CV大作业中期报告

2024.1.17 成员：xxx, xxx, xxx

Part1 Introduction[xly]

(a brief introduction，such as"我们选择的题目是xxx，经过xxx时间我们目前完成了xxx内容、达到了xxx效果，后续计划完成xxx")

Part2 Related Works

(1) GAN based models[wmq]

Basic GAN model[https://arxiv.org/abs/1406.2661] consists of two parts: generator and discriminator. In the first version, both the generator and the discriminator are MLPs. The advanced GANs make many modifications to the model structure, such as using CNNs --- which is called DCGAN[https://arxiv.org/abs/1701.07875], using Varation-AutoEncoder --- which is called VAE-GAN[https://arxiv.org/abs/1406.2661], changing loss function from JS divergence to Wasserstein loss --- which is called WGAN[https://arxiv.org/abs/1701.07875]. These modifications focused on higher ability and better stablity. Another kind of modification concerns how to generate higher resolutions and bigger results. One basic way is to adding new structure and stronger computation, such as SA-GAN[https://arxiv.org/abs/1805.08318] using self-attention, SN-GAN[https://arxiv.org/abs/1802.05957] using spectral normalization, bigGAN[https://arxiv.org/abs/1809.11096] using bigger batch size. Some method changes the way the model works, including using progressive scale-growing GANs, which is called ProGAN[https://arxiv.org/abs/1710.10196]; using progressive pixel-growing models, which is called PixelCNN[https://arxiv.org/abs/1606.05328](this is not a GAN, but a kind of generative model, so we put here). Another improving way is adding controls to the result, such as cGANs[https://arxiv.org/abs/1411.1784], styleGANs[https://arxiv.org/abs/1812.04948].

(2) main paper introduction[xly]

- introduce the first two papers mentioned in writeup

Till now, we've read the first two papers mentioned in writeup, namely *Generative Adversarial Nets* (Goodfellow, 2014) and *3D Reconstruction of Incomplete Archaeological Objects Using Adversarial Network* (Hermoza, 2018).

From the first paper, we've learnt the essence of a GAN which is a combination of a generative model G that captures the data distribution, and a discriminative model D that estimates the probability that a sample came from the training data rather than G. We've also gone through the mathematics foundation of the global optimality and convergence of a GAN-based algorithm, from which we derived several training strategies including alternating between the training of D and G. The paper also opens our minds to further improvement of GAN structure such as introducing a variational auto-encoder in the generator to allow for the learning of the conditional variance.

The second one mainly illustrated a variant of GAN, namely the ORGAN(OR is short for object reconstruction), drawing a fundamental picture of 3D GAN. It gave detailed description regarding its structure(a 3D CNN with

skip-connections, as a **generator** on a Conditional GAN (CGAN) architecture. With two optimization targets: a mean absolute error (MAE) and an Improved Wasserstein GAN (IWGAN) loss) in additional to training details including the choice of hyper-parameters, which provided us with much reference and experience to rely on. Nevertheless, the dataset we use is quite different from the latter paper, which pushes us to design our own dataloader and consequently the exact training techniques need further experiments.

Part3 Our Approach

(1) Problem Definition

In a word, what we are going to do is to predict models of the complete pottaries, given models of the fragments. The 3D models are represented as voxels of shape $32\times32\times32$ or $64\times64\times64$.

It's worth noting that the task is a prediction task, instead of a generation task. The difference is that in out prediction task, the ground-truths are given, while in a generation task, they are not given. The task determined that our approach was fully-supervised.

Our task setting is almost the same as *"3D reconstruction of incomplete archaeological objects using a generative adversarial network" (Hermoza, 2018)*, with a slight difference that the inputs of our tasks were fragments (or combinations of fragments), while in Hermoza (2018)'s work, the inputs were randomly sampled from the complete models.

(2) Model Setting[wmq]

model layer structure/hyper parameters/loss function/optimizer/training strategy We build a generator32 and a discriminator32 in `utils/model.py` separately and combine them into a GAN in `training.py`. The generator takes the input fragments and the label and outputs the complete model. The discriminator takes the complete model and the label and outputs a score. It should be emphasized that label are needed for both the generator and the discriminator, which means that the GAN model we built is actually a conditional-GAN.

