     return{'mono': mono, 'heat': heat,
```

```
9         'point_a':point_a , "point_b":point_b,
```

```
10        'labels': labels}
```

```
11 Executed at 2023.12.02 18:25:01 in 4ms
```

```
In 82 1 # we can still call the old methods on this class
```

```
2 loader = OnlineRedWeb(root_dir="../ReDWeb_V1")
```

```
3 loader._show_sample()
```

```
Executed at 2023.12.02 18:25:01 in 163ms
```

Monocular Image9371097114\_ac36d386c6\_h Heatmap Image9371097114\_ac36d386c6\_h



SSL\_Loss.ipynb x DataLoader.py

Managed: http://localhost:8850 Python 3 (ipykernel) Trusted

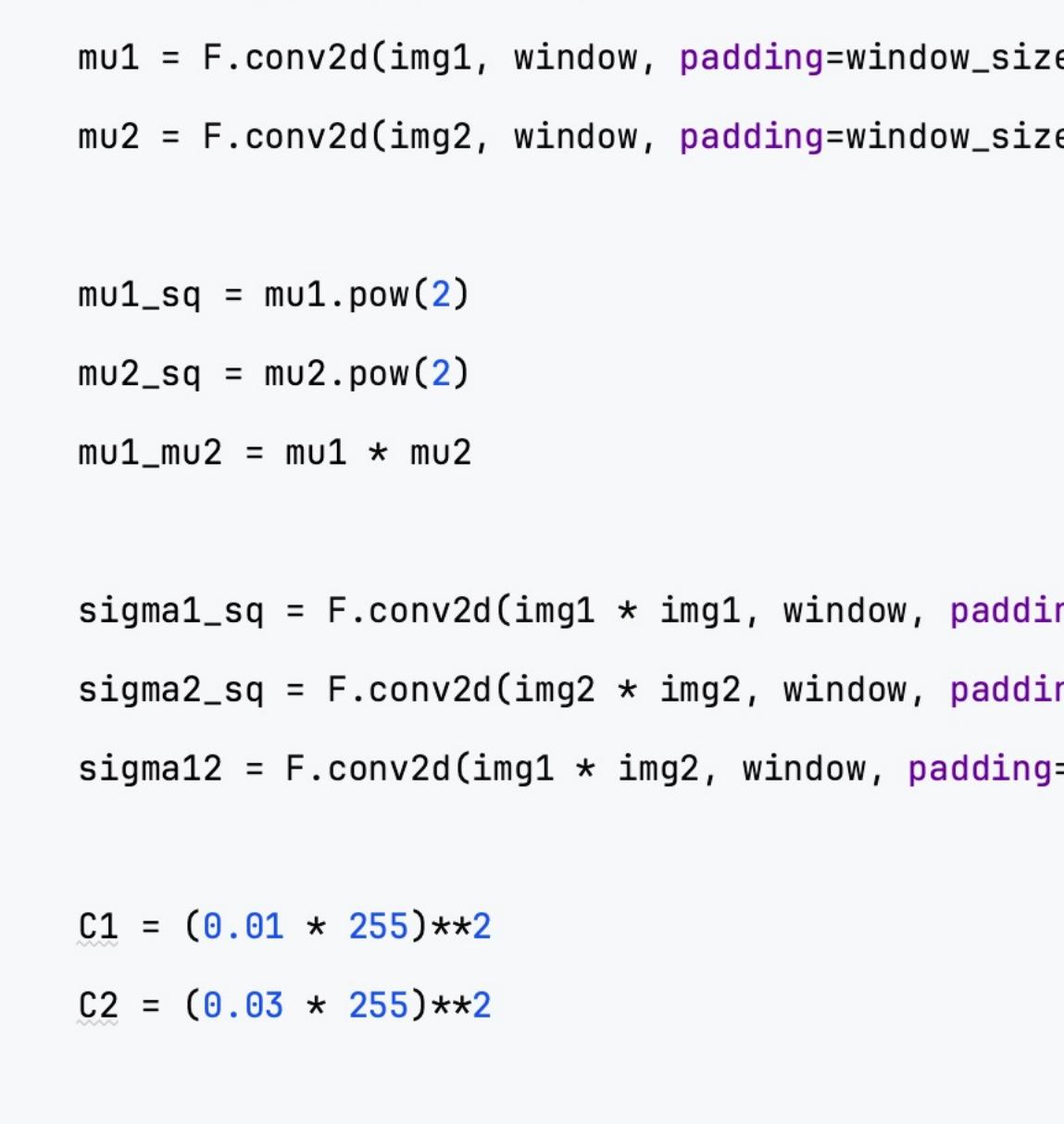
```
In 1 1 import numpy as np
2 import torch
3 import torch.nn.functional as F
4 from torchvision import transforms
5 from torch.utils.data import Dataset, Dataloader
6 # from skimage.metrics import structural_similarity as ssim
7 # from data.loaders.OnlineLoader import OnlineRedWeb
8 from data.loaders.DataLoader import RedWebDataset, Rescale, RandomCrop, ToTensor
Executed at 2023.12.03 13:34:17 in 209ms
> /Users/adi/opt/anaconda3/lib/python3.10/site-packages/torchvision/io/image.py:13: UserWarning: Failed to load image Pyth
```

```
In 2 1 normal_dataset = RedWebDataset(root_dir="..//data/RedWeb_V1", transform=transforms.Compose([
2     Rescale(256),
3     RandomCrop(225),
4     ToTensor()
5 ]))
6 batcher = Dataloader(normal_dataset, batch_size=1, shuffle=True)
Executed at 2023.12.03 13:34:17 in 43ms
```

```
In 3 1 for sample1 in batcher:
2     print(sample1["mono"].shape)
3     print(sample1["heat"].shape)
4     break
5
6 for sample2 in batcher:
7     print(sample2["mono"].shape)
8     print(sample2["heat"].shape)
9     break
10
Executed at 2023.12.03 13:34:17 in 83ms
```

```
< torch.Size([1, 3, 225, 225])
```

```
In 4 1 import matplotlib.pyplot as plt
2 fig, ax = plt.subplot_mosaic("AB;CD")
3 ax["A"].imshow(sample1["mono"][:, :].T)
4 ax["C"].imshow(sample1["heat"][:, :].T)
5
6
7 ax["B"].imshow(sample2["mono"][:, :].T)
8 ax["D"].imshow(sample2["heat"][:, :].T)
9
10 for k in ax.keys():
11     ax[k].axis("off")
12     ax[k].grid("off")
13
Executed at 2023.12.03 13:34:17 in 91ms
> /var/folders/f0/_f_gy0t91jqchv14f0hcl_gh000gn/T/ipykernel_94717/2951813992.py:3: UserWarning: The use of `x.T` on tensor
```



```
# lets try to calculate the aggregate similarity score of these pairwise map
```

```
In 5 1
2 def gaussian(window_size, sigma):
3     gauss = torch.exp(-(torch.arange(window_size) - window_size // 2)**2 / (2.0 * sigma**2))
4     return gauss / gauss.sum()
5
6 def create_window(window_size, channel):
7     _1D_window = gaussian(window_size, 1.5).unsqueeze(1)
8     _2D_window = _1D_window.mm(_1D_window.t()).float().unsqueeze(0).unsqueeze(0)
9     window = _2D_window.expand(channel, 1, window_size, window_size).contiguous()
10    return window
11
12 def _ssim(img1, img2, window, window_size, channel, size_average=True):
13     mu1 = F.conv2d(img1, window, padding=window_size // 2, groups=channel)
14     mu2 = F.conv2d(img2, window, padding=window_size // 2, groups=channel)
15
16     mu1_sq = mu1.pow(2)
17     mu2_sq = mu2.pow(2)
18     mu1_mu2 = mu1 * mu2
19
20     sigma1_sq = F.conv2d(img1 * img1, window, padding=window_size // 2, groups=channel) - mu1_sq
21     sigma2_sq = F.conv2d(img2 * img2, window, padding=window_size // 2, groups=channel) - mu2_sq
22     sigma12 = F.conv2d(img1 * img2, window, padding=window_size // 2, groups=channel) - mu1_mu2
23
24     C1 = (0.01 * 255)**2
25     C2 = (0.03 * 255)**2
26
27     ssim_map = ((2 * mu1_mu2 + C1) * (2 * sigma12 + C2)) / ((mu1_sq + mu2_sq + C1) * (sigma1_sq + sigma2_sq + C2))
28
29     if size_average:
30         return ssim_map.mean()
31     else:
32         return ssim_map.mean(1).mean(1).mean(1)
33
34 class SSIM(torch.nn.Module):
35     def __init__(self, window_size=11, size_average=True):
36         super(SSIM, self).__init__()
37         self.window_size = window_size
38         self.size_average = size_average
39         self.channel = 1
40         self.register_buffer('window', create_window(window_size, self.channel))
41
42     def forward(self, img1, img2):
43         _, channel, _, _ = img1.size()
44
45         if channel == self.channel and self.window.type() == img1.type():
46             window = self.window
47         else:
48             window = create_window(self.window_size, channel)
49             window = window.to(img1.device).type_as(img1)
50             self.window = window
51             self.channel = channel
52
53         return _ssim(img1, img2, window, self.window_size, channel, self.size_average)
54
55 def ssim(img1, img2, window_size=11, size_average=True):
56     _, channel, _, _ = img1.size()
57     window = create_window(window_size, channel)
58     window = window.to(img1.device).type_as(img1)
59     return _ssim(img1, img2, window, window_size, channel, size_average)
60
Executed at 2023.12.03 13:34:17 in 1ms
```

```
In 6 1 ssim_loss = SSIM()
Executed at 2023.12.03 13:34:17 in 39ms
```

```
In 7 1 ssim_loss(sample1["mono"], sample1["heat"])
Executed at 2023.12.03 13:34:18 in 274ms
```

```
Out 7 tensor(0.9881, dtype=torch.float64)
```

```
In 8 1 ssim_loss(sample2["mono"], sample2["heat"])
Executed at 2023.12.03 13:34:18 in 217ms
```

```
Out 8 tensor(0.9762, dtype=torch.float64)
```

```
In 1 1 import matplotlib.pyplot as plt
2 import torch
3 from torchvision.io import read_image
4 from torchvision import transforms
5 import torch.nn as nn
6 import torch.nn.functional as F
7
8
9 from torch.utils.data import DataLoader
Executed at 2023.12.04 20:06:38 in 1s 590ms
```

```
class InverseDepthSmoothnessLoss [source]
```

Criterion that computes image-aware inverse depth smoothness loss.

$$\text{loss} = |\partial_x d_{ij}| e^{-\|\partial_x I_{ij}\|} + |\partial_y d_{ij}| e^{-\|\partial_y I_{ij}\|}$$

Shape:

- Inverse Depth:  $(N, 1, H, W)$
- Image:  $(N, 3, H, W)$
- Output: scalar

```
In 2 1 from data.loaders.DataLoader import RedWebDataset, Rescale, RandomCrop
2 normal_dataset = RedWebDataset(root_dir="data/ReDWeb_V1", transform=transforms.Compose([
3     Rescale((256, 256)),
4 ])
5 batcher = DataLoader(normal_dataset, batch_size=1, shuffle=True)
Executed at 2023.12.04 20:06:38 in 7ms
```

