1 项目概要

不管是在人工智能的科研中还是项目成果的筹备中，数据是十分重要且难得的，尤其是更加复杂的多维数据。因此，我们目标在这个课程设计中，能够**实现将一组数据转化为与其具有物理相关性的数据**，为了使项目更加具有扩展性和创新性，我们采用三维时变体数据，就是具有时间片的三维数据。我们的目标类似于最终能够实现将空间体中一段时间的温度数据转化为这个空间体中这段时间内的气压数据。

由于没有一定空间体内气压随时间变化的三维时变体数据，因此我们选择采用空气电离产生的单原子气体数据作为我们模型的训练和测试数据（该气体电离数据是一套三维时变体数据，来自于科研组）。不同的气体原子有不同的物理性质，在空气的电离中，会产生 H 离子和 He 离子以及各种气体离子，其中 H 离子和 He 离子就具有这相似而不同的物理特性，它们之间的特征差异就类似于空间中温度和压强的特征差异。因此我们采用这个气体电离数据来作为我们机器学习研究的数据，通过生成对抗网络实现特征学习，**最终根据特征能够实现将 H 离子的三维时变数据转化为可以以假乱真的“He 离子的三维时变体数据”**。

这个课程设计不仅考验了我们对于复杂数据的处理能力，还有对神经网络的处理能力以及对于机器学习模型的掌控能力，对于我们整组成员来来说都是一个极大的挑战，大大锻炼了我们的模型分析能力与代码能力等等。

2 数据集的可视化展示

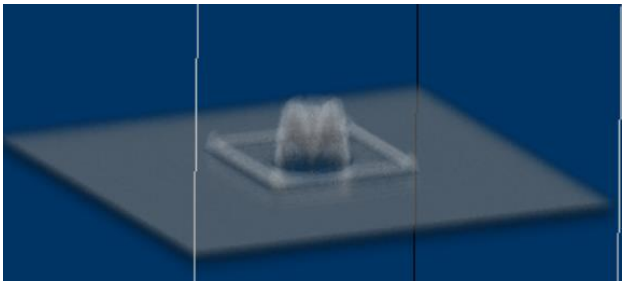
2.1 两种不同类型的数据解释

我们模型中所用的 source 数据为 H 离子三维时变数据，目标生成的 target 数据为 He 离子的三维时变数据，在训练过程中我们会对模型输入 H 离子和 He 离子的真实数据，让模型进行特征学习。

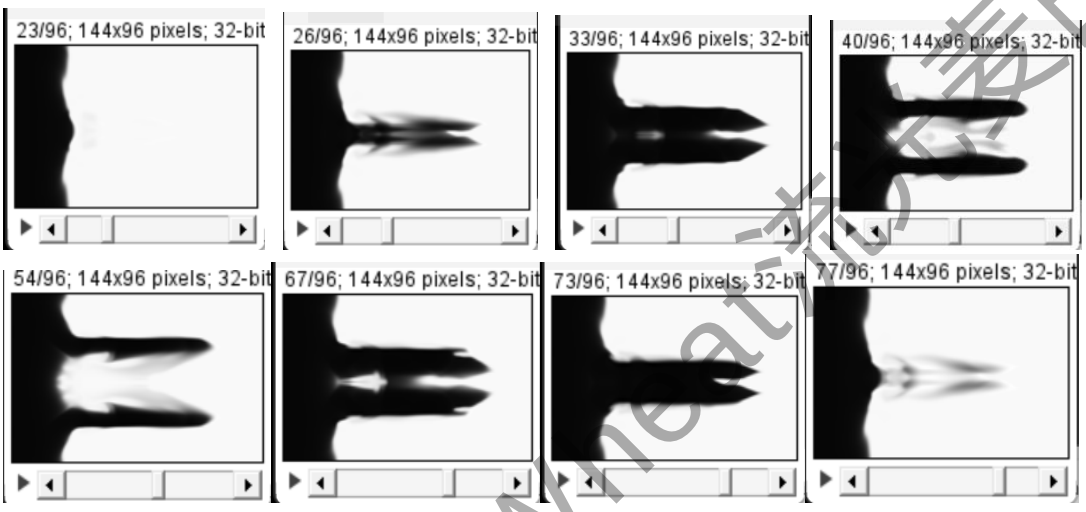
之后在测试过程中，我们让该模型生成能够以假乱真的“He 离子三维时变数据”。以下是对 H 离子和 He 离子三维时变数据的展示，我们从 H 离子和 He 离子数据集中都选取了一个时间片内变化较小的数据和一个时间片内变化较大的数据。（**H 离子和 He 离子两个数据集中编号相同的数据为同一团气体在同一时间中电离喷发的 H 离子和 He 离子**）