The generator is built as the picture shown: In the first branch we use 1 Encoder (named as encoderl) to convert the label into a 1024 dim feature vector. In the second branch, we use 5 sequentially connected Encoder layers (named encoder_i) to encode the input fragments into a 1024 dim feature vector. Then we concatenate these two convert them into a 1024 dim feature vector. After these step, we use 5 Decoder layers (named as decoder_i) to decode the feature vector into a 3D-voxel, which is the output of the generator. The discriminator is similar in the first two step, and convert the feature vector into a 1 dim feature vector, which is the final output of the discriminator.

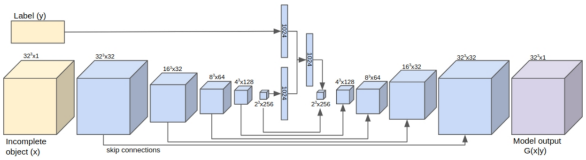


Figure 2. Network architecture for the generator.

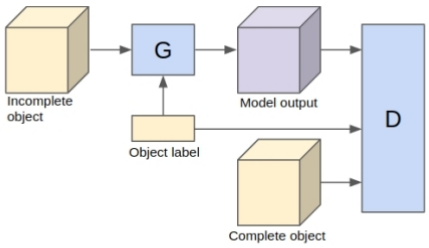


Figure 3. Reconstruction GAN architecture, conditioned on the object label.

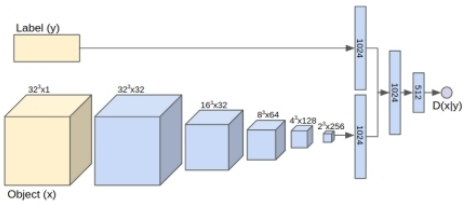


Figure 4. Network architecture for the discriminator.

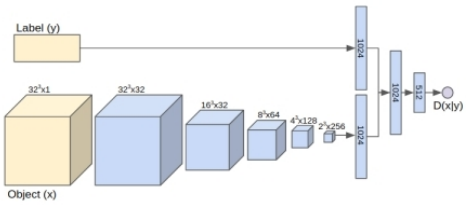
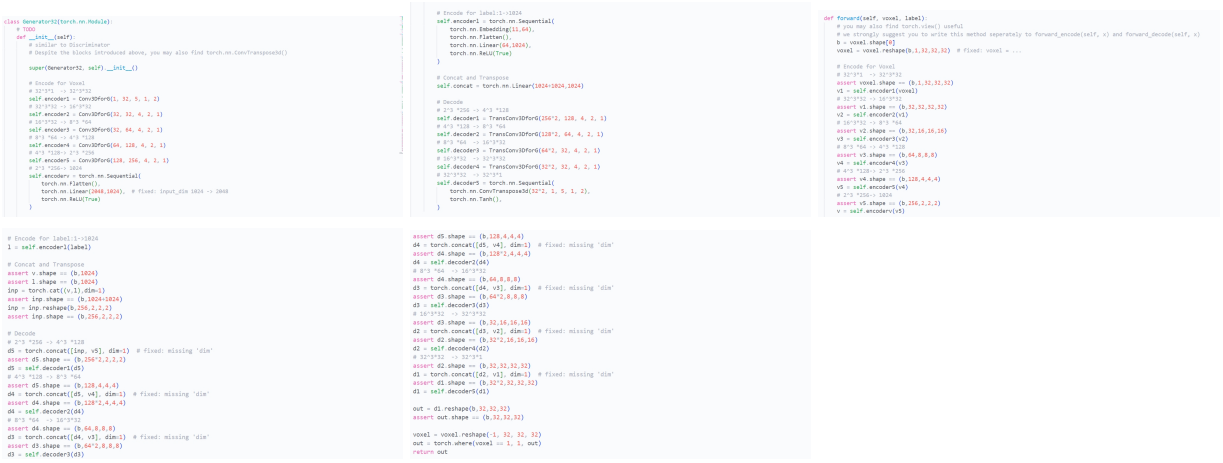