```
In 3 1 batched = iter(batcher)
2 sample0 = next(batched)
3 print(sample0["mono"].shape)
4 print(sample0["heat"].shape)
5
6 sample1 = next(batched)
7 print(sample1["mono"].shape)
8 print(sample1["heat"].shape)
9
Executed at 2023.12.04 20:06:38 in 12ms
```

```
<torch.Size([1, 3, 256, 256])>
```

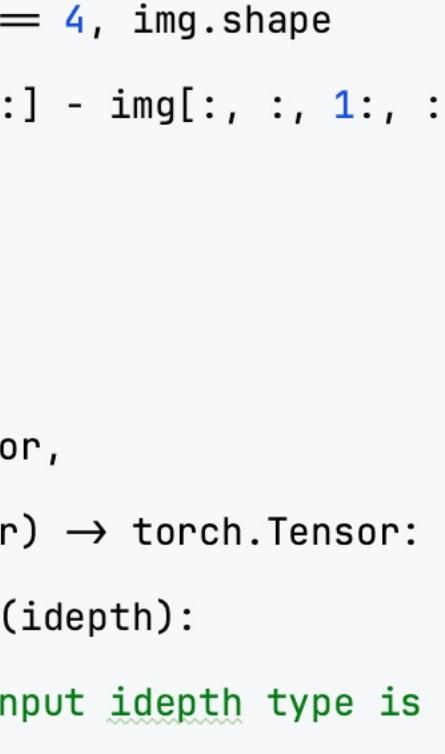
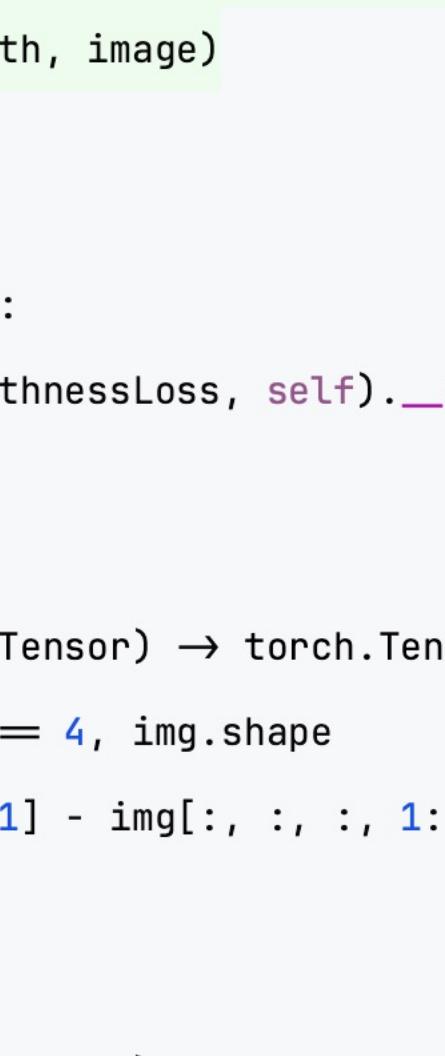
```
<torch.Size([1, 1, 256, 256])>
```

```
<torch.Size([1, 3, 256, 256])>
```

```
<torch.Size([1, 1, 256, 256])>
```

```
> /Users/adi/opt/anaconda3/lib/python3.10/site-packages/torchvision/transforms/functional.py:1603: UserWarning: The defaul
```

```
In 4 1 # lets see these images
2 import matplotlib.pyplot as plt
3 fig, ax = plt.subplot_mosaic("AB;CD")
4 ax["A"].imshow(sample0["mono"].squeeze(0).T)
5 ax["C"].imshow(sample1["mono"].squeeze(0).T)
6
7 ax["B"].imshow(sample0["heat"].squeeze(0).T, cmap="inferno")
8 ax["D"].imshow(sample1["heat"].squeeze(0).T, cmap="inferno")
9
10 for k in ax.keys():
11     ax[k].grid("off")
12     ax[k].axis("off")
13
14
15 Executed at 2023.12.04 20:06:38 in 93ms
```



```
In 5 1 # Based on
# https://github.com/tensorflow/models/blob/master/research/struct2depth/model.py#L625-L641
```

```
2
3
4
5 class InverseDepthSmoothnessLoss(nn.Module):
6     r"""Criterion that computes image-aware inverse depth smoothness loss.
```

```
7         ... math::
```

$$\text{loss} = \left| \partial_x I_{ij} \right| e^{-\|\partial_x I_{ij}\|} + \left| \partial_y I_{ij} \right| e^{-\|\partial_y I_{ij}\|}$$

```
15 Shape:
```

```
- Inverse Depth: :math:`(N, 1, H, W)`
```

```
- Image: :math:`(N, 3, H, W)`
```

```
- Output: scalar
```

```
19 Examples::
```

```
22     >>> idepth = torch.rand(1, 1, 4, 5)
```

```
23     >>> image = torch.rand(1, 3, 4, 5)
```

```
24     >>> smooth = tgm.losses.DepthSmoothnessLoss()
```

```
25     >>> loss = smooth(idepth, image)
```

```
26     nnn
```

```
28     def __init__(self) -> None:
```

```
29         super(InverseDepthSmoothnessLoss, self).__init__()
```

```
31     @staticmethod
```

```
32     def gradient_x(img: torch.Tensor) -> torch.Tensor:
```

```
33         assert len(img.shape) == 4, img.shape
```

```
34         return img[:, :, :, :-1] - img[:, :, :, 1:]
```

```
36     @staticmethod
```

```
37     def gradient_y(img: torch.Tensor) -> torch.Tensor:
```

```
38         assert len(img.shape) == 4, img.shape
```

```
39         return img[:, :, :-1, :] - img[:, :, 1:, :]
```

```
41     def forward(

```

```
42         self,

```

```
43         idepth: torch.Tensor,

```

```
44         image: torch.Tensor) -> torch.Tensor:

```

```
45         if not torch.is_tensor(idepth):

```

```
46             raise TypeError("Input idepth type is not a torch.Tensor. Got {}"

```

```
47                 .format(type(idepth)))

```

```
48         if not torch.is_tensor(image):

```

```
49             raise TypeError("Input image type is not a torch.Tensor. Got {}"

```

```
50                 .format(type(image)))

```

```
51         if not len(idepth.shape) == 4:

```

```
52             raise ValueError("Invalid idepth shape, we expect BxCxHxW. Got: {}"

```

```
53                 .format(idepth.shape))

```

```
54         if not len(image.shape) == 4:

```

```
55             raise ValueError("Invalid image shape, we expect BxCxHxW. Got: {}"

```

```
56                 .format(image.shape))

```

```
57         if not idepth.shape[-2:] == image.shape[-2:]:

```

```
58             raise ValueError("idepth and image shapes must be the same. Got: {}"

```

```
59                 .format(idepth.shape, image.shape))

```

```
60         if not idepth.device == image.device:

```

```
61             raise ValueError(

```

```
62                 "idepth and image must be in the same device. Got: {}" .format(

```

```
63                     idepth.device, image.device))

```

```
64         if not idepth.dtype == image.dtype:

```

```
65             raise ValueError(

```

```
66                 "idepth and image must be in the same dtype. Got: {}" .format(

```

```
67                     idepth.dtype, image.dtype))

```

```
68         # compute the gradients

```

```
69         idepth_dx: torch.Tensor = self.gradient_x(idepth)

```

```
70         idepth_dy: torch.Tensor = self.gradient_y(idepth)

```

```
71         image_dx: torch.Tensor = self.gradient_x(image)

```

```
72         image_dy: torch.Tensor = self.gradient_y(image)

```

```
73

```

```
74         # compute image weights

```

```
75         weights_x: torch.Tensor = torch.exp(

```

```
76             -torch.mean(torch.abs(image_dx), dim=1, keepdim=True))

```

```
77         weights_y: torch.Tensor = torch.exp(

```

```
78             -torch.mean(torch.abs(image_dy), dim=1, keepdim=True))

```

```
79

```

```
80         # apply image weights to depth

```

```
81         smoothness_x: torch.Tensor = torch.abs(idepth_dx * weights_x)

```

```
82         smoothness_y: torch.Tensor = torch.abs(idepth_dy * weights_y)

```

```
83

```

```
84         # compute the mean

```

```
85         inv_loss = torch.mean(smoothness_x) + torch.mean(smoothness_y)

```

```
86

```

```
87 Executed at 2023.12.04 20:06:38 in 1ms
```

```
In 6 1 sample0["mono"] = sample0["mono"].to(torch.float)
2 sample0["heat"] = sample0["heat"].to(torch.float)
3
4 sample1["mono"] = sample1["mono"].to(torch.float)
5 sample1["heat"] = sample1["heat"].to(torch.float)
6
7 Executed at 2023.12.04 20:06:38 in 5ms
```

```
In 7 1 inv_loss = InverseDepthSmoothnessLoss()
2 inv_loss(sample0["heat"], sample0["mono"])
3
4 Executed at 2023.12.04 20:06:38 in 20ms
```

```
5 tensor(0.0443)
```

```
In 8 1 inv_loss = InverseDepthSmoothnessLoss()
2 inv_loss(sample0["heat"], sample1["mono"])
3
4 Executed at 2023.12.04 20:06:38 in 9ms
```

```
5 tensor(0.4271)
```

```
In 1 # So we will combine the ideas from the losses we have experimented with
2 # 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
3
4 # 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 1 import torch
2 from torch import nn
3 import numpy as np
4 import torch.nn.functional as F
5 from torchvision import transforms
6
7 from data.loaders.DataLoader import RedWebDataset , Rescale , RandomCrop
8 from torch.utils.data import DataLoader
9
10 normal_dataset = RedWebDataset(root_dir="../data/ReDWeb_V1",transform=transforms.Compose([
11     Rescale((256,256)),
12 ]))
13 batcher = DataLoader(normal_dataset,batch_size=1,shuffle=True)
Executed at 2023.12.10 18:21:34 in 9ms

In 6 1 # Lets Device our online sampling first
2
3 # Note this is derived completely from the Redweb paper
4 def onlineSampling(inputs, targets, masks, threshold, sample_num):
5
6     # find A-B point pairs from predictions (mostly random)
7     inputs_index = torch.masked_select(inputs, targets.gt(threshold))
8     num_effect_pixels = len(inputs_index)
9     shuffle_effect_pixels = torch.randperm(num_effect_pixels).cuda()
10    rgb_a = inputs_index[shuffle_effect_pixels[0:sample_num*2:2]]
11    rgb_b = inputs_index[shuffle_effect_pixels[1:sample_num*2:2]]
12
13    # find corresponding pairs from ground truth
14    depth_index = torch.masked_select(targets, targets.gt(threshold))
15    depth_a = depth_index[shuffle_effect_pixels[0:sample_num*2:2]]
16    depth_b = depth_index[shuffle_effect_pixels[1:sample_num*2:2]]
17
18    # only compute the losses of point pairs with valid ground truth i.e consistent masked
19    consistent_masks_index = torch.masked_select(masks, targets.gt(threshold))
20    consistent_masks_A = consistent_masks_index[shuffle_effect_pixels[0:sample_num*2:2]]
21    consistent_masks_B = consistent_masks_index[shuffle_effect_pixels[1:sample_num*2:2]]
22
23    # The amount of A and B should be the same!!
24    if len(depth_a) > len(depth_b):
25        depth_a = depth_a[:-1]
26        rgb_a = rgb_a[:-1]
27        consistent_masks_A = consistent_masks_A[:-1]
28
29    return rgb_a, rgb_b, depth_a, depth_b, consistent_masks_A, consistent_masks_B
30
Executed at 2023.12.10 18:32:20 in 2ms