（H 离子数据的时间片展示）来自于 H 离子数据集中的 normInten_0005

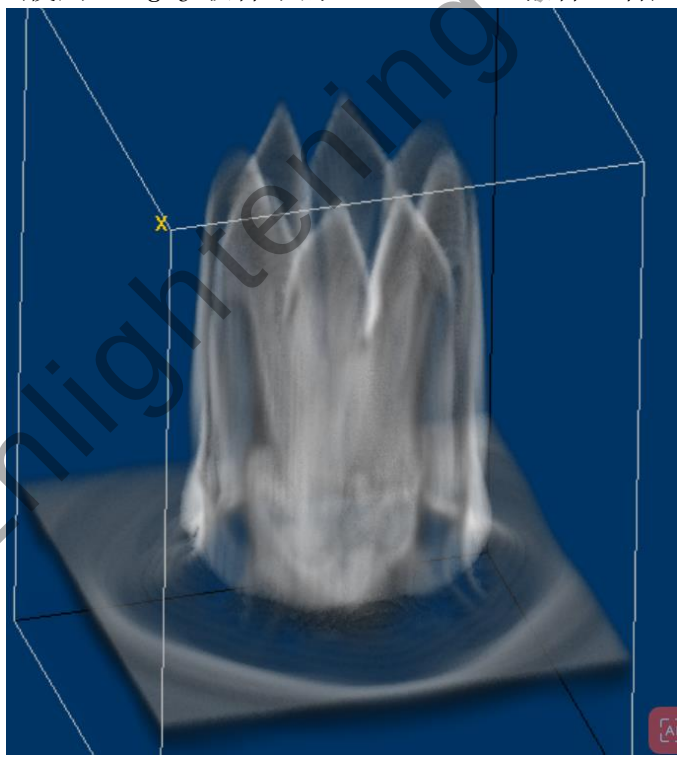
（使用 imageJ 软件中的 Volume View 插件查看）



（H 离子数据的时间片展示）来自于 H 离子数据集中的 normInten_0075

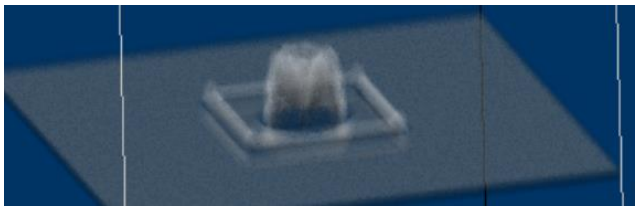


（使用 imageJ 软件中的 Volume View 插件查看）

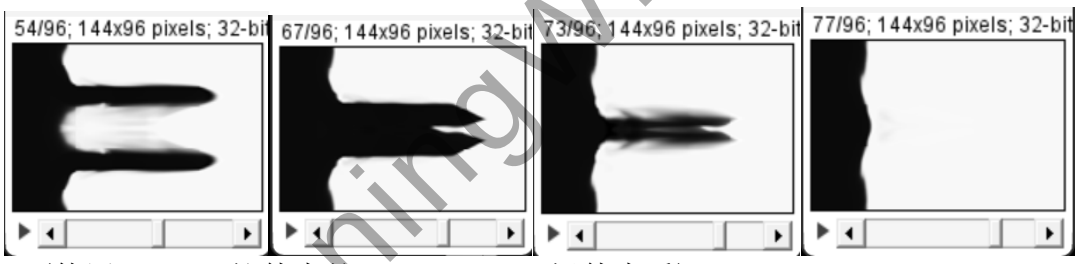


（He 离子数据的时间片展示）来自于 He 离子数据集中的 normInten_0005

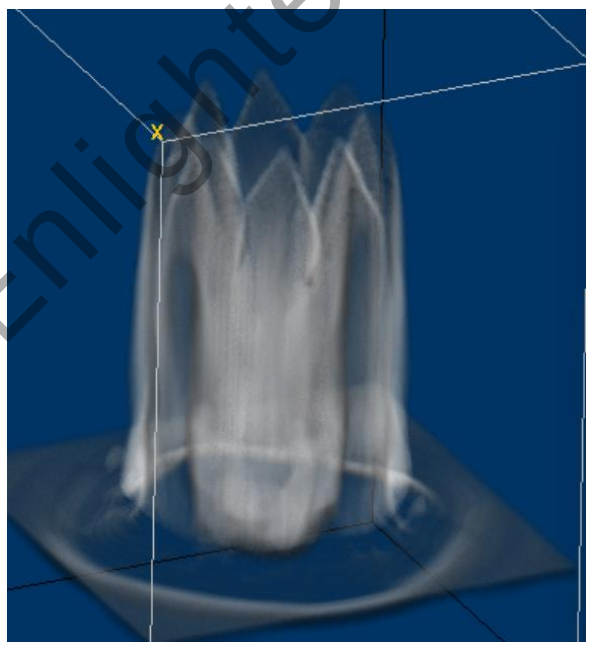
（使用 imageJ 软件中的 Volume View 插件查看）



（He 离子数据的时间片展示）来自于 H 离子数据集中的 normInten_0075



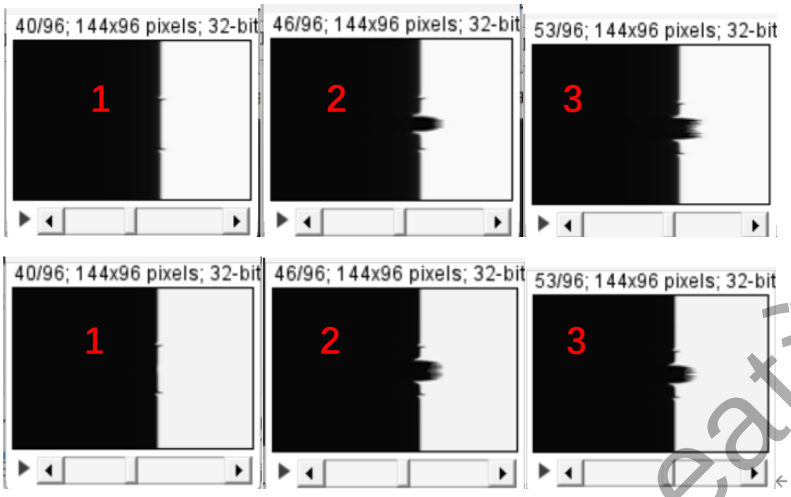
（使用 imageJ 软件中的 Volume View 插件查看）



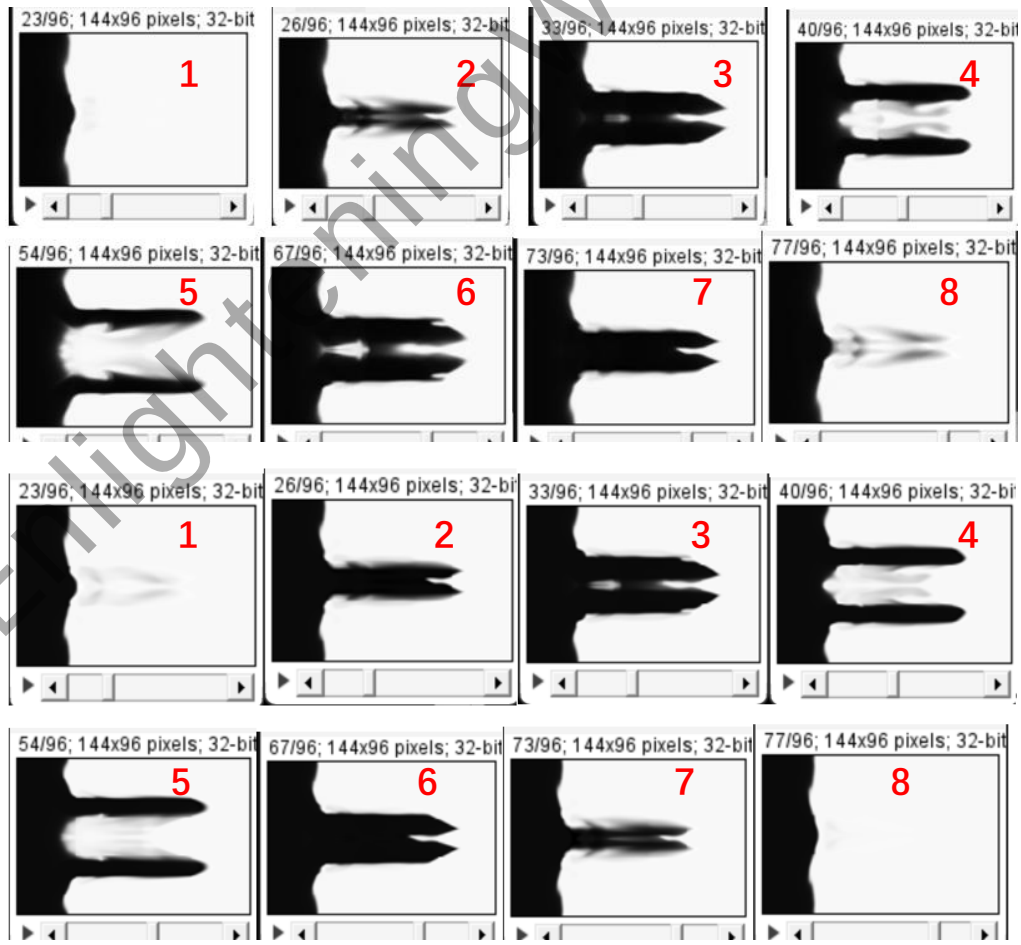
2.2 数据的特征比较

由于 H 和 He 在原子结构相似，所以在电离过程中所喷发的能量图像具有许多的相似之处，而 H 和 He 本身又是不同的原子，所以它们的能量图像即便很相似也不会完全相同。我们相信 H 和 He 的电离数据的相似之处和不同之处都是有迹可循的，因此这是相当好的一组可以应用于机器学习特征训练以及生成性对抗网络的数据。以下是相同时间片中 H 和 He 原子在电离中的能量喷发图像：

