```
#Code originally on kaggle
!pip install -q POT
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Subset, TensorDataset,
Dataset
import torchvision.datasets as datasets
import torchvision.transforms as transforms
import torch.nn.functional as F
import numpy as np
import h5py
import matplotlib.pyplot as plt
import ot
from tqdm import tqdm
import random

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
pin_memory = True if device.type == 'cuda' else False

# Part 1: MNIST Conditional Variational Autoencoder (Digits 0 and 4)

transform = transforms.Compose([transforms.ToTensor()])
mnist_train = datasets.MNIST(root='./data', train=True, download=True,
transform=transform)
selected_digits = [0, 4]
indices = [i for i, (_, label) in enumerate(mnist_train) if label in
selected_digits]
mnist_subset = Subset(mnist_train, indices)
mnist_loader = DataLoader(mnist_subset, batch_size=128, shuffle=True,
num_workers=2, pin_memory=pin_memory)

class MNISTCVAE(nn.Module):
    def __init__(self, latent_dim=16, num_classes=2):
        super(MNISTCVAE, self).__init__()
        self.num_classes = num_classes
        self.encoder_conv = nn.Sequential(
            nn.Conv2d(1, 16, 3, 2, 1),
            nn.ReLU(),
            nn.Conv2d(16, 32, 3, 2, 1),
            nn.ReLU(),
            nn.Flatten()
        )
        self.feature_dim = 32 * 7 * 7
        self.fc_mu = nn.Linear(self.feature_dim + num_classes,
latent_dim)
        self.fc_logvar = nn.Linear(self.feature_dim + num_classes,
latent_dim)

        self.decoder_fc = nn.Sequential(
```

```
            nn.Linear(latent_dim + num_classes, 32 * 7 * 7),
            nn.ReLU()
        )
        self.decoder_conv = nn.Sequential(
            nn.ConvTranspose2d(32, 16, 3, 2, 1, output_padding=1),
            nn.ReLU(),
            nn.ConvTranspose2d(16, 1, 3, 2, 1, output_padding=1),
            nn.Sigmoid()
        )

    def reparameterize(self, mu, logvar, force_mean=False):
        if force_mean:
            return mu
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        return mu + eps * std

    def encode(self, x, y):
        x_feat = self.encoder_conv(x)
        x_feat_cond = torch.cat([x_feat, y], dim=1)
        mu = self.fc_mu(x_feat_cond)
        logvar = self.fc_logvar(x_feat_cond)
        return mu, logvar

    def decode(self, z, y):
        z_cond = torch.cat([z, y], dim=1)
        x_fc = self.decoder_fc(z_cond)
        x_fc = x_fc.view(z.size(0), 32, 7, 7)
        return self.decoder_conv(x_fc)

    def forward(self, x, y, force_mean=False):
        mu, logvar = self.encode(x, y)
        z = self.reparameterize(mu, logvar, force_mean=force_mean)
        recon = self.decode(z, y)
        return recon, mu, logvar

model_mnist = MNISTCVAE(latent_dim=16, num_classes=2).to(device)
optimizer_mnist = optim.Adam(model_mnist.parameters(), lr=1e-3)

def loss_function(recon, x, mu, logvar, kl_weight=0.001):
    BCE = F.binary_cross_entropy(recon, x, reduction='mean')
    KL = -0.5 * kl_weight * torch.mean(1 + logvar - mu.pow(2) -
logvar.exp())
    return BCE + KL

num_epochs = 20
print("Training MNIST CVAE...")