In 7 1 # now lets penalize wrong edges
2 # i.e if the edges derived from the depth map does not map with the edges derived from the original image. there should be a corresponding
penalty for it.
3
4 # convenience wrapper function to get pixels
5 def ind2sub(idx, cols):
6     r = idx / cols
7     c = idx - r * cols
8     return r, c
9
10
11 def sub2ind(r, c, cols):
12     idx = r * cols + c
13     return idx
14
15
Executed at 2023.12.10 19:26:27 in 5ms

In 8 1 def edgeGuidedSampling(inputs, targets, edges_img, thetas_img, masks, h, w):
2
3     # find edges
4     edges_max = edges_img.max()
5     edges_mask = edges_img.ge(edges_max*0.1)
6     edges_loc = edges_mask.nonzero()
7
8     inputs_edge = torch.masked_select(inputs, edges_mask)
9     targets_edge = torch.masked_select(targets, edges_mask)
10    thetas_edge = torch.masked_select(thetas_img, edges_mask)
11    minlen = inputs_edge.size()[0]
12
13    # find anchor points (i.e, edge points)
14    sample_num = minlen
15    index_anchors = torch.randint(
16        0, minlen, (sample_num,), dtype=torch.long).cuda()
17    anchors = torch.gather(inputs_edge, 0, index_anchors)
18    theta_anchors = torch.gather(thetas_edge, 0, index_anchors)
19    row_anchors, col_anchors = ind2sub(edges_loc[index_anchors].squeeze(1), w)
20
21    # compute the coordinates of 4-points, distances are from {2, 30}
22    distance_matrix = torch.randn(2, 31, (4, sample_num)).cuda()
23    pos_or_neg = torch.ones(4, sample_num).cuda()
24    pos_or_neg[:, :] = -pos_or_neg[:, :]
25    distance_matrix = distance_matrix.float() * pos_or_neg
26    col = col_anchors.unsqueeze(0).expand(4, sample_num).long(
27        ) + torch.round(distance_matrix.double()) * torch.cos(theta_anchors).unsqueeze(0).long()
28    row = row_anchors.unsqueeze(0).expand(4, sample_num).long(
29        ) + torch.round(distance_matrix.double()) * torch.sin(theta_anchors).unsqueeze(0).long()
30
31    # constrain 0=<c<w, 0=<r<h
32    # Note: index should minus 1
33    col[col < 0] = 0
34    col[col > w-1] = w-1
35    row[row < 0] = 0
36    row[row > h-1] = h-1
37
38    # a-b, b-c, c-d
39    a = sub2ind(row[0, :], col[0, :], w)
40    b = sub2ind(row[1, :], col[1, :], w)
41    c = sub2ind(row[2, :], col[2, :], w)
42    d = sub2ind(row[3, :], col[3, :], w)
43    A = torch.cat((a, b, c), 0)
44    B = torch.cat((b, c, d), 0)
45
46    rgb_a = torch.gather(inputs, 0, A.long())
47    rgb_b = torch.gather(inputs, 0, B.long())
48    depth_a = torch.gather(targets, 0, A.long())
49    depth_b = torch.gather(targets, 0, B.long())
50    masks_A = torch.gather(masks, 0, A.long())
51    masks_B = torch.gather(masks, 0, B.long())
52
53    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 1 class EdgeguidedRankingLoss(nn.Module):
2     def __init__(self, point_pairs=10000, sigma=0.03, alpha=1.0, mask_value=-1e-8):
3         super(EdgeguidedRankingLoss, self).__init__()
4         self.point_pairs = point_pairs # number of point pairs
5         self.sigma = sigma # used for determining the ordinal relationship between a selected pair
6         self.alpha = alpha # used for balancing the effect of = and <, >
7         self.mask_value = mask_value
8         # self.regularization_loss = GradientLoss(scales=4)
9
10
11     def getEdge(self, images):
12         n, c, h, w = images.size()
13         a = torch.Tensor([-1, 0, 1], [-2, 0, 2], [-1, 0, 1])
14             .cuda().view(1, 1, 3, 3)).repeat(1, 1, 1, 1)
15         b = torch.Tensor([1, 2, 1], [0, 0, 0], [-1, -2, -1])
16             .cuda().view(1, 1, 3, 3)).repeat(1, 1, 1, 1)
17
18         if c == 3:
19             gradient_x = F.conv2d(images[:, 0, :, :], a)
20             gradient_y = F.conv2d(images[:, 0, :, :], b)
21         else:
22             gradient_x = F.conv2d(images, a)
23             gradient_y = F.conv2d(images, b)
24
25         edges = torch.sqrt(torch.pow(gradient_x, 2) + torch.pow(gradient_y, 2))
26         edges = F.pad(edges, (1, 1, 1, 1), "constant", 0)
27         thetas = torch.atan2(gradient_y, gradient_x)
28         thetas = F.pad(thetas, (1, 1, 1, 1), "constant", 0)
29
30         return edges, thetas
31
32     def forward(self, inputs, targets, images, masks=None):
33
34         if masks == None:
35             masks = targets > self.mask_value
36
37         # Comment this line if you don't want to use the multi-scale gradient matching term !!
38         # regularization_loss = self.regularization_loss(inputs.squeeze(1), targets.squeeze(1), masks.squeeze(1))
39
40         # find edges from RGB
41         edges_img, thetas_img = self.getEdge(images)
42
43
44         # =====
45         n, c, h, w = targets.size()
46         if n != 1:
47             inputs = inputs.view(n, -1).double()
48             targets = targets.view(n, -1).double()
49             masks = masks.view(n, -1).double()
50             edges_img = edges_img.view(n, -1).double()
51             thetas_img = thetas_img.view(n, -1).double()
52
53         # initialization
54         loss = torch.DoubleTensor([0.0]).cuda()
55
56
57         for i in range(n):
58             # Edge-Guided sampling
59             rgb_a, rgb_b, depth_a, depth_b, masks_A, masks_B, sample_num = edgeGuidedSampling(
60                 inputs[i, :], targets[i, :], edges_img[i], thetas_img[i], masks[i, :, h, w])
61
62             # Random Sampling
63             random_sample_num = sample_num
64             random_rgb_a, random_rgb_b, random_depth_a, random_depth_b, random_masks_A, random_masks_B = onlineSampling(
65                 inputs[i, :], targets[i, :], masks[i, :, self.mask_value, random_sample_num])
66
67
68             # Combine EGS + RS
69             rgb_a = torch.cat((rgb_a, random_rgb_a), 0)
70             rgb_b = torch.cat((rgb_b, random_rgb_b), 0)
71             depth_a = torch.cat((depth_a, random_depth_a), 0)
72             depth_b = torch.cat((depth_b, random_depth_b), 0)
73             masks_A = torch.cat((masks_A, random_masks_A), 0)
74             masks_B = torch.cat((masks_B, random_masks_B), 0)
75
76
77             # GT ordinal relationship
78             target_ratio = torch.div(depth_a+1e-6, depth_b+1e-6)
79             mask_eq = target_ratio.lt(
80                 1.0 + self.sigma) * target_ratio.gt(1.0/(1.0+self.sigma))
81             labels[target_ratio.ge(1.0 + self.sigma)] = 1
82             labels[target_ratio.le(1.0/(1.0+self.sigma))] = -1
83
84
85             # consider forward-backward consistency checking, i.e, only compute losses of point pairs with valid GT
86             consistency_mask = masks_A * masks_B
87
88             equal_loss = (rgb_a - rgb_b).pow(2) * \
89                         mask_eq.double() * consistency_mask
90             unequal_loss = torch.log(
91                 1 + torch.exp((-rgb_a + rgb_b) * labels)) * (~mask_eq).double() * consistency_mask
92
93
94             # Please comment the regularization term if you don't want to use the multi-scale gradient matching loss !!
95             # + 0.2 * regularization_loss.double()
96             loss = loss + self.alpha * equal_loss.mean() + 1.0 * unequal_loss.mean()
97
98
99         return loss[0].float()/n
```

```
In 1 1 import numpy as np
2 import matplotlib.pyplot as plt
3 import matplotlib.patches as patches
4 from torchvision import transforms
5 from data.loaders.DataLoader import RedWebDataset, Rescale, RandomCrop, ToTensor
Executed at 2023.12.02 11:12:22 in 485ms
> /Users/adi/opt/anaconda3/lib/python3.10/site-packages/torchvision/io/image.py:13: UserWarning: Failed to load image Pyth
```

|# lets choose a sample image to show how mini batch sampling works to calculate ordinal relations

```
In 2 1 transformed_dataset = RedWebDataset(root_dir="~/data/RedWeb_V1", transform=transforms.Compose([
2     Rescale(256),
3     RandomCrop(225)
4 ]))
Executed at 2023.12.02 11:12:22 in 41ms
```

```
In 3 1 sample_image = transformed_dataset[150]
```

```
2 fig,ax = plt.subplot_mosaic("AB")
```

```
3 ax["A"].imshow(sample_image["mono"])
4 ax["B"].imshow(sample_image["heat"])
```