（上面是 H，下面是 He）



（上面是 H，下面是 He）

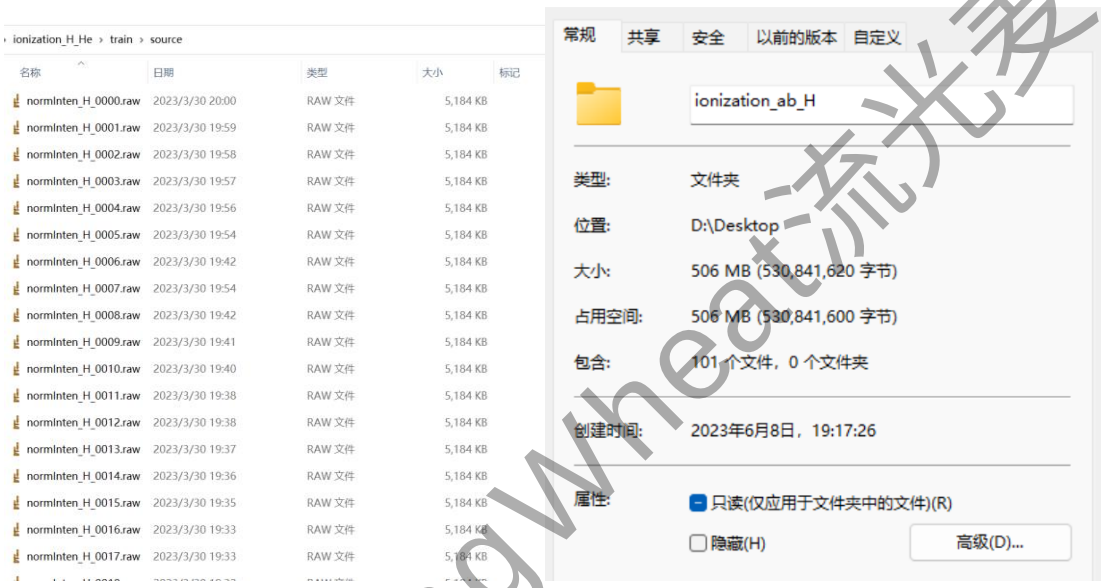


由上图可见，H 离子和 He 离子电离图像中的微小差别，十分适用于本课程设计“实现将一组数据转化为与其具有物理相关性的数据”模型的训练。

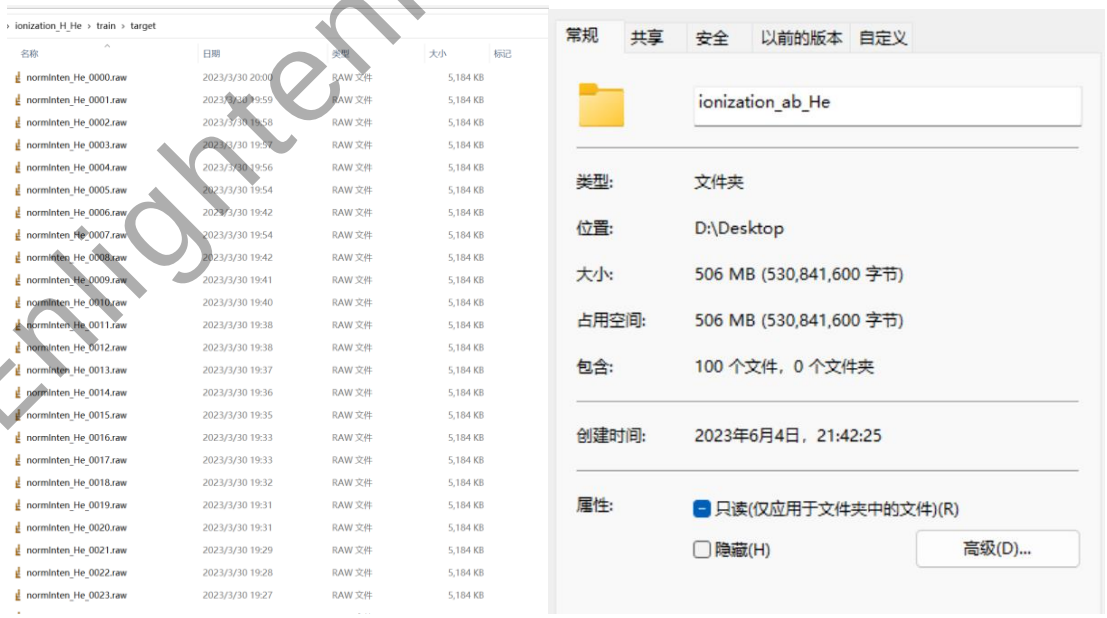
2.3 数据集的体量展示

本课程设计中所用到的 H 离子和 He 离子的有效数据各共有 100 个，每一个数据集的长宽高大小为[96,96,144],这个数据量在 GAN 模型的训练中并不算多，但在实验过程中我们会运用数据切分以及增大 epoch 等方式大大增加我们模型训练过程中能用到的数据量。

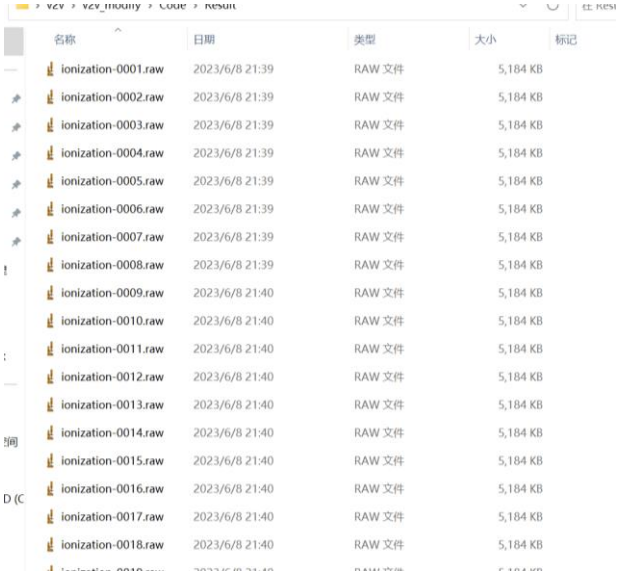
H 离子数据集



He 离子数据集



由 GAN 模型生成的以假乱真的 He 离子结果



名称	日期	类型	大小	标记
ionization-0001.raw	2023/6/8 21:39	RAW 文件	5,184 KB	
ionization-0002.raw	2023/6/8 21:39	RAW 文件	5,184 KB	
ionization-0003.raw	2023/6/8 21:39	RAW 文件	5,184 KB	
ionization-0004.raw	2023/6/8 21:39	RAW 文件	5,184 KB	
ionization-0005.raw	2023/6/8 21:39	RAW 文件	5,184 KB	
ionization-0006.raw	2023/6/8 21:39	RAW 文件	5,184 KB	
ionization-0007.raw	2023/6/8 21:39	RAW 文件	5,184 KB	
ionization-0008.raw	2023/6/8 21:39	RAW 文件	5,184 KB	
ionization-0009.raw	2023/6/8 21:40	RAW 文件	5,184 KB	
ionization-0010.raw	2023/6/8 21:40	RAW 文件	5,184 KB	
ionization-0011.raw	2023/6/8 21:40	RAW 文件	5,184 KB	
ionization-0012.raw	2023/6/8 21:40	RAW 文件	5,184 KB	
ionization-0013.raw	2023/6/8 21:40	RAW 文件	5,184 KB	
ionization-0014.raw	2023/6/8 21:40	RAW 文件	5,184 KB	
ionization-0015.raw	2023/6/8 21:40	RAW 文件	5,184 KB	
ionization-0016.raw	2023/6/8 21:40	RAW 文件	5,184 KB	
ionization-0017.raw	2023/6/8 21:40	RAW 文件	5,184 KB	
ionization-0018.raw	2023/6/8 21:40	RAW 文件	5,184 KB	

3 数据预处理

3.1 数据集标准化

我们使用的每一个三维体数据的长宽高大小都是[96,96,144]，为了将图像信息转化为计算方便的矩阵，我们用 np.zeros 初始化一个四维矩阵 s 用来放入图像信息，其中第一个维度用于索引不同的样本。我们用 np.fromfile 读取图像的 raw 源文件，并将其进行归一化处理，整理成我们需要的维度后放入目标数据列表。

在读取数据中我们用了循环和 format 读取各个标号的文件。

```
def ReadData(self):  # 定义读取数据方法
    self.source = []  # 初始化源数据列表
    self.target = []  # 初始化目标数据列表
    for i in self.train_samples:  # 对于每个训练样本
        print(i)
        s = np.zeros((1, self.dim[0], self.dim[1], self.dim[2]))  # 初始化源数据矩阵
        d = np.fromfile(self.s + '{:04d}'.format(i) + '.raw', dtype='<f')  # 读取训练数据文件

        d = 2 * (d - np.min(d)) / (np.max(d) - np.min(d)) - 1  # 将源数据进行归一化处理
        d = d.reshape(self.dim[2], self.dim[1], self.dim[0]).transpose()  # 将源数据进行形状变换
        s[0] = d  # 将变换后的源数据赋值给s
        self.source.append(s)  # 将s添加到目标数据列表中

    for i in self.test_samples:  # 对于每个测试样本
        print(i)
        t = np.zeros((1, self.dim[0], self.dim[1], self.dim[2]))  # 初始化目标数据矩阵
        o = np.fromfile(self.t + '{:04d}'.format(i) + '.raw', dtype='<f')  # 读取测试数据文件
        o = 2 * (o - np.min(o)) / (np.max(o) - np.min(o)) - 1  # 将目标数据进行归一化处理
        o = o.reshape(self.dim[2], self.dim[1], self.dim[0]).transpose()  # 将目标数据进行形状变换
        t[0] = o  # 将变换后的目标数据赋值给t
        self.target.append(t)  # 将t添加到目标数据列表中

    self.source = np.asarray(self.source)  # 将源数据列表转为numpy数组
    self.target = np.asarray(self.target)  # 将目标数据列表转为numpy数组
```

3.2 数据集处理

为了机器学习更好提取特征，我们要对原数据标准化后再处理一下以成为良好的训练数据。对于扩大训练中能使用的数据集，采用的策略是进行数据裁剪，根据指定的裁剪尺寸，从源数据（self.source）和目标数据（self.target）中裁剪出多个子数据。每个子数据对应一个训练样本，通过随机裁剪可以增加数据的多样性。将裁剪后的源数据和目标数据存储在 o 和 a 列表中。

源数据的尺寸为[96,96,144]，我们定义的裁剪尺寸为 cropsize = [64, 64, 96]，在每个原数据默认的裁剪次数为 4，裁剪位置是随机选取的且一对源数据和目标数据所裁剪的位置是一样的，我们最后将这一对数据一一对应放入数据集中方便后续的机器学习。