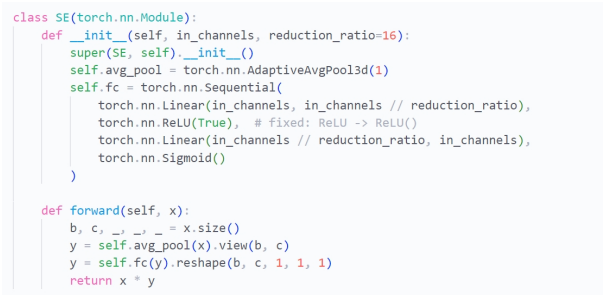
Figure 4. Network architecture for the discriminator.

picture in the paper



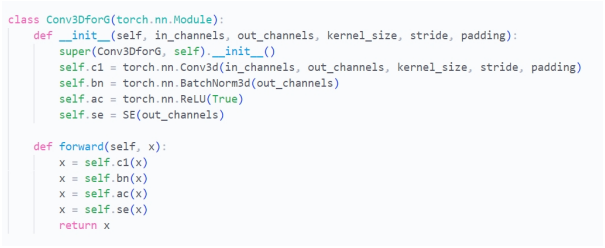
generator32 we build

discriminator32 we build



```python
class SE(torch.nn.Module):
    def __init__(self, in_channels, reduction_ratio=16):
        super(SE, self).__init__()
        self.avg_pool = torch.nn.AdaptiveAvgPool3d(1)
        self.fc = torch.nn.Sequential(
            torch.nn.Linear(in_channels, in_channels // reduction_ratio),
            torch.nn.ReLU(True),  # fixed: ReLU -> ReLU()
            torch.nn.Linear(in_channels // reduction_ratio, in_channels),
            torch.nn.Sigmoid()
        )

    def forward(self, x):
        b, c, _, _, _ = x.size()
        y = self.avg_pool(x).view(b, c)
        y = self.fc(y).reshape(b, c, 1, 1, 1)
        return x * y
```

```python
class Conv3DforD(torch.nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, stride, padding):
        super(Conv3DforD, self).__init__()
        self.c1 = torch.nn.Conv3d(in_channels, out_channels, kernel_size, stride, padding)
        self.ac = torch.nn.LeakyReLU(0.2)
        self.se = SE(out_channels)

    def forward(self, x):
        x = self.c1(x)
        x = self.ac(x)
        x = self.se(x)
        return x
```



```python
class Conv3DforG(torch.nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, stride, padding):
        super(Conv3DforG, self).__init__()
        self.c1 = torch.nn.Conv3d(in_channels, out_channels, kernel_size, stride, padding)
        self.bn = torch.nn.BatchNorm3d(out_channels)
        self.ac = torch.nn.ReLU(True)
        self.se = SE(out_channels)

    def forward(self, x):
        x = self.c1(x)
        x = self.bn(x)
        x = self.ac(x)
        x = self.se(x)
        return x
```

```python
class TransConv3DforG(torch.nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, stride, padding):
        super(TransConv3DforG, self).__init__()
        self.c1 = torch.nn.ConvTranspose3d(in_channels, out_channels, kernel_size, stride, padding)
        self.bn = torch.nn.BatchNorm3d(out_channels)
        self.ac = torch.nn.ReLU(True)
        self.se = SE(out_channels)

    def forward(self, x):
        x = self.c1(x)
        x = self.bn(x)
        x = self.ac(x)
        x = self.se(x)
        return x
```

```
SE layer/Encoder/Decoder we build
```

There're 2 points to note in this model: 1. Every encoder follows not only the 'Conv-Act-Norm' paradiam, but also follows a SE-layer. Decoders are the same except for replacing the 'Conv-layer' with 'TransConv-layer'. Details parameters are in the picture. 2. The generator use 'skip connection' technique which means directly concatenanting one feature map with a very late feature map.

Part4 Working Details

(1) Visualization[xly]

introduce what kind of visualization we did(maybe some pictures)

(2) Data Processing

a) the raw data

The raw data contained the voxel models of the fragments. The models were not larger than $64\times64\times64$. Each voxel of the models either belonged to a unique fragment or was empty.