```
5
```

```
6 for k in ax.keys():
7     ax[k].axis("off")
8     ax[k].grid("off")
```

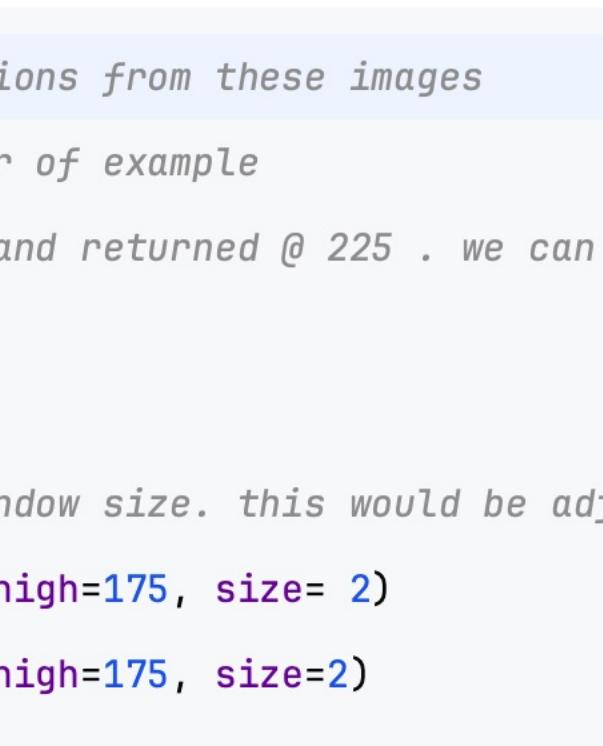
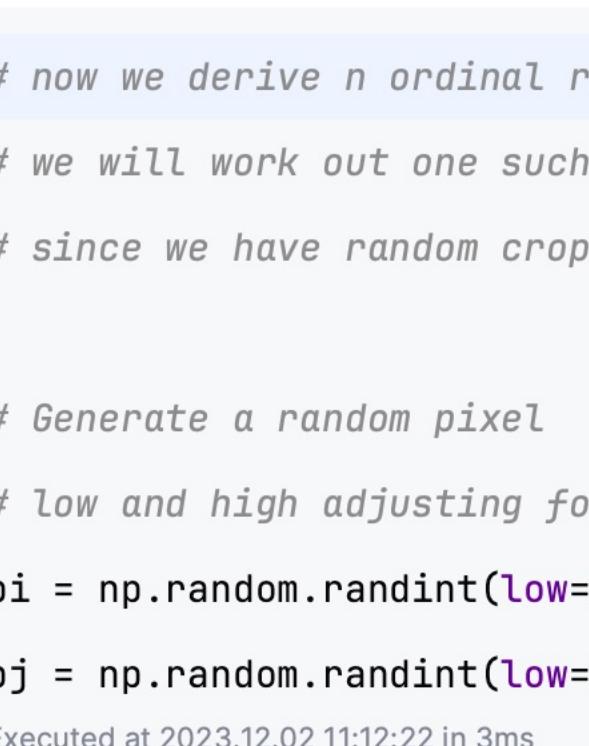
```
9
```

```
10 plt.show()
```

```
11
```

```
12
```

Executed at 2023.12.02 11:12:22 in 106ms



```
In 4 1 # now we derive n ordinal relations from these images
```

```
2 # we will work out one such pair of example
```

```
3 # since we have random cropped and returned @ 225 . we can hardcode these values for any image
```

```
4
```

```
5 # Generate a random pixel
```

```
6 # low and high adjusting for window size. this would be adjusted and corrected while making the loss function
```

```
7 pi = np.random.randint(low=50, high=175, size= 2)
```

```
8 pj = np.random.randint(low=50, high=175, size=2)
```

Executed at 2023.12.02 11:12:22 in 3ms

```
In 5 1 # window padding to get idea about the region we are going to talk about
```

```
2 fig , ax = plt.subplot_mosaic("AB;CD")
```

```
3 ax["A"].set_title("Point i", fontsize=6)
```

```
4 ax["A"].imshow(sample_image["mono"][pi[0]-40 : pi[0]+40 , pi[1]-40:pi[1]+40 , :])
```

```
5 ax["B"].set_title("Point j", fontsize=6)
```

```
6 ax["B"].imshow(sample_image["mono"][pj[0]-40 : pj[0]+40 , pj[1]-40:pj[1]+40 , :])
```

```
7 ax["C"].set_title("Ground truth i", fontsize=6)
```

```
8 ax["C"].imshow(sample_image["heat"][pi[0]-40 : pi[0]+40 , pi[1]-40:pi[1]+40])
```

```
9 ax["D"].set_title("Ground truth j", fontsize=6)
```

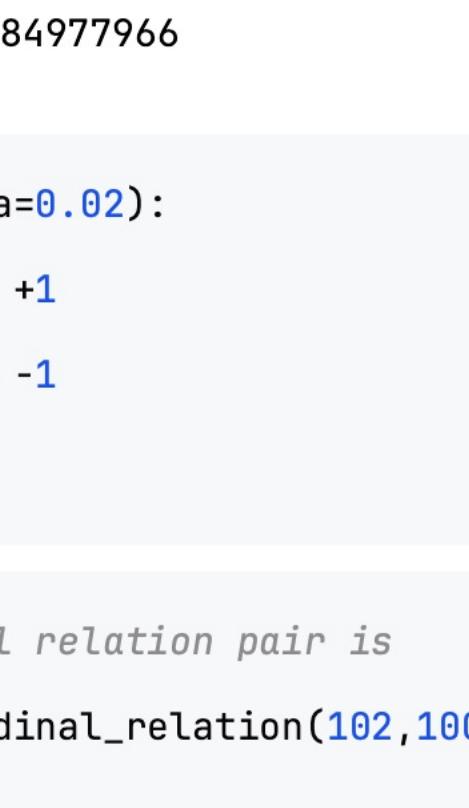
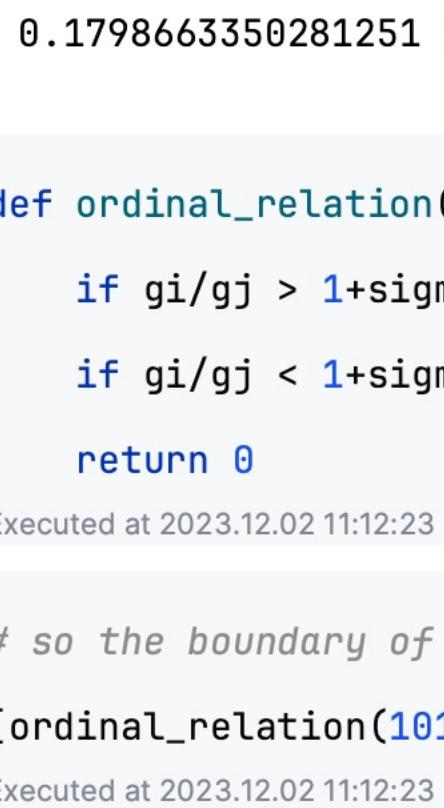
```
10 ax["D"].imshow(sample_image["heat"][pj[0]-40 : pj[0]+40 , pj[1]-40:pj[1]+40])
```

```
11
```

```
12 for k in ax.keys():
13     ax[k].axis("off")
14     ax[k].grid("off")
```

```
15
```

Executed at 2023.12.02 11:12:22 in 117ms



Ground truth i

Ground truth j

```
In 6 1 # lets calculate the ordinal relation of the pair of ordinal relation we are going to calculate
```

```
2 gi =sample_image["heat"][pi[0],pi[1]]
```

```
3 gj =sample_image["heat"][pj[0],pj[1]]
```

```
4 print(gi,gj)
```

Executed at 2023.12.02 11:12:22 in 3ms

0.1798663350281251 0.08230662784977966

```
In 7 1 def ordinal_relation(gi,gj,sigma=0.02):
```

```
2     if gi/gj > 1+sigma : return +1
```

```
3     if gi/gj < 1-sigma : return -1
```

```
4     return 0
```

Executed at 2023.12.02 11:12:23 in 2ms

```
In 8 1 # so the boundary of our ordinal relation pair is
```

```
2 [ordinal_relation(101,100) , ordinal_relation(102,100) , ordinal_relation(103,100)]
```

Executed at 2023.12.02 11:12:23 in 7ms

```
Out 8 [-1, 0, 1]
```

# now we just repeat it n times for a given image to generate such ordinal mappings

```
88 10 18 △ 34 ✓ 50 ▲ 100 99 98 97 96 95 94 93 92 91 90 89 88 87 86 85 84 83 82 81 80 79 78 77 76 75 74 73 72 71 70 69 68 67 66 65 64 63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
```

```
In 1 1 from google.colab import drive
2 drive.mount('/content/drive')
3 %cd /content/drive/MyDrive/DepthSense
```

```
In 2 1 !ls
```

```
In 3 1 !ls ReDWeb_V1/Img* | wc -l
```

```
In 4 1 !ls ReDWeb_V1/R0* | wc -l
```

```
In 5 1 !pip install -Uq segmentation-models-pytorch
```

```
In 6 1 !pip install torchmetrics
```

```
In 7 1 import os
2 import numpy as np
3 import pandas as pd
4 import torch
5 import torch.nn as nn
6 from torch.cuda.amp import GradScaler, autocast
7 from tqdm.auto import tqdm
8 import cv2 as cv
9 from PIL import Image
10 import segmentation_models_pytorch as smp
11 from sklearn.model_selection import train_test_split
12 import albumentations as A
13 from albumentations.pytorch import ToTensorV2
14 from torchmetrics.image import StructuralSimilarityIndexMeasure as SSIM
15 from torchmetrics.regression import MeanSquaredError as MSE
16 from torchmetrics.collections import MetricCollection
17 import gc
18 from torchvision.transforms import Normalize
19
20 import matplotlib.pyplot as plt
21 import torch
22 from torchvision import transforms
23 import torch.nn as nn
24
25
26 from torch.utils.data import DataLoader
27 from loss.Structure6Guide import EdgeguidedRankingLoss
28
29 from data.loaders.Dataset import RedWebDataset , Rescale , RandomCrop
30 normal_dataset = RedWebDataset(root_dir='../data/ReDWeb_V1',transform=transforms.Compose([
31     Rescale((256,256)),
32 ]))
33 batcher = DataLoader(normal_dataset,batch_size=1,shuffle=True)
```

```
In 8 1 rgb_dir = "ReDWeb_V1/Img*"
2 depth_dir = "ReDWeb_V1/R0*"
3 dataset_df = normal_dataset
```

```
In 9 1 train_df = dataset_df[:3200]
2 test_df = dataset_df[3200:]
3 print(train_df.shape , test_df.shape)

(3200, 2) (400, 2)
```

```
In 10 1 train_df, val_df = train_test_split(train_df,test_size=0.1,shuffle=True)
2 val_df, test_df = train_test_split(val_df, test_size=0.1,shuffle=True)
3 train_df.reset_index(drop=True,inplace=True)
4 val_df.reset_index(drop=True,inplace=True)
5 test_df.reset_index(drop=True,inplace=True)
6 len(train_df),len(val_df), len(test_df)
```

```
Out 10 (2880, 360, 40)
```

```
In 11 1 sample_tfms = [
2     A.HorizontalFlip(),
3     A.GaussNoise(p=0.2),
4     A.OneOf([
5         A.MotionBlur(p=0.3),
6         A.MedianBlur(blur_limit=3, p=0.3),
7         A.Blur(blur_limit=3, p=0.5),
8     ], p=0.3),
9     A.RGBShift(),
10    A.RandomBrightnessContrast(),
11    A.RandomResizedCrop(384,384),
12    A.ColorJitter(),
13    A.ShiftScaleRotate(shift_limit=0.1, scale_limit=0.3, rotate_limit=45, p=0.5),
14    A.HueSaturationValue(p=0.3),
15 ]
16 > train_tfms = A.Compose([...],is_check_shapes=False)
22 > valid_tfms = A.Compose([...],is_check_shapes=False)
```

```
In 12 1 > class Dataset:...
```

```
In 13 1 train_ds = Dataset(train_df,train_tfms)
2 val_ds = Dataset(val_df,valid_tfms)
3 test_ds = Dataset(test_df,valid_tfms)
4 len(train_ds), len(val_ds), len(test_ds)
```

```
Out 14 (2880, 360, 40)
```

```
In 15 1 def colored_depthmap(depth, d_min=None, d_max=None,cmap=plt.cm.inferno):
2     if d_min is None:
3         d_min = np.min(depth)
4     if d_max is None:
5         d_max = np.max(depth)
6     depth_relative = (depth - d_min) / (d_max - d_min)
7     return 255 * cmap(depth_relative)[...,3] # H, W, C
```