同时为了对比出数据裁剪后的效果，TrainingData 方法中还做了没有裁剪操作的数据处理。

```
def TrainingData(self): # 定义获取训练数据的方法
    if self.crop == 'yes':  # 如果需要裁剪数据
        a = []  # 初始化a列表
        o = []  # 初始化o列表
        for k in range(len(self.source)):  # 对于每个训练样本
            n = 0  # 初始化n为0
            while n < self.croptimes:  # 如果n小于裁剪次数
                if self.dim[0] == self.cropsize[0]:  # 如果第一维大小等于裁剪大小
                    x = 0  # x赋值为0
                else:
                    x = np.random.randint(0, self.dim[0] - self.cropsize[0])  # 否则，x为随机数

                if self.dim[1] == self.cropsize[1]:  # 同理
                    y = 0
                else:
                    y = np.random.randint(0, self.dim[1] - self.cropsize[1])

                if self.dim[2] == self.cropsize[2]:
                    z = 0
                else:
                    z = np.random.randint(0, self.dim[2] - self.cropsize[2])

                c0 = self.source[k][:, x:x + self.cropsize[0], y:y + self.cropsize[1],
                     z:z + self.cropsize[2]]  # 裁剪源数据
                o.append(c0)  # 添加到o列表

                # 扩充目标数据以匹配源数据的数量
                idx = k % len(self.target)  # 计算目标数据的索引
                c1 = self.target[idx][:, x:x + self.cropsize[0], y:y + self.cropsize[1],
                     z:z + self.cropsize[2]]  # 裁剪目标数据
                a.append(c1)  # 添加到a列表

                n += 1  # n加1

        o = np.asarray(o)  # 将o列表转为numpy数组
        a = np.asarray(a)  # 将a列表转为numpy数组
        a = torch.FloatTensor(a)  # 将a转为FloatTensor
        o = torch.FloatTensor(o)  # 将o转为FloatTensor
    else:
        a = torch.FloatTensor(self.target)  # 将目标数据转为FloatTensor
        o = torch.FloatTensor(self.source)  # 将源数据转为FloatTensor

    dataset = torch.utils.data.TensorDataset(o, a)  # 构建数据集
    train_loader = DataLoader(dataset=dataset, batch_size=1, shuffle=True)  # 创建DataLoader
    return train_loader  # 返回DataLoader
```

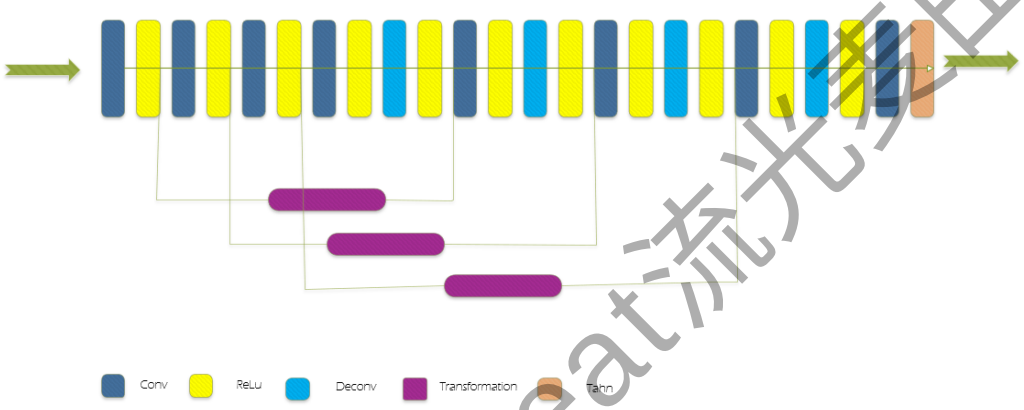
4 机器学习

4.1 模型介绍

我们的目标是将一组变量序列转化为与之有物理相关性的变量序列，该过程有三个阶段：特征学习、变量转换模型构建以及变量转换。

4.1.1 生成器构造

生成器由三个模块组成。



（1）特征提取模块利用四个 Conv 层从输入变量中提取特征。每个 Conv 将输入量降低一半，并遵循一个 ReLU 来加速训练和提高模型性能。

```
self.conv1 = nn.Conv3d(1, init_channels, 4, 2, 1)
self.conv2 = nn.Conv3d(init_channels, 2 * init_channels, 4, 2, 1)
self.conv3 = nn.Conv3d(2 * init_channels, 4 * init_channels, 4, 2, 1)
self.conv4 = nn.Conv3d(4 * init_channels, 8 * init_channels, 4, 2, 1)

x1 = F.relu(self.conv1(x))
x2 = F.relu(self.conv2(x1))
x3 = F.relu(self.conv3(x2))
x4 = F.relu(self.conv4(x3))
```

（2）特征转换模块利用三个转换块将特征从源变量以不同的尺度转换为目标变量，并输入到变量转换模块中。每个转换块包括两条路径。同时还在解码器部分，采用了残差连接的方式将编码器的特征图与解码器的特征图相结合。这种连接方式可以帮助信息传递和梯度流动，避免信息丢失和梯度消失问题。

```
self.b3 = Block(inchannels=4 * init_channels, outchannels=4 * init_channels, dropout=False, kernel=3,
                bias=False, depth=2, mode='same', factor=2)
self.b2 = Block(inchannels=2 * init_channels, outchannels=2 * init_channels, dropout=False, kernel=3,
                bias=False, depth=2, mode='same', factor=2)
self.b1 = Block(inchannels=init_channels, outchannels=init_channels, dropout=False, kernel=3, bias=False,
                depth=2, mode='same', factor=2)
```

```
torch.cat((self.b3(x3), u1), dim=1)
torch.cat((self.b2(x2), u2), dim=1)
```

```
torch.cat((self.b1(x1), u3), dim=1)
```

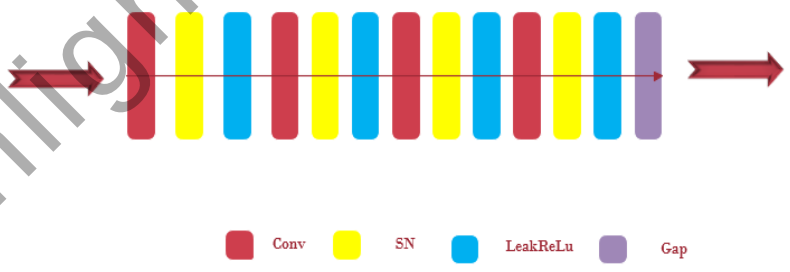
（3）变量转换模块利用四个 DeConv 层和四个 Conv 层，然后使用 ReLU 将特征映射到输出变量域。每个 DeConv 都会升级输入两次。此外，在每个 DeConv 层之后，我们将来自 DeConv 和特征转换阶段的输出堆叠在一起，并送入一个 Conv 层。两个堆叠的输出共享相同的规模，并且 Conv 层不会改变输入的规模。

```
### upsample
self.deconv4 = nn.ConvTranspose3d(8 * init_channels, 4 * init_channels, 4, 2, 1)
self.conv_u4 = nn.Conv3d(8 * init_channels, 4 * init_channels, 3, 1, 1)
self.deconv3 = nn.ConvTranspose3d(4 * init_channels, 2 * init_channels, 4, 2, 1)
self.conv_u3 = nn.Conv3d(4 * init_channels, 2 * init_channels, 3, 1, 1)
self.deconv2 = nn.ConvTranspose3d(2 * init_channels, init_channels, 4, 2, 1)
self.conv_u2 = nn.Conv3d(2 * init_channels, init_channels, 3, 1, 1)
self.deconv1 = nn.ConvTranspose3d(init_channels, init_channels // 2, 4, 2, 1)
self.conv_u1 = nn.Conv3d(init_channels // 2, 1, 3, 1, 1)

u1 = F.relu(self.deconv4(x4))
u1 = F.relu(self.conv_u4(torch.cat((self.b3(x3), u1), dim=1)))
u2 = F.relu(self.deconv3(u1))
u2 = F.relu(self.conv_u3(torch.cat((self.b2(x2), u2), dim=1)))
u3 = F.relu(self.deconv2(u2))
u3 = F.relu(self.conv_u2(torch.cat((self.b1(x1), u3), dim=1)))
u4 = F.relu(self.deconv1(u3))
out = self.conv_u1(u4)
out = torch.tanh(out)
return out
```

这一系列的上采样操作将逐渐恢复输入体积的细节信息，并在最后的卷积层 self.conv_u1 中生成最终的输出体积数据。

4.1.2 鉴别器构造



判别器包括四个 Conv 层，四个归一化（SN）层，和一个全局平均池化（GAP）层。每个 Conv 将输入量缩小一半，SN 之后用 Conv 来规范化 Conv 中的权重，以实现训练稳定。在每个 Conv 层后，应用泄漏的 ReLU 激活（$\alpha = 0.2$）。最后，利用一个 GAP 将输出压缩成一个具有 $1\times1\times1\times1$ 的张量。GAP 后不添加任何激活函数。

```python
class Dis(nn.Module):
    # Dis模型的定义，主要用于鉴别器部分
    def __init__(self):
        super(Dis, self).__init__()
        ### downsample
        self.conv1 = spectral_norm(nn.Conv3d(1, 32, 4, 2, 1), eps=1e-4)
        self.conv2 = spectral_norm(nn.Conv3d(32, 64, 4, 2, 1), eps=1e-4)
        self.conv3 = spectral_norm(nn.Conv3d(64, 128, 4, 2, 1), eps=1e-4)
        self.conv4 = spectral_norm(nn.Conv3d(128, 1, 4, 2, 1), eps=1e-4)
        self.ac = nn.LeakyReLU(0.2, inplace=True)

    def forward(self, x):
        x1 = self.ac(self.conv1(x))
        x2 = self.ac(self.conv2(x1))
        x3 = self.ac(self.conv3(x2))
        x4 = self.ac(self.conv4(x3))
        x5 = F.avg_pool3d(x4, x4.size()[2:])
        return [x1, x2, x3, x4], x5.view(-1)
```