There were 11 categories of pottaries indexed from 1 to 11. Each category contains several pottaries with different shapes, and for each pottary, voxel models of fragments with different number of fragments (at least 2 and no more than 17) are given.

b) pre-processing

To stack the voxel data into Torch arrays, we first padded the voxel models to ensure they were of the shape $64\times64\times64$. In detail, we check whether the size of the dimension is less than 64 for each dimension. If so, we inserted zeros to the back of the dimension.

We implemented two settings. For the low-resolution setting, we downsampled the voxel models by sampling the voxels with even indices (i.e. $0,2,\dots,63$). For the high-resolution setting, we did nothing.

For the training stage, we first shuffled the data. For each voxel model, we randomly selected several pieces of fragments and combined them as the input and took the complete model as the ground-truth. We guaranteed that the input contains at least one piece and not all the pieces.

For the test stage, to make the results stable, we only randomly sampled the pieces for each voxel model in the first epoch. And in the following epochs, we reused the inputs in the first epoch.

c) analysis

The same complete models were divided into different sets of fragments, which meant there were sereral data with the same ground-truths. This might break the property of IID.

However, it also allowed us to strengthen the dataset by randomly combining the fragments as inputs. We could generate a large number of combinations for each pottary, which contributed to preventing overfitting.

(3) Training Framework[wmq]

We used Pytorch to implement the training framework in training.py, named as `GAN_trainer`, shown in pictures below. In this model, we implements several functions, including args setting, data loading, model loading, model saving, model initialization, G_training, D_training, loss_drawing. In the main function, we initialize the model, load the data, and start the training process. We adapted the popular training way: update D every time and update G every 5 times. For better visualization, we also implemented a color progressing bar in console.