```
In 16 1 > plt.figure(figsize=(15, 6))
2 for i, idx in enumerate(np.random.randint(0,len(dataset_df),(16,))):
3     ax = plt.subplot(4, 4, i + 1)
4     image = Image.open(dataset_df.iloc[idx,0]).convert('RGB')
5     mask = Image.open(dataset_df.iloc[idx,1]).convert('L')
6     image_viz = merge_into_row([image,mask])
7     plt.imshow(image_viz.astype("uint8"))
8     plt.axis("off")
```

```
Out 16
```

```
In 17 1
2 class ResnetFF(nn.Module):
3     def __init__(self, encoder_name='resnext50_32x4d', in_channels=3, classes=1):
4         super().__init__()
5         self.encoder = smp.encoders.get_encoder(encoder_name, in_channels=in_channels)
6         self.decoder = smp.unet.decoder.UnetDecoder(
7             encoder_channels=self.encoder.out_channels,
8             decoder_channels=[256, 128, 64, 32],
9             n_blocks=(3, 2, 1, 0),
10            use_batchnorm=True,
11            center=False,
12            attention_type=None, # You can change this to 'scse' or 'cbam' for attention mechanisms
13        )
14        self.segmentation_head = smp.unet.model.UnetHead(
15            in_channels=self.decoder.out_channels[-1], out_channels=classes, activation=None
16        )
17
18    def trainable_encoder(self, trainable=True):
19        for p in self.encoder.parameters():
20            p.requires_grad = trainable
21
22    def forward(self, x):
23        features = self.encoder(x)
24        decoder_output = self.decoder(*features)
25        logits = self.segmentation_head(decoder_output[-1])
26        return logits
27
28    def _num_params(self):
29        return sum([p.numel() for p in self.parameters() if p.requires_grad])
30
31 model = ResnetFF(encoder_name='resnext50_32x4d', in_channels=3, classes=1)
```

```
In 18 1 > class UnNormalize(Normalize):
2     def __init__(self,*args,**kwargs):
3         mean=(0.485, 0.456, 0.406)
4         std=(0.229, 0.224, 0.225)
5         new_mean = [-m/s for m,s in zip(mean,std)]
6         new_std = [1/s for s in std]
7         super().__init__(new_mean, new_std, *args, **kwargs)
8
9 @torch.no_grad()
10 def plot_wals(imgs, preds, targets,n=4,figsize=(6,2),title=''):
11     plt.figure(figsize=figsize,dpi=150)
12     r = 2 if n == 4 else 8
13     c = 2
14     for i, idx in enumerate(np.random.randint(0,imgs.size(0),(n,))):
15         ax = plt.subplot(r, c, i + 1)
16         img,pred,gt = imgs[idx],preds[idx],targets[idx]
17         img = UnNormalize()(img)*255.
18         img,pred,gt = img.permute(1,2,0).numpy(), pred.permute(1,2,0).numpy(), gt.permute(1,2,0).numpy()
19         pred = colored_depthmap(np.squeeze(pred))
20         gt = colored_depthmap(np.squeeze(gt))
21         image_viz = np.hstack([img,pred,gt])
22         plt.imshow(image_viz.astype("uint8"))
23         plt.axis("off")
24         title = f'{title}\input - output - redweb_out' if len(title)!=0 else 'input - output - redweb_out'
25         plt.suptitle(title)
26         plt.show()
```

```
In 19 1 epochs = 5
2 freeze_epochs = 2
3 lr = 1e-3
4 device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
5 print(device)
6
7 metrics = MetricCollection([
8     SSIM(data_range=(0,1)),
9     MSE()
10 ]).to(device)
11 train_metrics = metrics.clone()
12 val_metrics = metrics.clone()
13
14 logs = pd.DataFrame()
15 logs[['loss_train','loss_val','ssim_train','ssim_val','mse_train','mse_val']] = None
16
17 model = model.to(device)
18 model.trainable_encoder(trainable=False)
19
20 loss_fn = EdgeguidedRankingLoss()
22 optim = torch.optim.AdamW(model.parameters(),lr=lr / 25.,weight_decay=0.02)
23 sched = torch.optim.lr_scheduler.OneCycleLR(optim,max_lr=lr,epochs=epochs,steps_per_epoch=(len(train_dl)))
24
25 scaler = GradScaler()

cuda
Download: "https://download.pytorch.org/models/resnext50_32x4d-7cfdf4587.pth" to /root/.cache/torch/hub/checkpoints/resnext50_32x4d-7cfdf4587.pth
100% [██████████] 95.8M/95.8M [00:00:00, 238MB/s]
```

```
In 20 1 best_ssim = -1e9
2 best_epoch = -1
3
4 print('training decoder only')
5
6 for epoch in tqdm(range(epochs)):
7
8     model.train()
9
10    if epoch == freeze_epochs:
11        model.trainable_encoder(trainable=True)
12        print('training encoder and decoder both')
13
14    running_loss = 0.
15    train_prog = tqdm(train_dl,total=len(train_dl))
16
17    for img, mask in train_prog:
18
19        with autocast():
20            img, mask = img.to(device), mask.to(device)
21            preds = model(img)
22
23            loss = loss_fn(preds,mask)
24            scaler.scale(loss).backward()
25            scaler.unscale_(optim)
26            nn.utils.clip_grad_norm_(model.parameters(), max_norm=2.0, norm_type=2)
27            scaler.step(optim)
```

```
Out 20
```

```
In 21 1 best_ssim = -1e9
2 best_epoch = -1
3
4 print('training decoder only')
5
6 for epoch in tqdm(range(epochs)):
7
8     model.train()
9
10    if epoch == freeze_epochs:
11        model.trainable_encoder(trainable=True)
12        print('training encoder and decoder both')
13
14    running_loss = 0.
15    train_prog = tqdm(train_dl,total=len(train_dl))
16
17    for img, mask in train_prog:
18
19        with autocast():
20            img, mask = img.to(device), mask.to(device)
21            preds = model(img)
22
23            loss = loss_fn(preds,mask)
24            scaler.scale(loss).backward()
25            scaler.unscale_(optim)
26            nn.utils.clip_grad_norm_(model.parameters(), max_norm=2.0, norm_type=2)
27            scaler.step(optim)
```

```
6:3 LF UTF-8 4 spaces /Users/adil/opt/anaconda3
```

```

27     scaler.step(optim)
28     scaler.update()
29     sched.step()
30     optim.zero_grad()
31
32     running_loss += loss.item()
33
34     train_prog.set_description(f'loss: {loss.item():.3f}')
35     train_metrics(preds, mask)
36
37     del img, mask, preds, loss
38
39     m = train_metrics.compute()
40     _ssim,_mse = m['StructuralSimilarityIndexMeasure'].cpu().item(), m['MeanSquaredError'].cpu().item()
41     logs.loc[epoch,['loss_train','ssim_train','mse_train']] = (running_loss/len(train_dl),_ssim,_mse)
42     train_metrics.reset()
43     model.eval()
44
45     with torch.no_grad():
46
47         running_loss = 0.
48
49         val_prog = tqdm(val_dl, total=len(val_dl))
50         for img, mask in val_prog:
51
52             with autocast():
53                 img, mask = img.to(device), mask.to(device)
54                 preds = model(img)
55                 loss = loss_fn(preds,mask)
56                 running_loss += loss.item()
57                 val_prog.set_description(f'loss: {loss.item():.3f}')
58
59             val_metrics(preds, mask)
60
61         del img, mask, preds, loss
62
63         m = val_metrics.compute()
64         _ssim,_mse = m['StructuralSimilarityIndexMeasure'].cpu().item(), m['MeanSquaredError'].cpu().item()
65         logs.loc[epoch,['loss_val','ssim_val','mse_val']] = (running_loss/len(val_dl),_ssim,_mse)
66         val_metrics.reset()
67
68     if _ssim > best_ssimm:
69         best_ssimm = _ssim
70         best_epoch = epoch
71         sd = model.state_dict()
72         torch.save(sd,'ResnetFF.pt')
73
74     print(f"\n\n{logs.tail(1)}\n\n")
75
76     with torch.no_grad():
77         with autocast():
78             img, mask = next(iter(test_dl))
79             img, mask = img.to(device), mask.to(device)
80             preds = model(img)
81             plot_vals(
82                 img.cpu(),
83                 preds.cpu(),
84                 mask.cpu()
85             )
86
87     gc.collect()
88     torch.cuda.empty_cache()

training decoder only
0%| 0/5 [00:00<?, ?it/s]

0%| 0/45 [00:00<?, ?it/s]

    /usr/local/lib/python3.10/dist-packages/torch/optim/lr_scheduler.py:136: UserWarning: Detected call of `lr_scheduler.step()` before `optimizer.step()`. In PyTorch 1.1.0 and later, you should call them in the opposite order: `optimizer.step()` before `lr_scheduler.step()`. Failure to do this will result in PyTorch skipping the first value of the learning rate schedule. See more details at https://pytorch.org/docs/stable/optim.html#how-to-adjust-learning-rate
      warnings.warn("Detected call of `lr_scheduler.step()` before `optimizer.step()`.")