鉴别器模型通过学习从输入数据中提取特征，并将其传递给全连接层等后续层进行判别。

4.2 机器学习过程

4.2.1 模型训练前准备

使用 Kaiming 函数为模型的权重进行适当的初始化，以提高模型的训练效果和收敛速度。weights_init_kaiming 接受一个模型的参数 m 作为输入。函数根据模型的不同类型（卷积层、线性层、批归一化层）来进行权重初始化。具体使用 Kaiming 初始化方法，也称为"He 初始化"，该方法适用于使用 ReLU 激活函数的神经网络。

```python
def weights_init_kaiming(m):
    # 权重初始化函数，使用Kaiming方法初始化权重
    classname = m.__class__.__name__
    if classname.find("Conv") != -1:
        # 如果是卷积层，使用kaiming_uniform_方法初始化权重
        init.kaiming_uniform_(m.weight.data)
    elif classname.find("Linear") != -1:
        # 如果是线性层，使用kaiming_uniform_方法初始化权重
        init.kaiming_uniform_(m.weight.data)
    elif classname.find("BatchNorm") != -1:
        # 如果是批量归一化层，使用正态分布初始化权重，均值为1，标准差为0.02，偏置设置为0
        init.normal_(m.weight.data, 1.0, 0.02)
        init.constant_(m.bias.data, 0.0)
```

4.2.2 生成器 G 和判别器 D 的训练

模型训练前各项参数的准备设置

```
# 创建存放损失的文件
loss_curve = open('ionization_ab_H/Exp/' + 'loss-' + args.dataset + '.txt', 'w')
# 设置设备为GPU或CPU
device = torch.device("cuda:0" if args.cuda else "cpu")
# 为生成器和判别器设置优化器
optimizer_G = optim.Adam(G.parameters(), lr=args.lr_G, betas=(0.9, 0.999))
optimizer_D = optim.Adam(D.parameters(), lr=args.lr_D, betas=(0.5, 0.999))
# 创建MSE损失函数
L2 = nn.MSELoss()
critic = 1
```

增大 epoch 进行训练，最后我们选择使用了 2000 个 epoch，就是将每一对源数据与目标数据训练了 2000 次。

```
# 迭代训练
for itera in range(1, 2000 + 1):
```

生成器训练过程

首先，使用 train_loader 迭代器遍历训练数据集，其中每次迭代返回一个源数据 i 和目标数据 o 的批次（batch）。接下来，固定生成器 G 的参数，更新鉴别器 D 的参数。使用真实数据进行训练和生成的假数据进行训练，计算真实数据和生成数据在鉴别器 D 上的输出，并通过均方误差损失函数（L2 Loss）计算真实数据和生成数据的损失。然后将真实数据和生成数据的损失取平均作为总损失，并进行反向传播和优化。

```
############
# 更新G网络 #
############
for p in G.parameters():
    p.requires_grad = True
for p in D.parameters():
    p.requires_grad = False
for j in range(1, 1 + 1):
    optimizer_G.zero_grad()
    label_real = Variable(torch.full((batch,), 1.0, device=device))
    fake_data = G(i)

    features_fake, output_real = D(fake_data)
    features_real, _ = D(o)

    # 对抗性损失
    L_adv = L2(output_real, label_real)
```

```
    # 内容损失
    L_c = L2(fake_data, o)

    # 特征损失
    L_p = L2(features_fake[0], features_real[0]) + L2(features_fake[1], features_real[1]) + L2(
        features_fake[2], features_real[2]) + L2(features_fake[3], features_real[3])

    # 总损失
    error = 1e-3 * L_adv + 1 * L_c + 1e-2 * L_p
    error.backward()
    loss_G += error.item()
    optimizer_G.step()
for p in D.parameters():
    p.requires_grad = True
```

鉴别器训练过程

固定鉴别器 D 的参数，更新生成器 G 的参数。首先生成假数据，然后通过鉴别器 D 计算生成数据的输出特征和真实数据的输出特征。计算总损失包括对抗性损失（L_adv）、内容损失（L_c）和特征损失（L_p）。对抗性损失是生成数据在鉴别器 D 上的输出与真实标签之间的均方误差。内容损失是生成数据与目标数据之间的均方误差。特征损失是生成数据在不同层级特征上与目标数据之间的均方误差。然后将总损失进行反向传播和优化。

在更新生成器 G 和鉴别器 D 的参数之前，通过设置 p.requires_grad 来控制是否对参数进行梯度计算，从而区分两个网络的参数更新过程。

最后，将鉴别器 D 的参数重新设置为可计算梯度，以便在下一次更新生成器 G 时使用。

```
#############
# 更新D网络 #
#############
for j in range(1, critic + 1):
    optimizer_D.zero_grad()
    # 使用真实数据进行训练
    label_real = Variable(torch.full((batch,), 1.0, device=device))
    _, output_real = D(o)
    real_loss = L2(output_real, label_real)

    # 使用生成的假数据进行训练
    fake_data = G(i)
    label_fake = Variable(torch.full((batch,), 0.0, device=device))

    _, output_fake = D(fake_data)
    fake_loss = L2(output_fake, label_fake)

    loss = 0.5 * (real_loss + fake_loss)

    loss.backward()
    loss_D += loss.mean().item()
    optimizer_D.step()
```

由 GAN 网络生成的模型展示

ionization-1-V2V.pth	2023/6/8 18:50	PTH 文件	8,830 KB
ionization-10-V2V.pth	2023/6/3 21:37	PTH 文件	8,829 KB
ionization-20-V2V.pth	2023/6/4 9:53	PTH 文件	8,829 KB
ionization-30-V2V.pth	2023/6/4 10:14	PTH 文件	8,829 KB
ionization-40-V2V.pth	2023/6/4 11:24	PTH 文件	8,829 KB
ionization-50-V2V.pth	2023/6/4 11:35	PTH 文件	8,829 KB
ionization-60-V2V.pth	2023/6/4 11:45	PTH 文件	8,829 KB
ionization-70-V2V.pth	2023/6/4 11:56	PTH 文件	8,829 KB
ionization-80-V2V.pth	2023/6/4 12:07	PTH 文件	8,829 KB
ionization-90-V2V.pth	2023/6/4 12:17	PTH 文件	8,829 KB
ionization-1790-V2V.pth	2023/6/4 7:44	PTH 文件	8,830 KB
ionization-1800-V2V.pth	2023/6/4 7:46	PTH 文件	8,830 KB
ionization-1810-V2V.pth	2023/6/4 7:48	PTH 文件	8,830 KB
ionization-1820-V2V.pth	2023/6/4 7:50	PTH 文件	8,830 KB
ionization-1830-V2V.pth	2023/6/4 7:52	PTH 文件	8,830 KB
ionization-1840-V2V.pth	2023/6/4 7:53	PTH 文件	8,830 KB
ionization-1850-V2V.pth	2023/6/4 7:55	PTH 文件	8,830 KB
ionization-1860-V2V.pth	2023/6/4 7:57	PTH 文件	8,830 KB
ionization-1870-V2V.pth	2023/6/4 7:59	PTH 文件	8,830 KB
ionization-1880-V2V.pth	2023/6/4 8:01	PTH 文件	8,830 KB
ionization-1890-V2V.pth	2023/6/4 8:02	PTH 文件	8,830 KB
ionization-1900-V2V.pth	2023/6/4 8:04	PTH 文件	8,830 KB
ionization-1910-V2V.pth	2023/6/4 8:06	PTH 文件	8,830 KB
ionization-1920-V2V.pth	2023/6/4 8:08	PTH 文件	8,830 KB
ionization-1930-V2V.pth	2023/6/4 8:10	PTH 文件	8,830 KB
ionization-1940-V2V.pth	2023/6/4 8:12	PTH 文件	8,830 KB
ionization-1950-V2V.pth	2023/6/4 8:13	PTH 文件	8,830 KB
ionization-1960-V2V.pth	2023/6/4 8:15	PTH 文件	8,830 KB
ionization-1970-V2V.pth	2023/6/4 8:17	PTH 文件	8,830 KB
ionization-1980-V2V.pth	2023/6/4 8:19	PTH 文件	8,830 KB
ionization-1990-V2V.pth	2023/6/4 8:21	PTH 文件	8,830 KB
ionization-2000-V2V.pth	2023/6/4 8:22	PTH 文件	8,830 KB

