1. Supplementary

1.1. Meta Augmentations

The most straightforward way of defining a custom augmentation regimen is through meta-augmentations. In Listing 1 we can see how someone with domain knowledge could express the process generating plausible distortions as an augmentation regimen. In Fig. 1 we can see the flow network that listing 1 produced. Finally, in Fig. 2 we can see an application of 24 augmentation instances from the defined regimen.

```
from tormentor import RandomColorJitter, RandomFlip, RandomPerspective,
RandomPlasmaBrightness, RandomGaussianAdditiveNoise, RandomRotate

linear_aug = (RandomFlip ^ RandomPerspective ^ RandomRotate) | RandomColorJitter
```

```
nonlinear_aug = RandomWrap | RandomPlasmaBrightness
final_augmentation = (linear_aug ^ nonlinear_aug) | RandomGaussianAdditiveNoise
epochs, batch_size, n_points, width, height = 10, 5, 20, 320, 240
for _ in range(epochs):
    image_batch = torch.rand(batch_size, 3, height, width)
    segmentation_batch = torch.rand(batch_size, 1, height, width).round()
    augmentation = final_augmentation()
    augmented_image = augmentation(image_batch)
    augmented_gt = augmentation(segmentation_batch)
    # Train and do other things.
```

Listing 1. Meta-augmentations used to define augmentation routing

1.2. Defining a New Augmentation

TorMentor can be customized by writing custom augmentation operations. An augmentation operation must be defined as a Python class. Although augmentations can deal both with samples (3D tensors) The class must implement the method `generate_batch_state` which takes as a parameter the input data in order to know its size. If it is a dorsal operation, it should be inheriting class `ColorAugmentation` and implement the functional method `functional_image` which takes as parameters a batch of data followed by zero to many tensors whose first dimension must be batch size. If on the other hand it is a ventral operation, the functional method must be applied on a sampling field pointing on the image pixels. Regardless of the image size, the sampling field has values between -1 and 1 both horizontally and vertically. In Listing 2 the definition of a ventral augmentation mimicking a lens effect can be seen, while in Fig. 3 we can see the defined augmentation applied on a sample of the COCO dataset.

```
import tormentor

class Lens(tormentor.SpatialImageAugmentation):
    center_x = tormentor.Uniform((-.3, .3))
```

1
Figure 2. Application of 24 different augmentation instances from the augmentation regiment defined in listing 1 and described in Fig. 1.

Figure 3. Custom lens augmentation applied on MS-COCO segmentation.

Listing 2. Custom augmentation definition of a lens effect

```python
center_y = tormentor.Uniform((-.3, .3))
gamma = tormentor.Uniform((1., 1.))

def generate_batch_state(self, samples):
    batch_sz = sampling_tensors[0].size(0)
    gamma = type(self).gamma(batch_sz, device=samples[0].device).view(-1)
    center_x = type(self).center_x(batch_sz, device=samples[0].device).view(-1)
    center_y = type(self).center_y(batch_sz, device=samples[0].device).view(-1)
    return center_x, center_y, gamma

@classmethod
def functional_sampling_field(cls, sampling_field, center_x, center_y, gamma):
    field_x, field_y = sampling_field
    center_x = center_x.unsqueeze(dim=1).unsqueeze(dim=1)
    center_y = center_y.unsqueeze(dim=1).unsqueeze(dim=1)
    gamma = gamma.unsqueeze(dim=1).unsqueeze(dim=1)
    distance = ((center_x - field_x)**2 + (center_y - field_y)**2)**.5
    field_x = field_x + field_x * distance ** gamma
    field_y = field_y + field_y * distance ** gamma
    return field_x, field_y

Notice that the pointcloud domain works perfectly well with polygon defined segmentations.
```