0%| 0/6 [00:00<?, ?it/s]

```

loss\_train loss\_val ssim\_train ssim\_val mse\_train mse\_val  
0 0.648853 0.226287 0.136301 0.185004 0.648853 0.225426

#### input - output - redweb\_out

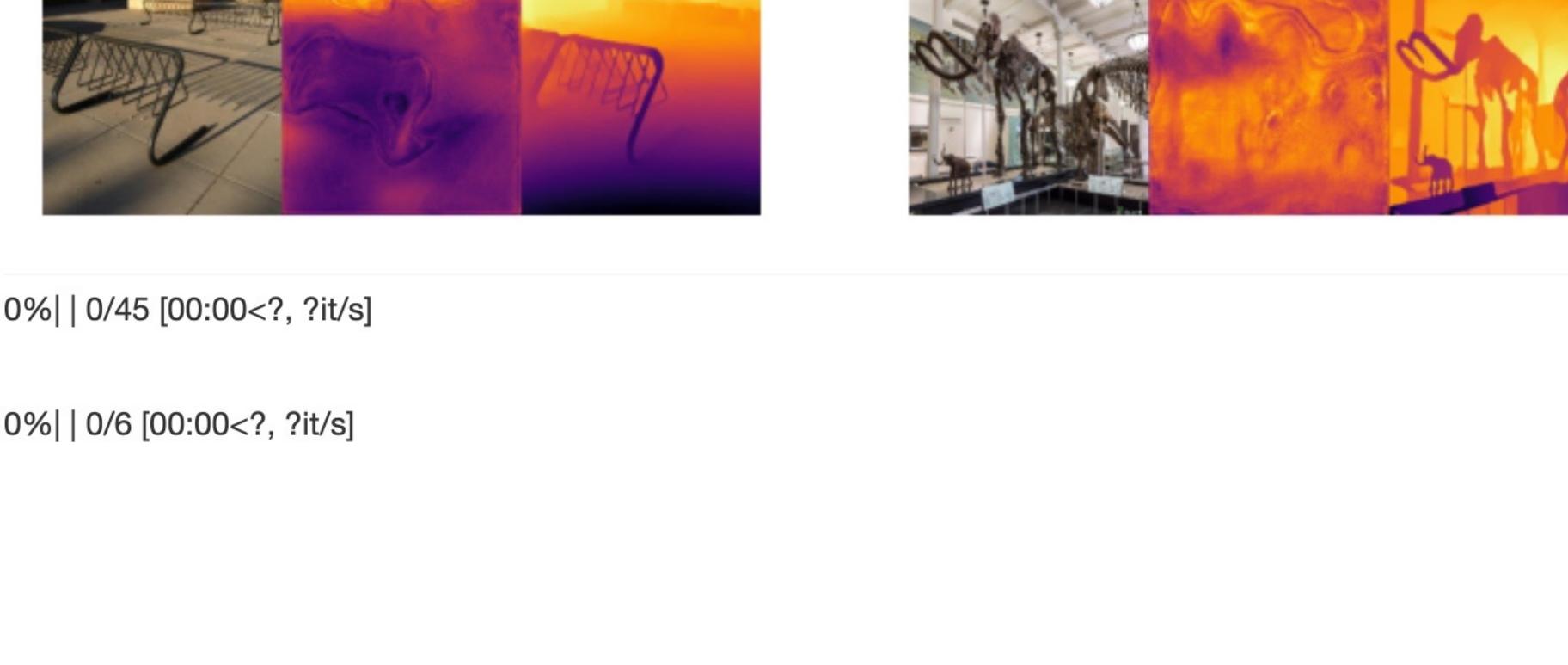


0%| 0/45 [00:00<?, ?it/s]

0%| 0/6 [00:00<?, ?it/s]

loss\_train loss\_val ssim\_train ssim\_val mse\_train mse\_val  
1 0.051893 0.04105 0.566209 0.571114 0.051893 0.041102

#### input - output - redweb\_out



0%| 0/45 [00:00<?, ?it/s]

0%| 0/6 [00:00<?, ?it/s]

loss\_train loss\_val ssim\_train ssim\_val mse\_train mse\_val  
2 0.05496 0.052545 0.641519 0.611616 0.05496 0.053002

#### input - output - redweb\_out



```
In 1 1 from google.colab import drive
2 import torch
3
4 drive.mount('/content/drive')
5 %cd /content/drive/MyDrive/DepthSense
```

Mounted at /content/drive

/content/drive/MyDrive/DepthSense

```
In 21 1 best_epoch
```

```
Out 21 4
```

```
In 23 1 from ResnetFF import ResnetFF
```

2 model = ResnetFF()

3 best\_sd = torch.load('ResnetFF.pt')

4 model.load\_state\_dict(best\_sd)

5

Out 23 <All keys matched successfully>