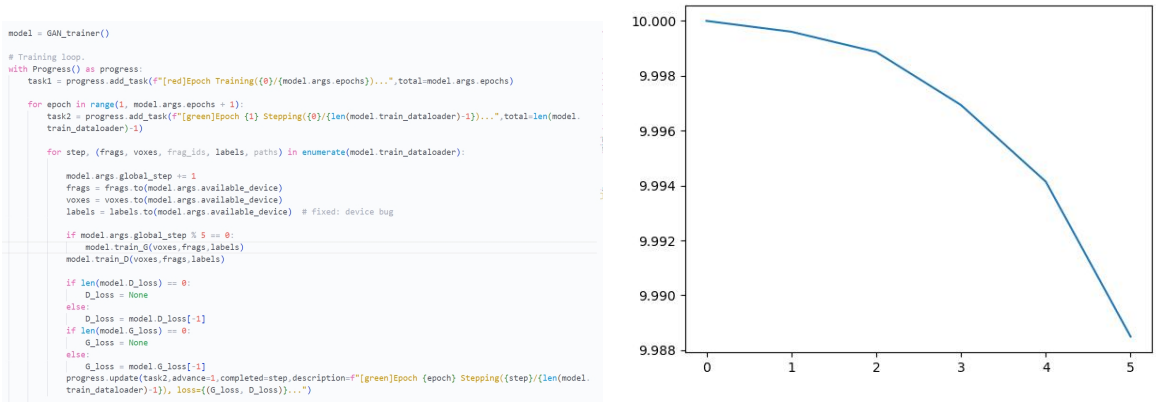
```python
class GAN_trainer:
    def __init__(self):
        self.init_Args()
        self.load_Data()
        self.init_Loss()
        self.G = Generator().to(self.args.available_device)  # fixed: define first
        self.D = Discriminator().to(self.args.available_device)
        self.G_optim = optim.AdamW(self.G.parameters(), lr=self.args.g_lr, betas=(self.args.g_beta1, self.args.g_beta2), eps=self.args.g_eps,
        weight_decay=self.args.g_weight_decay)
        self.D_optim = optim.AdamW(self.D.parameters(), lr=self.args.d_lr, betas=(self.args.d_beta1, self.args.d_beta2), eps=self.args.d_eps,
        weight_decay=self.args.d_weight_decay)
        self.load_Model()


    def init_Args(self):
    # Add hyperparameters
        parser = argparse.ArgumentParser()
        parser.add_argument('--train_vox_path', type=str, help="The path of the training dir.",default="./data/train")
        parser.add_argument('--test_vox_path', type=str, help="The path of the test dir.",default="./data/test")
        parser.add_argument('--hidden_dim', type=int, default=64, help="The hidden dim of GAN, or the resolution.")
        parser.add_argument('--g_lr', type=float, default=1e-4, help="The learning rate for AdamW of G.")  # Hyperparameters of AdamW for G.
        parser.add_argument('--g_beta1', type=float, default=0.9, help="Beta1 for AdamW of G.")
        parser.add_argument('--g_beta2', type=float, default=0.999, help="Beta2 for AdamW of G.")
        parser.add_argument('--g_eps', type=float, default=1e-8, help="Epsilon for AdamW of G.")
        parser.add_argument('--g_weight_decay', type=float, default=0.01, help="Weight decay for AdamW of G.")
        parser.add_argument('--d_lr', type=float, default=1e-4, help="The learning rate for AdamW of D.")  # Hyperparameters of AdamW for D.
        parser.add_argument('--d_beta1', type=float, default=0.9, help="Beta1 for AdamW of D.")
        parser.add_argument('--d_beta2', type=float, default=0.999, help="Beta2 for AdamW of D.")
        parser.add_argument('--d_eps', type=float, default=1e-8, help="Epsilon for AdamW of D.")
        parser.add_argument('--d_weight_decay', type=float, default=0.01, help="Weight decay for AdamW of D.")
        parser.add_argument('--batch_size', type=int, default=64, help="The batch size for both training and test.")
        parser.add_argument('--epochs', type=int, help="Total epochs of training.",default=1)
        parser.add_argument('--available_device', type=str, help="available device",default="cuda:0" if torch.cuda.is_available() else "cpu")
        parser.add_argument('--model_name', type=str, help="Name of the model.",default='gan32')
        parser.add_argument('--load_path', type=str, help="Where to load the model.",default=None)
        parser.add_argument('--save_path', type=str, help="Where to save the model.",default=None)
        parser.add_argument('--global_step', type=int, help="Global step of training.",default=0)
        self.args = parser.parse_args()
        self.args.save_path = None or "models/" + self.args.model_name + ".pt"
        print(self.args.available_device)
        print("Args Resolved Succesfully")
```

```python
def train_D(self, voxel_whole, voxel_frag, label):
    self.D_optim.zero_grad()
    self.G_optim.zero_grad()
    voxel_pred = self.G(voxel_frag, label)
    voxel_blend = self.blend(voxel_pred, voxel_whole).requires_grad_(True)
    loss_real = self.D_loss1(self.D(voxel_whole,label))
    loss_fake = self.D_loss2(self.D(voxel_pred,label))
    loss_grad = self.D_loss3(self.D(voxel_blend,label),voxel_blend)
    loss = loss_real + loss_fake + loss_grad
    loss_cpu = loss
    self.D_loss.append(loss_cpu.to('cpu').detach().numpy())
    loss.backward()
    self.D_optim.step()
    # self.save_Model()
    # self.draw_loss()

def train_G(self, voxel_whole, voxel_frag, label):
    voxel_pred = self.G(voxel_frag, label)
    self.D_optim.zero_grad()
    self.G_optim.zero_grad()
    loss_diff = self.G_loss1(voxel_pred, voxel_whole)
    loss_pred = self.G_loss2(self.D(voxel_pred,label))
    loss = loss_diff + loss_pred
    loss_cpu = loss
    self.G_loss.append(loss_cpu.to('cpu').detach().numpy())
    loss.backward()
    self.G_optim.step()
    # self.save_Model()
```

```python
def save_Model(self, path=None):
    if path is None:
        path = self.args.save_path
    torch.save({
        'model-G':self.G.state_dict(),
        'loss-G':self.G_loss,
        'optim-G':self.G_optim.state_dict(),
        'model-D':self.D.state_dict(),
        'loss-D':self.D_loss,
        'optim-D':self.D_optim.state_dict(),
        'args':self.args
        },
        path)
    print(f"Model Saved to {path} Successfully!")

def draw_loss(self):
    plt.figure()
    plt.plot(self.G_loss)
    plt.savefig(f"lossPics/{self.args.model_name}-G.jpg")
    plt.figure()
    plt.plot(self.D_loss)
    plt.savefig(f"lossPics/{self.args.model_name}-D.jpg")
```

