VOPCE manuscript

The model aims to learn a VAE that encode a point cloud into a fixed length latent vector.

Basic usage

The main file of the system is in “VOPCE.py”, which contains the main class of our model “VOPCE”, you can set up an object of VOPCE as:

```
if __name__ == '__main__':
    my_vopce = VOPCE(z_dim=50, x_dim=3, num_gauss=50)
```

Here we have 3 parameters,
z_dim The dimension of desired latent vector.
x_dim The dimension of input data, should be 3 for x, y, z
num_gauss Num of GMM clusters for representing the point cloud, for object usually 50 is ok.

Then you can train the model with ‘train’ function:

```
if __name__ == '__main__':
    my_vopce = VOPCE(z_dim=50, x_dim=3, num_gauss=50)
    for i in range(1000):
           my_vopce.train(torch.from_numpy(data).float())
```

Input data as point cloud is a numpy array with (N*3) size, N is the number of point, which can be arbitrary large.

After training with data, you can reconstruct a new point cloud from original point cloud

```
y = my_vopce.forward(torch.from_numpy(data).float()).detach().numpy()
```

Then you shall get numpy array ‘y’ as the same shape as ‘data’ with the same size (N*3) and should look similarly.
You can also extract the latent vector of ‘data’ by

```
z, _, _ = my_vopce.encode(torch.from_numpy(data).float())
z = z.detach().numpy()
```

Then you shall get numpy array ‘z’ with size (1, z_dim).

If you have some latent variables 'z' as a torch tensor of size (1, z_dim), you can directly see what it encodes by

```
x = my_vopce.decode(z)
```

Load Standard Point Cloud

So if you don't have point cloud data, you can create a folder "`./data/modelnet40_normal_resampled`" under the same directory with "VOPCE.py" with all models saved in such directory as txt file. Then you can use "myLoader" class to load ModelNet model. (I will put link to download data soon, but you can find you own point cloud by given an numpy array of (N, 3)).

First, you should assign how many sample for each object by

```
n_sample = 1024
my_loader = MyLoader('ModelNet', n_sample=n_sample)
```

Then you can load model from my_Loader, each object is recorded by class "RenderObj"

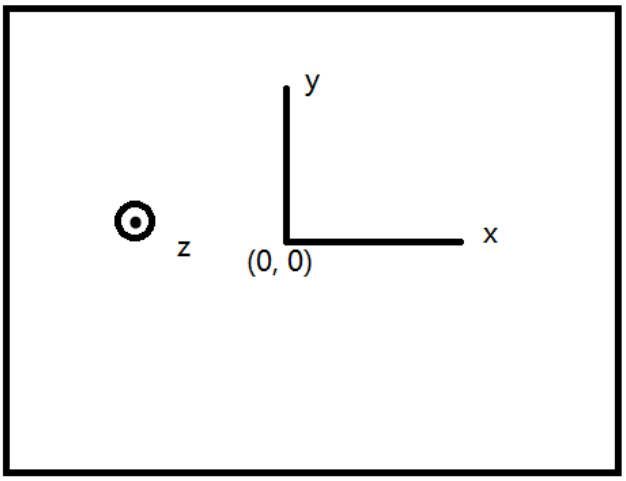
```
render_obj_1 = my_loader.get_obj(index=0, color=[1.0, 0.0, 1.0], pos=[-1.5, 1.0, -6.0],
rot=[0.0, 0.0, 0.0])
```

index	0-9000 (approximately, each one is a different object, like plane or something)
color	RGB to be rendered
pos	position rendered in scene x, y, x
rot	rotation rendered in scene row, pitch, yawl

loaded objects like `render_obj_1` has a member variable `render_obj_1.data` that is the numpy array (N*3) can be used for last section.

Visualize model

class "RenderObj" can be rendered by OpenGL engine. As you see for myLoader.get_obj, you provide a 'pos', that determines where it is rendered. You can produce as many as "RenderObj" as you want, and provide them with different position (they can be rendered at different positions). Then camera is set as (0, 0, 0) and looking at -z axis., the coordinate is

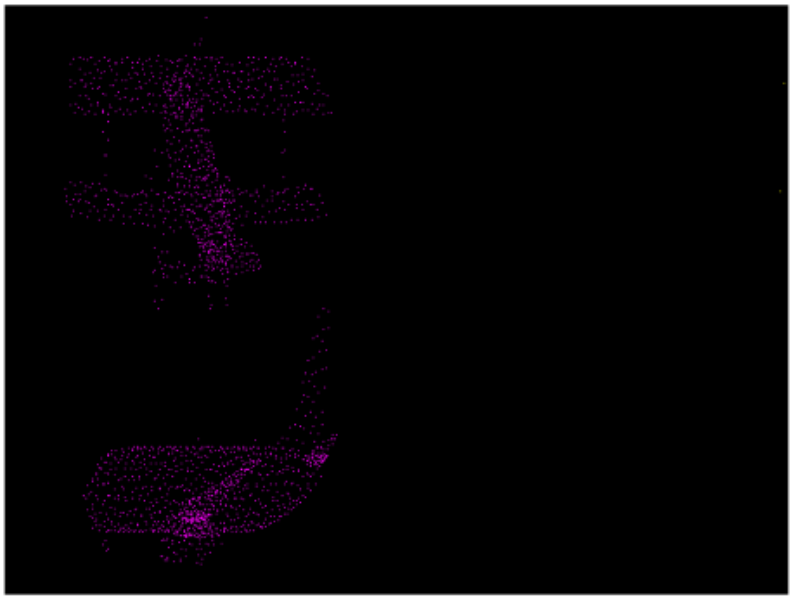


Z axis is pointing out of screen.

To render object, you should create a list and add all objects in the list, finally call “plot” function to show them in a window, like

```
if __name__ == '__main__':
    n_sample = 512
    my_loader = MyLoader('ModelNet', n_sample=n_sample)
    render_list = []
    render_obj = my_loader.get_obj(index=0, color=[1.0, 0.0, 1.0], pos=[-1.5, 1.0, -6.0], rot=[0.0, 0.0, 0.0])
    render_list.append(render_obj)
    render_obj = my_loader.get_obj(index=1, color=[1.0, 1.0, 0.0], pos=[1.5, 1.0, -6.0], rot=[0.0, 0.0, 0.0])
    render_list.append(render_obj)
    my_loader.plot(render_list)
```

It gives you something like



You can see two planes are rendered at the left side of screen.

Have fine!