```
In 24 1 all_imgs, all_preds, all_targets = [], [], []
```

2 with torch.no\_grad():

3 with autocast():

4 for img, mask in tqdm(test\_dl, total=len(test\_dl)):

5 img, mask = img.to(device), mask.to(device)

6 pred = model(img)

7 all\_imgs.append(img)

8 all\_preds.append(pred)

9 all\_targets.append(mask)

10

11 test\_metrics = metrics.clone()

12 test\_metrics[

13 torch.vstack(all\_preds),

14 torch.vstack(all\_targets)

15 ]

16 m = test\_metrics.compute()

17 title = f"SSIM: {m['StructuralSimilarityIndexMeasure'].cpu().item():.3f} MSE: {m['MeanSquaredError'].cpu().item():.3f}"

18 plot\_vals(

19 torch.vstack(all\_imgs).cpu(),

20 torch.vstack(all\_preds).cpu(),

21 torch.vstack(all\_targets).cpu(),

22 n=16,

23 figsize=(10,15),

24 title=title

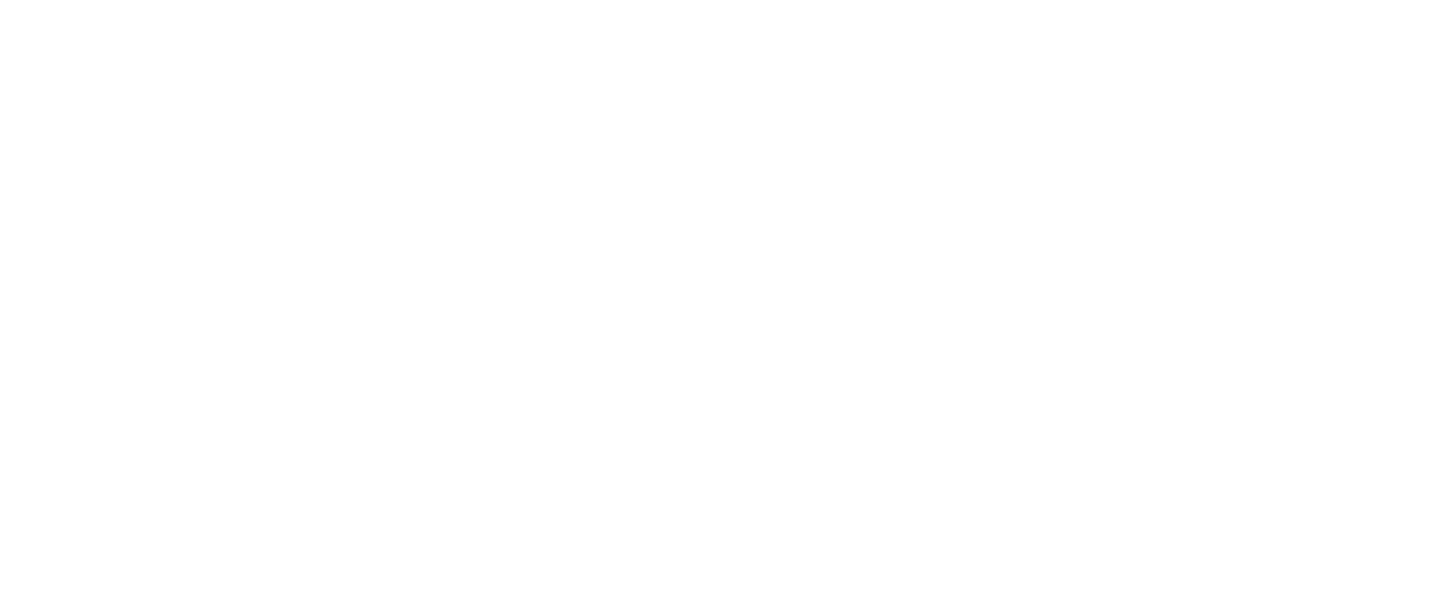
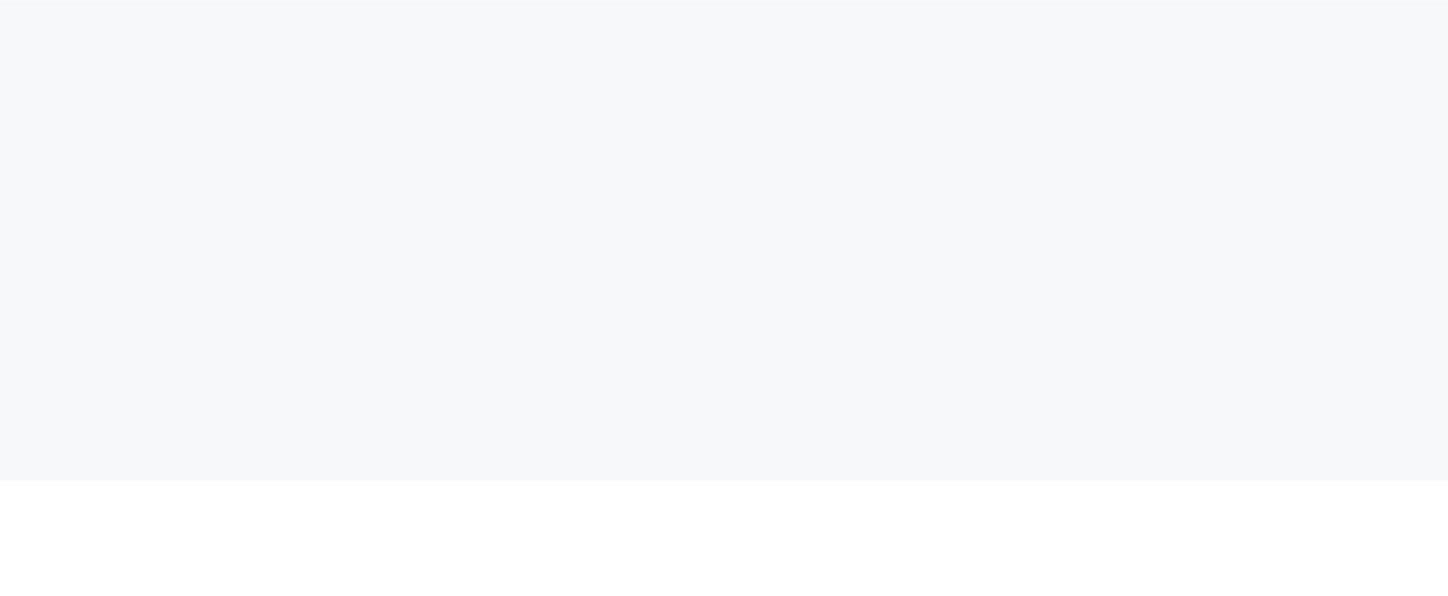
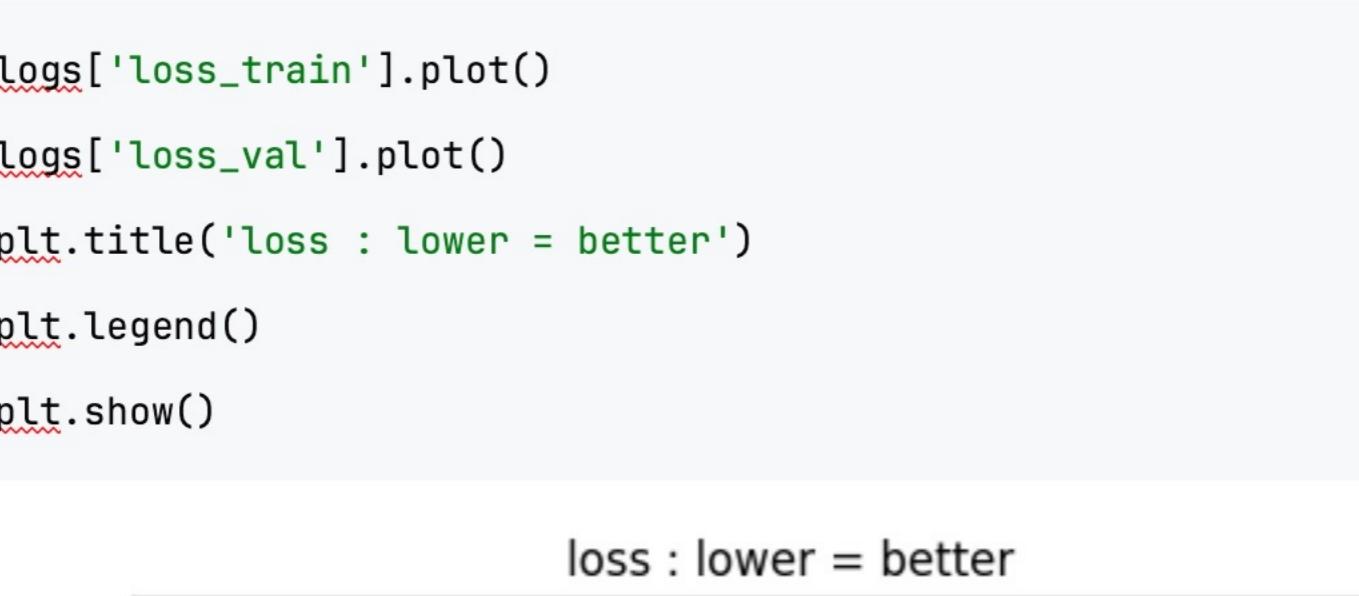
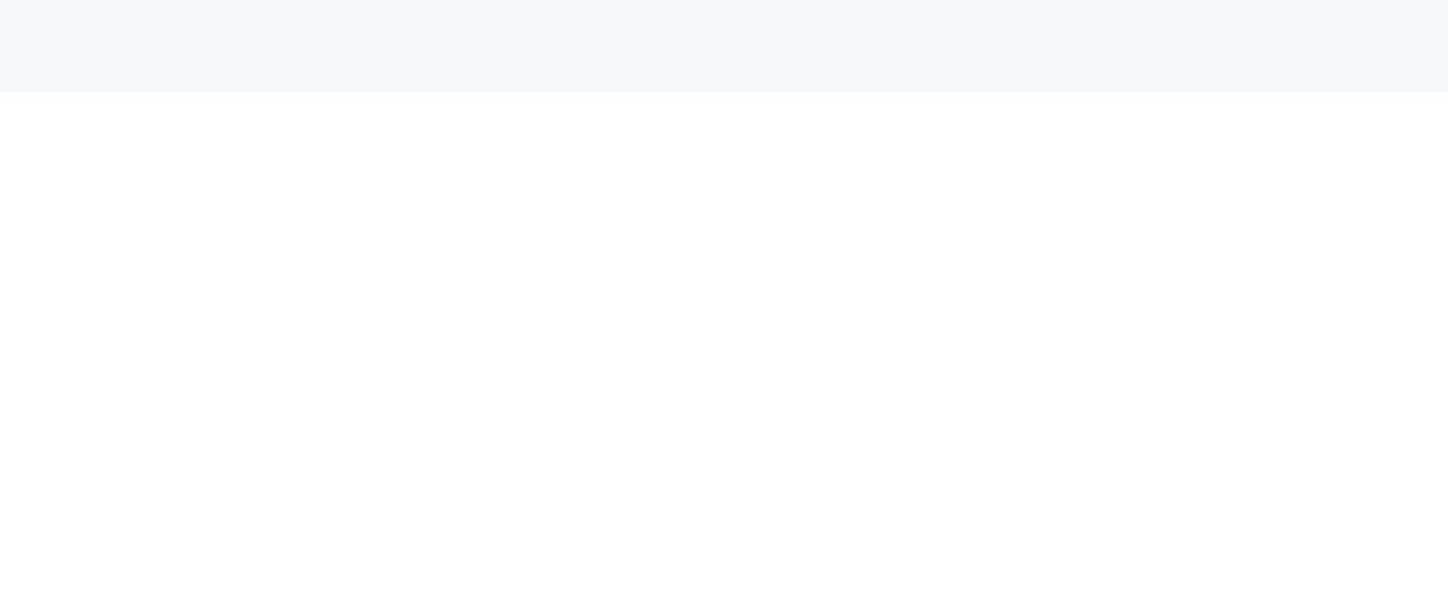
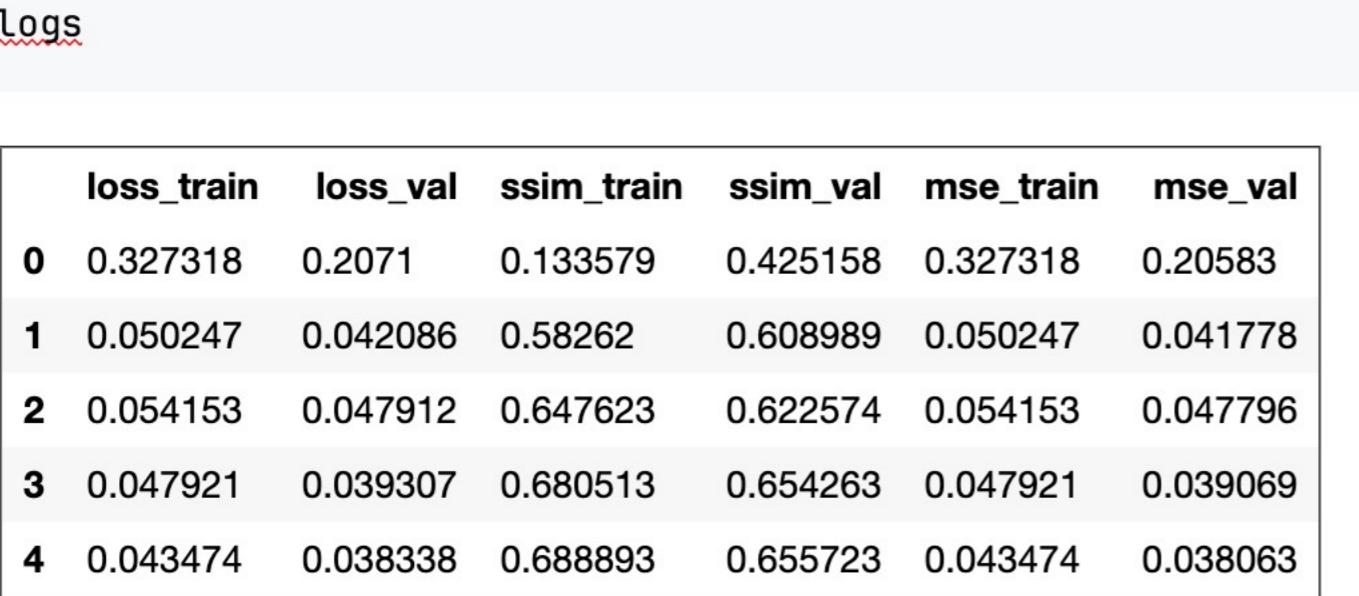
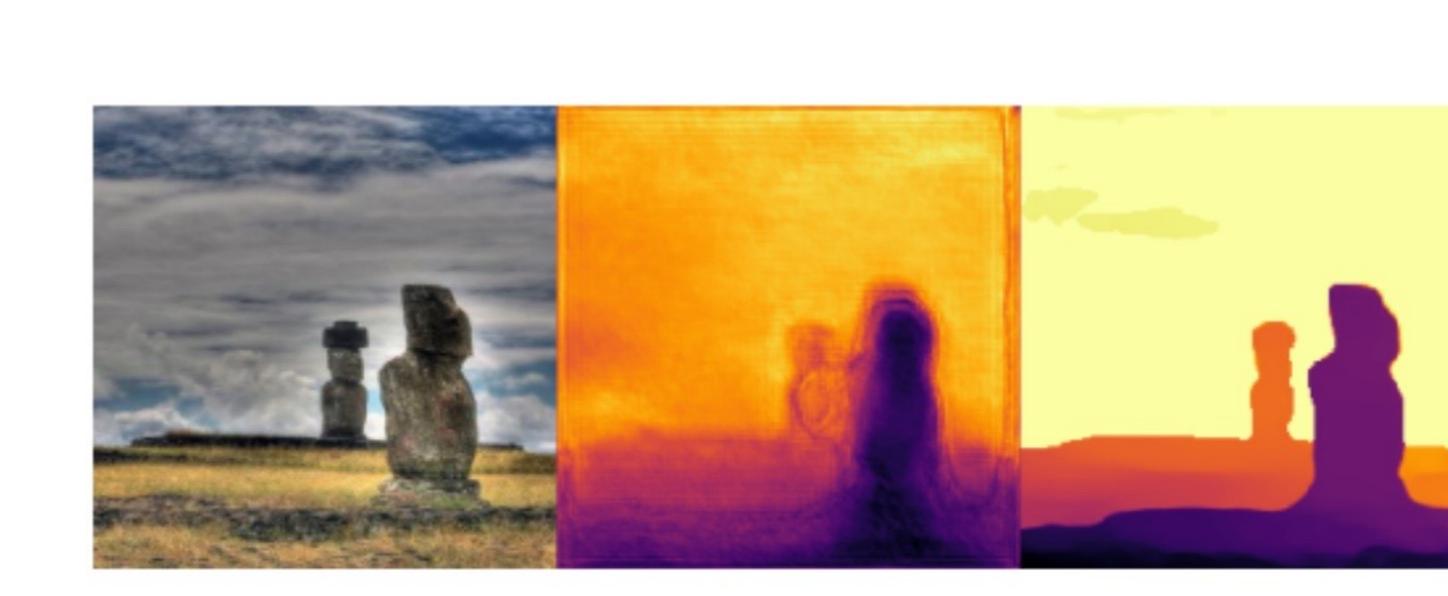
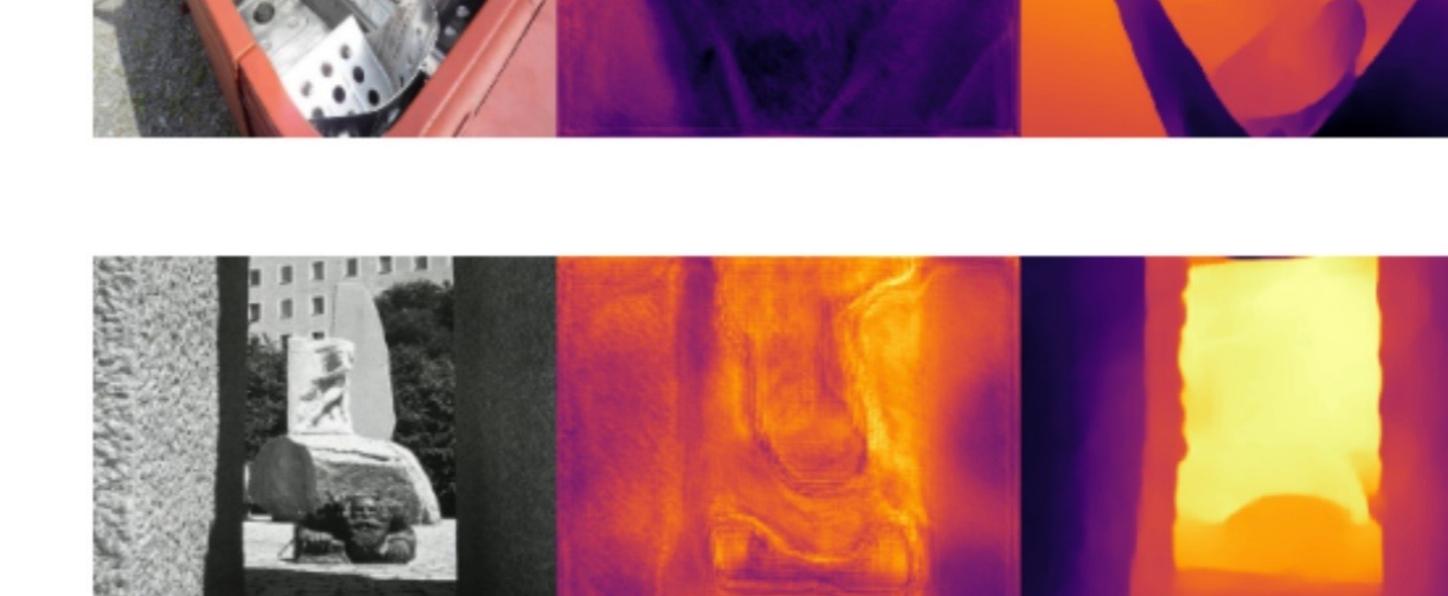
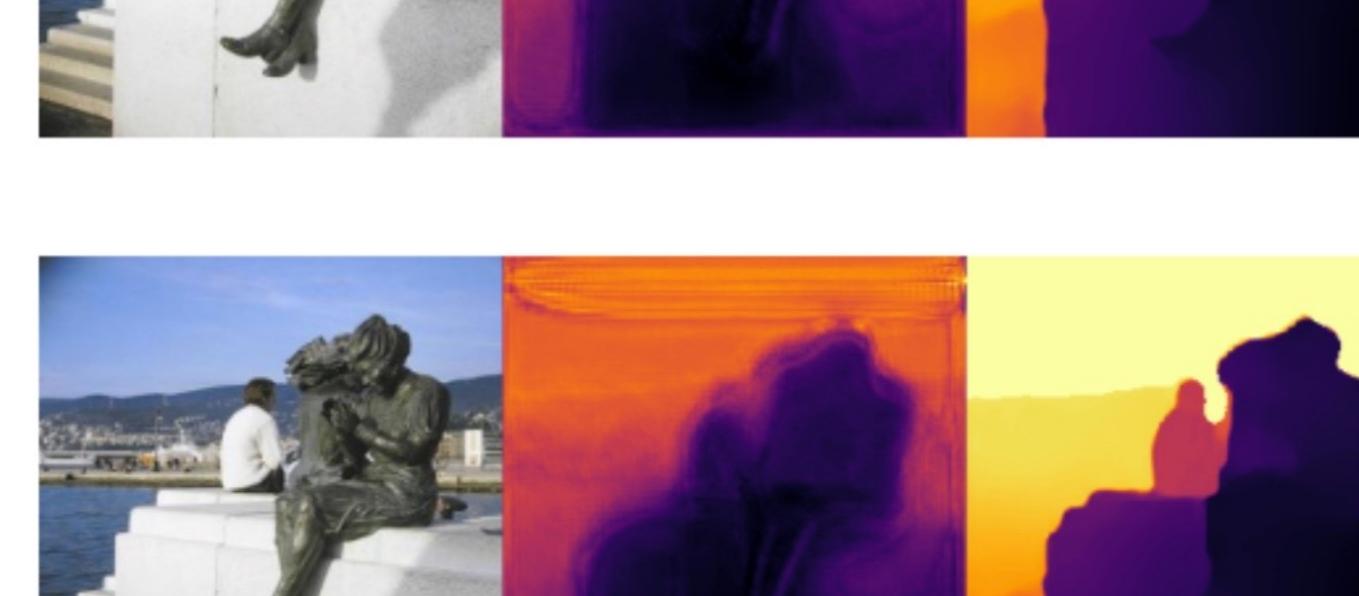
25 )

26

27

0%|| 0/10 [00:00<?, ?it/s]

SSIM: 0.684 MSE: 0.035  
image/target/prediction