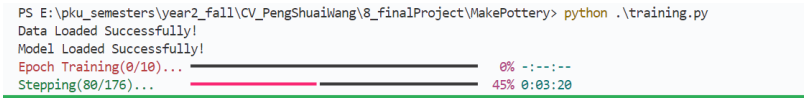
```python
def load_Model(self):
    try:
        checkpoint = torch.load(self.args.load_path)
        self.G.load_state_dict(checkpoint['model-G'])
        self.G_loss = checkpoint['loss-G']
        self.G_optim.load_state_dict(checkpoint['optim-G'])
        self.D.load_state_dict(checkpoint['model-D'])
        self.D_loss = checkpoint['loss-D']
        self.D_optim.load_state_dict(checkpoint['optim-D'])
        self.args = checkpoint['args']
        print(f"Model Loaded from '{self.args.load_path}' Successfully!")
    except:
        self.G_loss = []
        self.D_loss = []
        print(f"Model Load Failed from '{self.args.load_path}'! Using New Initialization!")

def load_Data(self):
    train_dataset = FragmentDataset(self.args.train_vox_path, "train", dim_size=self.args.hidden_dim)
    test_dataset = FragmentDataset(self.args.test_vox_path, "test", dim_size=self.args.hidden_dim)
    self.train_dataloader = data.DataLoader(train_dataset, batch_size=self.args.batch_size, shuffle=True)
    self.test_dataloader = data.DataLoader(test_dataset, batch_size=self.args.batch_size, shuffle=False)
    print("Data Loaded Successfully!")

def init_Loss(self):
    self.G_loss1 = nn.L1Loss()
    self.G_loss2 = lambda y_pred: torch.mean(y_pred)
    self.D_loss1 = lambda y_pred: torch.mean(y_pred)
    self.D_loss2 = lambda y_pred: -torch.mean(y_pred)
    self.D_loss3 = gradient_penalty
```

class GAN_trainer and some of its member functions



```
model = GAN_trainer()

# Training loop.
with Progress() as progress:
    task1 = progress.add_task(f"[red]Epoch Training({0}/{model.args.epochs})...",total=model.args.epochs)

    for epoch in range(1, model.args.epochs + 1):
        task2 = progress.add_task(f"[green]Epoch {1} Stepping({0}/{len(model.train_dataloader)-1})...",total=len(model.
        train_dataloader)-1)

        for step, (frags, voxes, frag_ids, labels, paths) in enumerate(model.train_dataloader):

            model.args.global_step += 1
            frags = frags.to(model.args.available_device)
            voxes = voxes.to(model.args.available_device)
            labels = labels.to(model.args.available_device)  # fixed: device bug

            if model.args.global_step % 5 == 0:
                model.train_G(voxes,frags,labels)
            model.train_D(voxes,frags,labels)

            if len(model.D_loss) == 0:
                D_loss = None
            else:
                D_loss = model.D_loss[-1]
            if len(model.G_loss) == 0:
                G_loss = None
            else:
                G_loss = model.G_loss[-1]
            progress.update(task2,advance=1,completed=step,description=f"[green]Epoch {epoch} Stepping({step}/{len(model.
            train_dataloader)-1}), loss={(G_loss, D_loss)}...")
```


left:training process; right:loss picture in one trial



```
PS E:\pku_semesters\year2_fall\CV_PengShuaiWang\8_finalProject\MakePottery> python .\training.py
Data Loaded Successfully!
Model Loaded Successfully!
Epoch Training(0/10)...  ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━  0% -:--:--
Stepping(80/176)...      ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━  45% 0:03:20
```

colorful progressing bar in console

(4) Remote Environment Setting[xly]

introduce how to set up environment in remote server(mention this time, won't appear in final report)