4.2.3 生成预测结果

生成预测结果的函数 inf()接受 args 和 dataset 作为参数，其中 args 是包含各种配置选项的对象，dataset 是数据集对象。

args 配置展示：

```
parser = argparse.ArgumentParser(description='PyTorch Implementation of V2V')
parser.add_argument('--lr_G', type=float, default=1e-4, metavar='LR',
                    help='学习率 of G')
parser.add_argument('--lr_D', type=float, default=4e-4, metavar='LR',
                    help='学习率 of D')
parser.add_argument('--no-cuda', action='store_true', default=False,
                    help='禁用CUDA训练')
parser.add_argument('--batch_size', type=int, default=1, metavar='N',
                    help='训练的输入批量大小')
parser.add_argument('--dataset', type=str, default = 'ionization',
                    help='数据集')
parser.add_argument('--mode', type=str, default='train' ,
                    help='训练或推理')
parser.add_argument('--epochs', type=int, default=200, metavar='N',
                    help='训练的轮数 (默认: 500)')
parser.add_argument('--croptimes', type=int, default=4, metavar='N',
                    help='每个数据的裁剪次数')
parser.add_argument('--crop', type=str, default='yes', metavar='N',
                    help='是否裁剪数据')

args = parser.parse_args()
args.cuda = not args.no_cuda and torch.cuda.is_available()
kwargs = {'num_workers': 30, 'pin_memory': True} if args.cuda else {}
args.mode = 'inf'
```

Inf()函数在每个循环中，首先读取一个原始数据文件（根据指定的命名规则），并进行预处理，将其转换为模型输入所需的形状和范围。然后将数据转换为 torch.FloatTensor 类型，并根据需要将其移动到 GPU 上。然后调用 concatsubvolume 函数对输入数据进行预测。预测结果保存在变量 t 中，并且对预测结果进行最小值和最大值的打印，并将结果保存到原始数据文件对应的结果文件中。最后，计算生成预测结果的平均时间，并将其打印出来。

Inf()函数就是使用预训练的模型对数据集中的数据进行推理，并生成对应的预测结果。

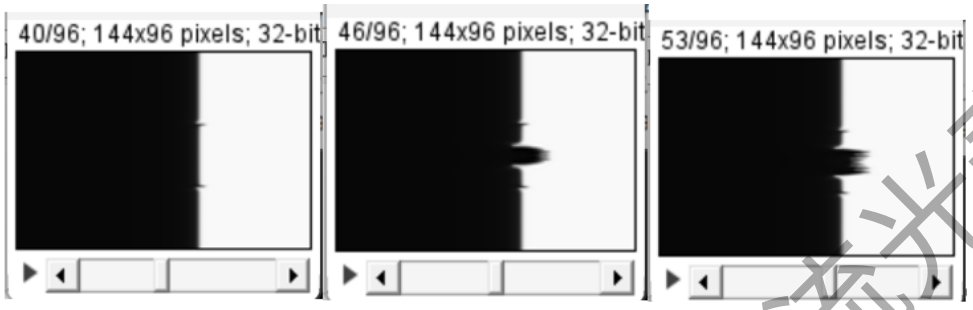
```
def inf(args, dataset):
    model = V2V()
    # 加载模型参数
    model.load_state_dict(torch.load('ionization_ab_H/Exp/' + args.dataset + '-' + str(args.epochs) + '-V2V.pth'))
    if args.cuda:
        model.cuda()
    x = time.time()
    for i in range(70, 100):
        print(i)
        v = np.zeros((1, 1, dataset.dim[0], dataset.dim[1], dataset.dim[2]))
        d = np.fromfile(dataset.t + '{:04d}'.format(i) + '.raw', dtype='<f')
        d = 2 * (d - np.min(d)) / (np.max(d) - np.min(d)) - 1
        d = d.reshape(dataset.dim[2], dataset.dim[1], dataset.dim[0]).transpose()
        v[0][0] = d
        v = torch.FloatTensor(v)
        if args.cuda:
            v = v.cuda()
        if args.dataset == 'ionization':
            t = concatsubvolume(model, v, [96, 96, 144], args)
        print(t.min())
        print(t.max())
        t = t.flatten('F')
        t = np.asarray(t, dtype='<f')
        t.tofile('ionization_ab_H/Result/' + args.dataset + '/' + '{:04d}'.format(i) + '.raw', format='<f')
    y = time.time()
    print((y - x) / dataset.total_samples)
```