```
In 25 1 logs
```

```
Out 25 <Table> loss_train loss_val ssim_train ssim_val mse_train mse_val
```

	loss_train	loss_val	ssim_train	ssim_val	mse_train	mse_val
0	0.327318	0.2071	0.133579	0.425158	0.327318	0.20583
1	0.050247	0.042086	0.58262	0.608989	0.050247	0.041778
2	0.054153	0.047912	0.647623	0.622574	0.054153	0.047796
3	0.047921	0.039307	0.680513	0.654263	0.047921	0.039069
4	0.043474	0.038338	0.688893	0.655723	0.043474	0.038063

```
In 26 1 logs['loss_train'].plot()
```

2 logs['loss\_val'].plot()

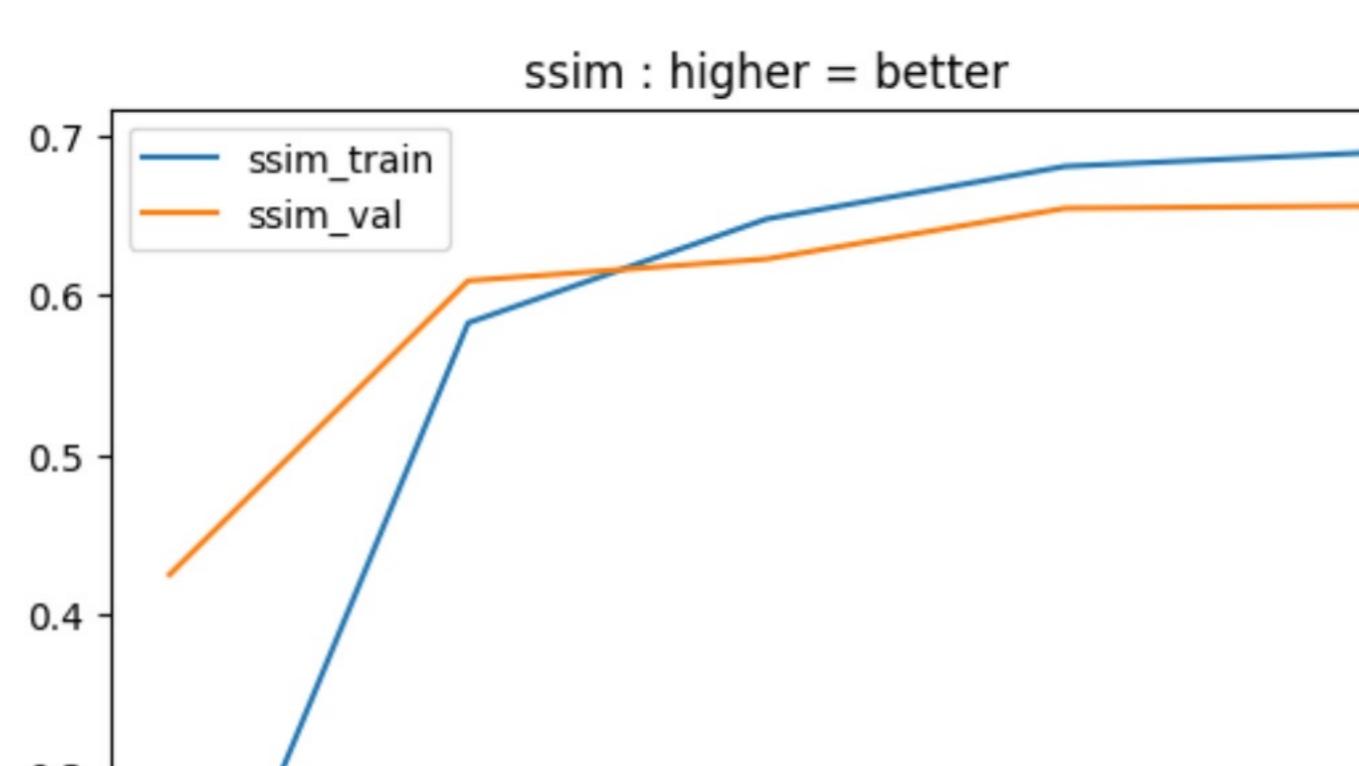
3 plt.title('loss : lower = better')

4 plt.legend()

5 plt.show()

loss : lower = better

loss\_train loss\_val



```
In 27 1 logs['mse_train'].plot()
```

2 logs['mse\_val'].plot()

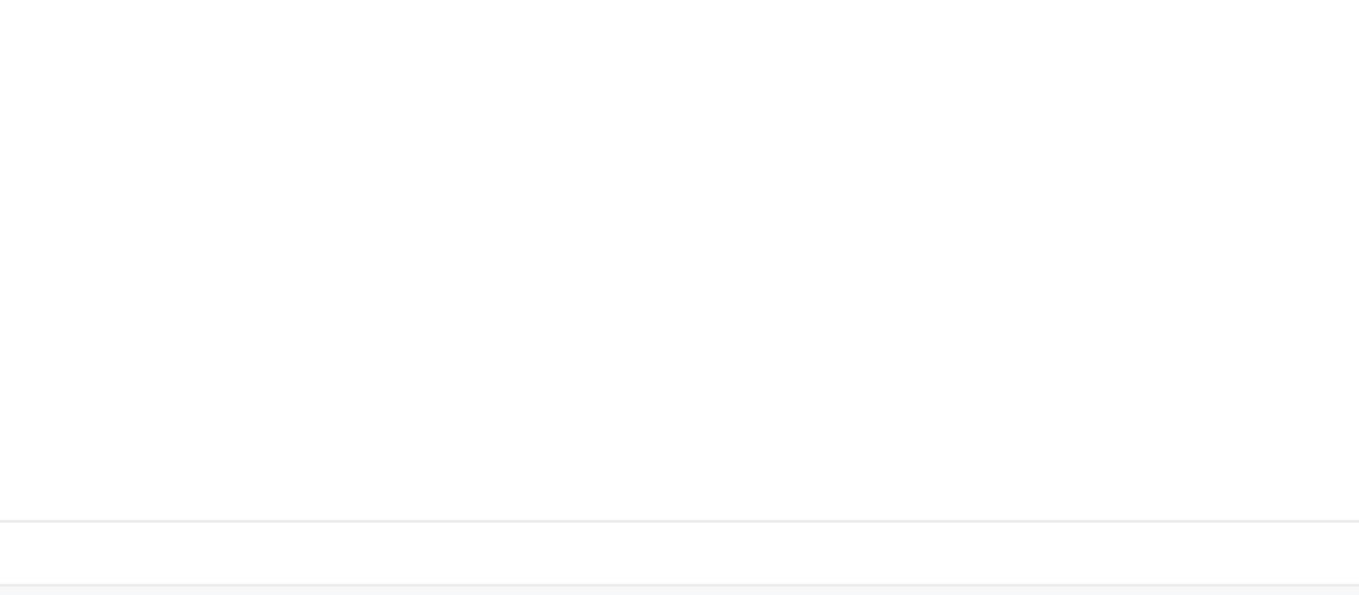
3 plt.title('mse : lower = better')

4 plt.legend()

5 plt.show()

mse : lower = better

mse\_train mse\_val



```
In 28 1 logs['ssim_train'].plot()
```

2 logs['ssim\_val'].plot()

3 plt.title('ssim : higher = better')

4 plt.legend()

5 plt.show()

ssim : higher = better

ssim\_train ssim\_val