5 结果可视化

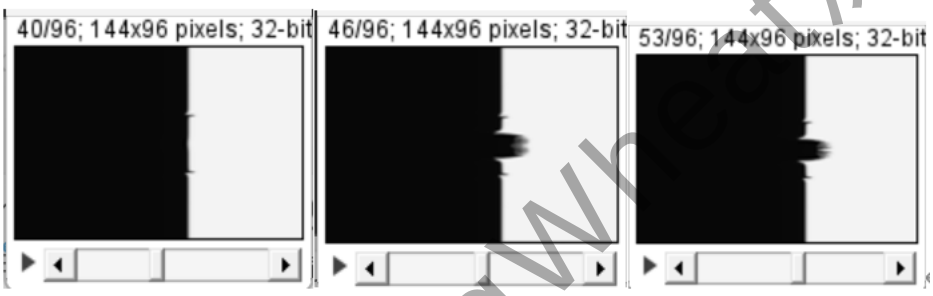
5.1 图像结果展示

GAN 网络生成的结果展示：
（展示的结果来自于上文所展示的数据样本 normInten_0005 与 normInten_0075）

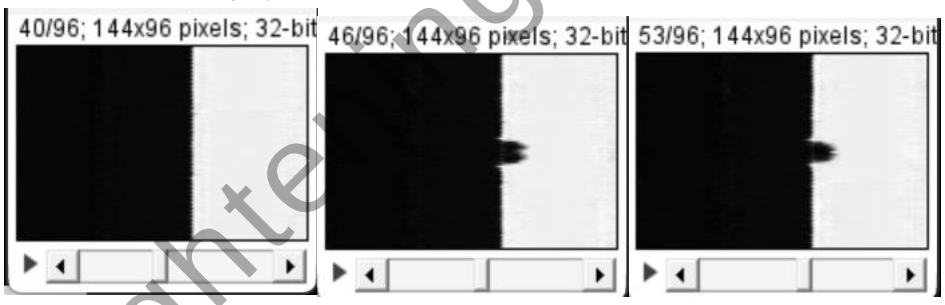
源数据

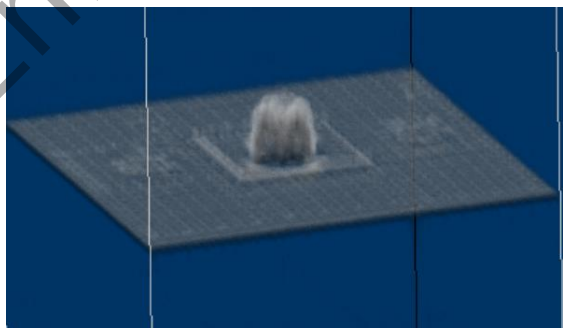


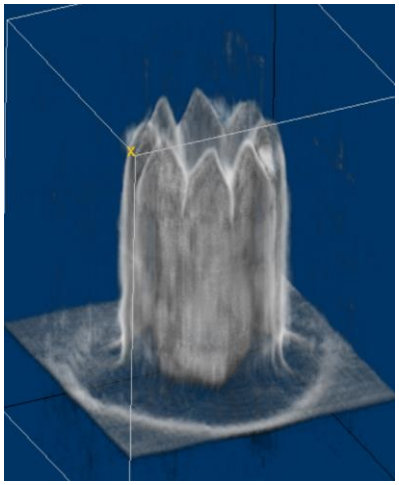
目标数据



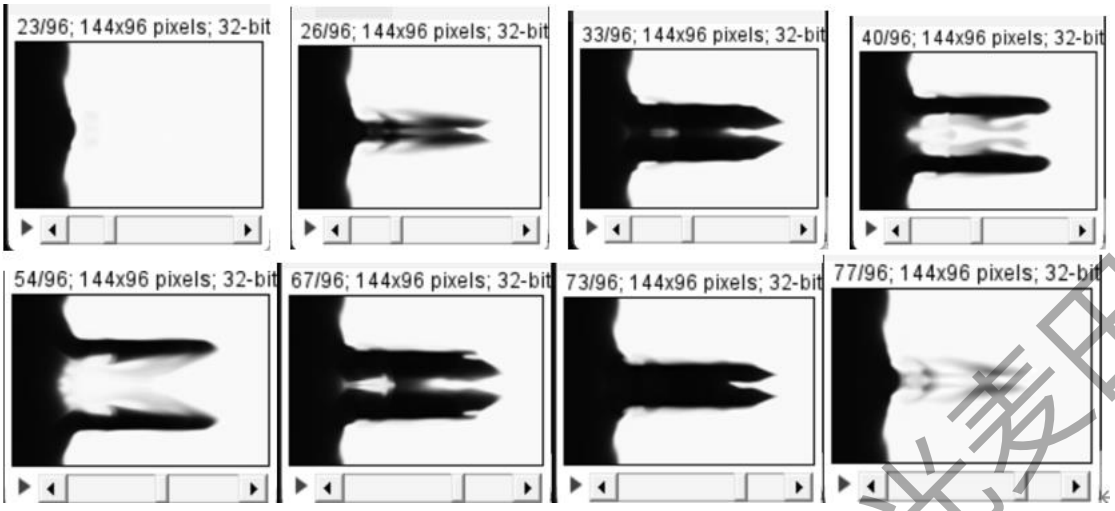
GAN 生成预测的数据



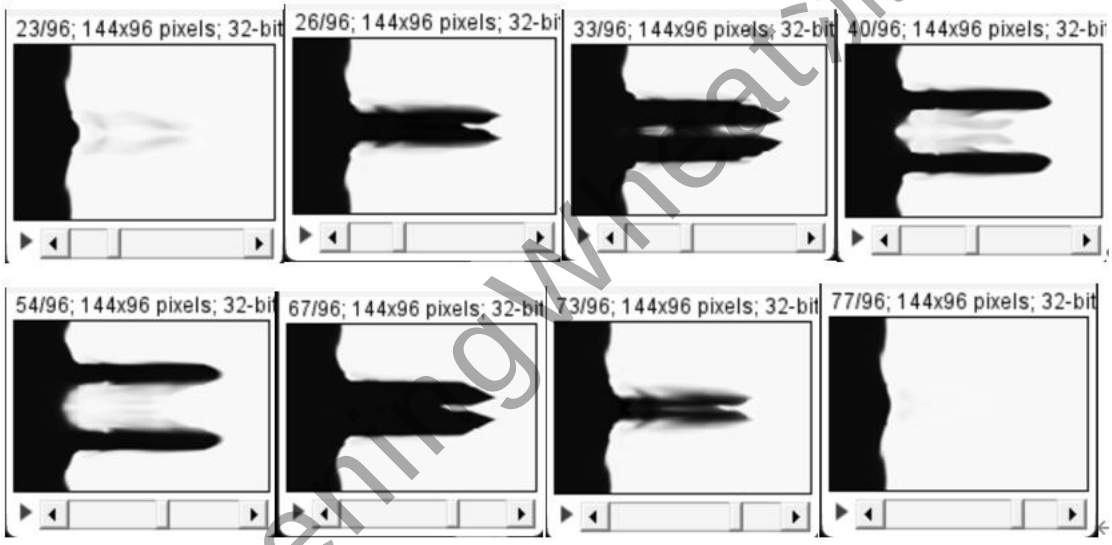




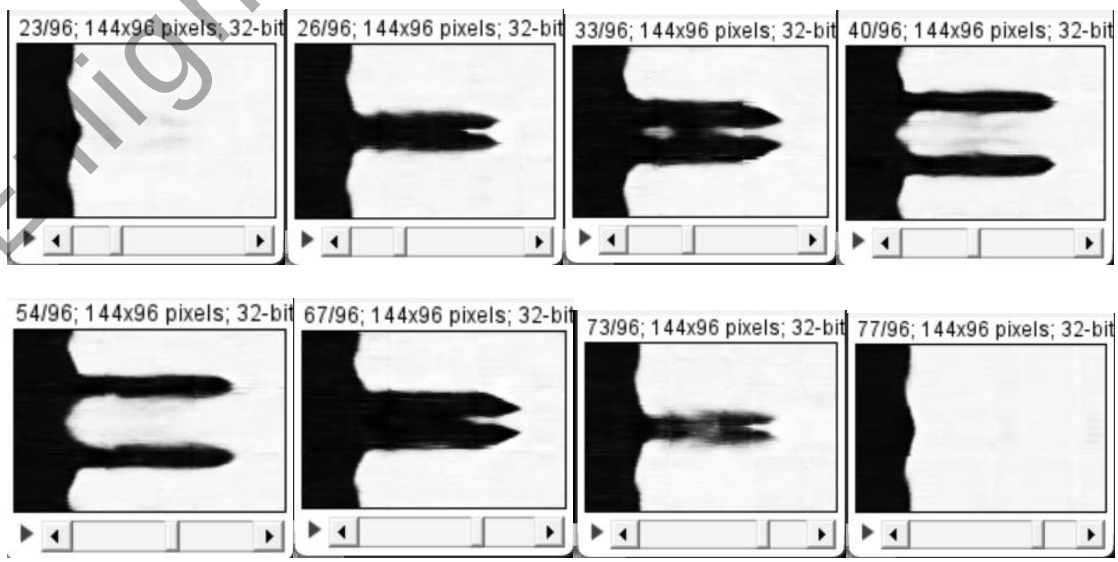
源数据



目标数据



GAN 网络生成预测的数据



从目标数据与 GAN 网络生成预测的数据对比来看，能够基本实现从源数据到目标数据比较成功的预测与转化。但图像的可视化上来看还是有一些噪点，尤其是图形边缘区域，这也是我们对于这个模型的一个改进方向。

5.2 结果与目标的损失函数结果展示

我们用均方误 nn.MSELoss()差来计算“以假乱真的 He 离子数据”和真正的 He 离子之间的差距来展示我们模型的成功度。

```
# 对抗性损失
L_adv = L2(output_real, label_real)

# 内容损失
L_c = L2(fake_data, o)

# 特征损失
L_p = L2(features_fake[0], features_real[0]) + L2(features_fake[1], features_real[1]) + L2(
    features_fake[2], features_real[2]) + L2(features_fake[3], features_real[3])

# 总损失
error = 1e-3 * L_adv + 1 * L_c + 1e-2 * L_p
error.backward()
loss_G += error.item()
```

loss-ionization.txt

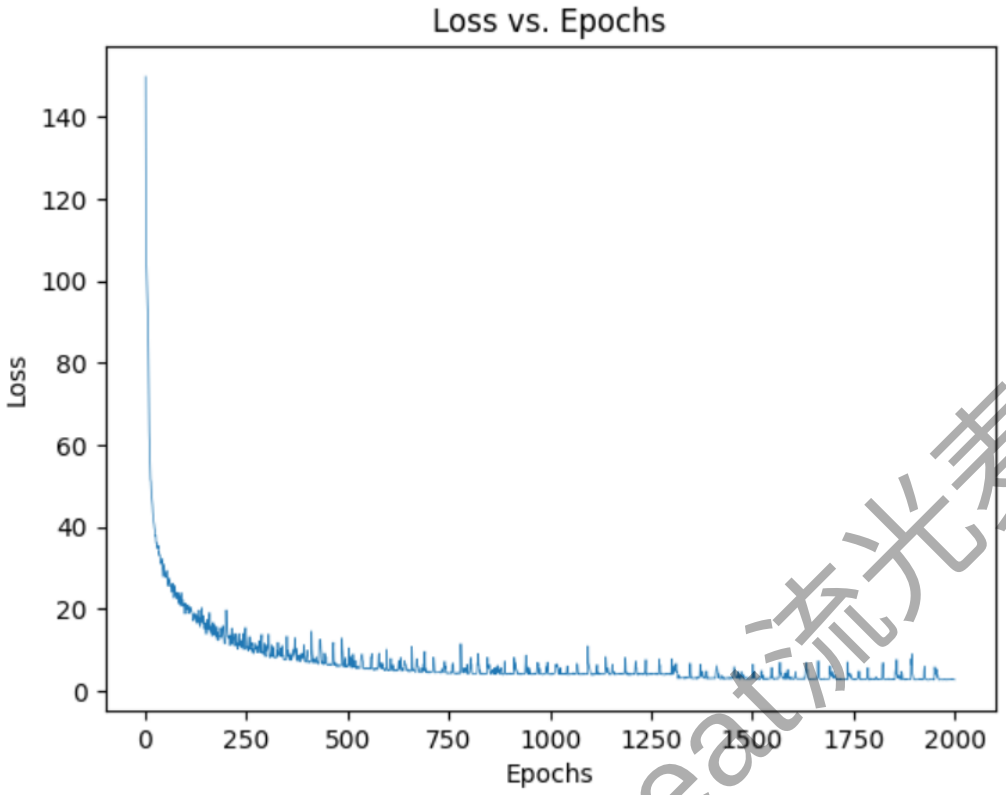
文件 编辑 查看

```
Epochs 1: Loss = 149.82808810472488
Epochs 2: Loss = 114.22741711139679
Epochs 3: Loss = 103.77709087729454
Epochs 4: Loss = 100.25977957248688
Epochs 5: Loss = 97.98170490562916
Epochs 6: Loss = 93.44821073114872
Epochs 7: Loss = 89.57245562970638
Epochs 8: Loss = 79.21506926417351
Epochs 9: Loss = 70.4526554942131
Epochs 10: Loss = 63.017879001796246
Epochs 11: Loss = 58.87763640284538
Epochs 12: Loss = 55.70134403556585
Epochs 13: Loss = 51.53265742957592
Epochs 14: Loss = 51.017416924238205
Epochs 15: Loss = 48.42874372005463
Epochs 16: Loss = 47.31970925629139
Epochs 17: Loss = 46.17509023472667
Epochs 18: Loss = 45.11173442006111
Epochs 19: Loss = 43.229060277342796
Epochs 20: Loss = 42.66483476012945
Epochs 21: Loss = 41.11137226969004
Epochs 22: Loss = 40.6103754453361
Epochs 23: Loss = 39.61195054277778
Epochs 24: Loss = 39.25359224528074
Epochs 25: Loss = 37.65728501975536
Epochs 26: Loss = 37.86501260101795
Epochs 27: Loss = 36.12169252336025
Epochs 28: Loss = 36.17090937867761
Epochs 29: Loss = 35.136870723217726
Epochs 30: Loss = 35.446044601500034
Epochs 31: Loss = 34.683724250644445
Epochs 32: Loss = 35.42062394693494
Epochs 33: Loss = 33.38917850330472
Epochs 34: Loss = 33.957357473671436
Epochs 35: Loss = 32.89243880286813
Epochs 36: Loss = 32.99185423180461
Epochs 37: Loss = 32.60490481555462
```

loss-ionization.txt

文件 编辑 查看

```
Epochs 1965: Loss = 2.702854255679995
Epochs 1966: Loss = 2.69419554900378
Epochs 1967: Loss = 2.7294763340614736
Epochs 1968: Loss = 2.7014635051600635
Epochs 1969: Loss = 2.7467417144216597
Epochs 1970: Loss = 2.6540559171698987
Epochs 1971: Loss = 2.653234998229891
Epochs 1972: Loss = 2.625910575967282
Epochs 1973: Loss = 2.721312173176557
Epochs 1974: Loss = 2.6606843965128064
Epochs 1975: Loss = 2.753160406369716
Epochs 1976: Loss = 2.7805228936485946
Epochs 1977: Loss = 2.65099911717698
Epochs 1978: Loss = 2.6999424416571856
Epochs 1979: Loss = 2.730066204443574
Epochs 1980: Loss = 2.635514197871089
Epochs 1981: Loss = 2.6857330021448433
Epochs 1982: Loss = 2.596905943006277
Epochs 1983: Loss = 2.6831691125407815
Epochs 1984: Loss = 2.742626511491835
Epochs 1985: Loss = 2.812495821621269
Epochs 1986: Loss = 2.7963029160164297
Epochs 1987: Loss = 2.7263859044760466
Epochs 1988: Loss = 2.734657464083284
Epochs 1989: Loss = 2.7046702560037374
Epochs 1990: Loss = 2.6587429894134402
Epochs 1991: Loss = 2.7822795007377863
Epochs 1992: Loss = 2.8651584880426526
Epochs 1993: Loss = 2.7179745151661336
Epochs 1994: Loss = 2.72005244391039
Epochs 1995: Loss = 2.7016179296188056
Epochs 1996: Loss = 2.708859160076827
Epochs 1997: Loss = 2.77196689741686
Epochs 1998: Loss = 2.8661272283643484
Epochs 1999: Loss = 2.783345422707498
Epochs 2000: Loss = 2.633971825707704
```



上图为 python 绘制的损失函数折线图。

显而易见，随着 epoch 迭代次数的增加，损失函数在稳步且明显的降低减少，但次数越多下降幅度越小。也正是因为多次的迭代，大大增加了模型生成“假数据”的仿真程度。

6 项目特色

在我们组基于 GAN 模型的课程设计上具有许多机器学习的特色，以下是本项目的特色分析：

（1） 模型架构创新

生成器模型采用了 **U-Net 结构**，包括编码器和解码器部分。编码器通过逐步降采样（downsampling）来提取图像特征，而解码器通过逐步上采样（upsampling）来生成最终的输出。这种结构可以帮助模型捕捉不同尺度的特征信息，特别适用于图像生成和分割任务。同时在 GAN 网络解码器部分，采用了残差连接（residual connection）的方式将编码器的特征图与解码器的特征图相结合。这种连接方式可以帮助信息传递和梯度流动，避免信息丢失和梯度消失问题。

（2） 数据预处理

代码中定义了 ReadData 方法和 TrainingData 方法来读取和处理数据。在读取数据时，它首先将原始数据进行归一化处理，并将其转换为指定的形状。通过设置 crop 参数为'yes'，可以进行数据裁剪，即随机从原始数据中截取指定大小的子图作为训练样本。这样做可以增加数据的多样性，提高模型的鲁棒性和泛化能力。在预处理中代码中使用了 PyTorch 的 torch.utils.data.TensorDataset 和 DataLoader 来构建数据集和数据加载器。

这些工具提供了高效的数据处理和批量加载功能，方便地将数据传入模型进行训练。

（3） 权重初始化和优化算法

使用 Adam 优化器来更新生成器和判别器的参数。Adam 优化器是一种常用的优化算法，结合了动量方法和自适应学习率的特点，适用于训练深度神经网络。通过设置不同的学习率和 beta 参数，可以影响优化器的收敛速度和稳定性，我们根据经验和实验设置参数，提高了训练效果。

使用 Kaiming 方法初始化权重：对于卷积层和线性层，代码使用 init.kaiming_uniform_方法以 Kaiming 均匀分布初始化权重。Kaiming 方法适用于使用 ReLU 激活函数的层，可以提高网络的收敛速度和表达能力。

（4） 损失函数和评估指标

在生成器的训练中，使用了多个损失函数来定义不同的训练目标。其中包括对抗性损失（L_adv），内容损失（L_c）和特征损失（L_p）。这些损失函数的综合考虑了生成器生成数据的真实性、内容一致性和特征匹配程度。

（5） 模型性能和效果

使用了 torch.FloatTensor 来将数据转换为 PyTorch 的张量格式，并可以将其移动到 GPU 上进行加速计算。这可以提高模型训练的速度和效率，特别是对于大规模数据和复杂模型。在训练中有着较大的 epoch 数量，有利于训练效率的提高和实验者发现训练效果与 epoch 之间的关系规律。

（6）实际应用和部署

本课程设计中的 GAN 神经网络能在一定程度上实现变量的预测与转化，根据特征提取实现变量到变量的转化，相信在不断改进之下可以进一步运用到科研的实践中去。

7 项目分工

黄启明（组长）：组织全组同学工作安排与几次集体合作，模型规划与分析，项目流程安排。
黄一玮：资料查找与文献分析，记录模型优化与改进。
林士皓、邹莲麦子：数据处理，模型代码实现与训练。

PS：本组成员中黄启明、林士皓、邹莲麦子同学有参与科研组，本课程设计与科研项目有部分重合，但都是本组成员在这个学期内独立完成的。

8 总结与感悟

本课程设计利用生成对抗网络模型，结合数据处理、模型训练、模型保存加载和推理过程等功能，实现了对逼真数据的生成和预测。

在这个课程设计中我们组成员花费了很多时间，查阅了很多资料，约在一起组会讨论了多次，结合了在本学期的 Python 程序设计课程中的学习以及在科研组的经验和积累，最终十分不容易得实现了这个生成对抗网络模型。当然，我们也从这个课程设计中收获了很多，虽然没有将 python 程序设计课程上的知识全部用上，但我们比较深刻的领悟到了机器学习的思想，以及感悟特别深的是见识到了 PyTorch 在机器学习上的功能强大与使用方便。我们的这个模型当

然还有很多不足之处，比如图像边缘处理模糊、大块像素相近的地方噪点较多以及还没有检验模型的拟合程度等等。但总体上来说，这是对我们收获很大的一次课程设计。