for epoch in range(num_epochs):
    model_mnist.train()
    running_loss = 0.0
```

```
    for images, labels in tqdm(mnist_loader, desc=f"Epoch
{epoch+1}/{num_epochs}"):
        images = images.to(device, non_blocking=True)

        condition = F.one_hot((labels == 4).long(),
num_classes=2).float().to(device)
        optimizer_mnist.zero_grad()
        recon, mu, logvar = model_mnist(images, condition)
        loss = loss_function(recon, images, mu, logvar)
        loss.backward()
        optimizer_mnist.step()
        running_loss += loss.item()
    epoch_loss = running_loss / len(mnist_loader)
    print(f"Epoch {epoch+1}/{num_epochs} Loss: {epoch_loss:.4f}")

def optimal_transport_mapping_conditioned(latent_batch, condition,
reg=0.0005, adjust_distribution=False):
    batch_size, latent_dim = latent_batch.shape
    transported = torch.zeros_like(latent_batch)
    classes = condition.argmax(dim=1)
    for class_idx in [0, 1]:
        idx = (classes == class_idx).nonzero(as_tuple=True)[0]
        if idx.numel() == 0:
            continue
        latent_group = latent_batch[idx]
        group_size = latent_group.size(0)
        shift = -5 if class_idx == 0 else 5
        target = torch.randn(group_size, latent_dim,
device=latent_batch.device) + shift
        x_norm = (latent_group ** 2).sum(dim=1, keepdim=True)
        y_norm = (target ** 2).sum(dim=1, keepdim=True).t()
        cost_matrix = x_norm + y_norm - 2 * torch.mm(latent_group,
target.t())
        cost_matrix_np = cost_matrix.detach().cpu().numpy()
        a = np.ones(group_size) / group_size
        b = np.ones(group_size) / group_size
        transport_plan = ot.sinkhorn(a, b, cost_matrix_np, reg)
        transported_np = np.dot(transport_plan,
target.detach().cpu().numpy())
        transported_group = torch.tensor(transported_np,
dtype=latent_batch.dtype, device=latent_batch.device)
        if adjust_distribution:
            eps = 1e-6
            mean_latent = latent_group.mean(dim=0, keepdim=True)
            std_latent = latent_group.std(dim=0, keepdim=True) + eps
            if (std_latent < 1e-9).all():
                transported_group = latent_group.clone()
            else:
                transported_group = (transported_group -
```

```
transported_group.mean(dim=0, keepdim=True)) /
(transported_group.std(dim=0, keepdim=True) + eps)
                transported_group = transported_group * std_latent +
mean_latent
        transported[idx] = transported_group
    return transported

def optimal_transport_mapping_general(latent_batch, reg=0.0005,
adjust_distribution=True, alpha=0.5):
    batch_size, latent_dim = latent_batch.shape
    target = torch.randn(batch_size, latent_dim,
device=latent_batch.device)
    x_norm = (latent_batch ** 2).sum(dim=1, keepdim=True)
    y_norm = (target ** 2).sum(dim=1, keepdim=True).t()
    cost_matrix = x_norm + y_norm - 2 * torch.mm(latent_batch,
target.t())
    cost_matrix_np = cost_matrix.detach().cpu().numpy()
    a = np.ones(batch_size) / batch_size
    b = np.ones(batch_size) / batch_size
    transport_plan = ot.sinkhorn(a, b, cost_matrix_np, reg)
    transported_np = np.dot(transport_plan,
target.detach().cpu().numpy())
    transported = torch.tensor(transported_np,
dtype=latent_batch.dtype, device=latent_batch.device)
    if adjust_distribution:
        eps = 1e-6
        mean_latent = latent_batch.mean(dim=0, keepdim=True)
        std_latent = latent_batch.std(dim=0, keepdim=True) + eps
        if (std_latent < 1e-9).all():
            transported = latent_batch.clone()
        else:
            transported = (transported - transported.mean(dim=0,
keepdim=True)) / (transported.std(dim=0, keepdim=True) + eps)
            transported = transported * std_latent + mean_latent
    transported = (1 - alpha) * latent_batch + alpha * transported
    return transported

model_mnist.eval()
with torch.no_grad():
    for images, labels in mnist_loader:
        images = images.to(device, non_blocking=True)
        condition = F.one_hot((labels == 4).long(),
num_classes=2).float().to(device)

        recon, _, _ = model_mnist(images, condition, force_mean=True)
        mu, logvar = model_mnist.encode(images, condition)
        transported_latent = optimal_transport_mapping_conditioned(mu,
condition, reg=0.0005, adjust_distribution=True)
        generated_ot = model_mnist.decode(transported_latent,
condition)
```

```
        break

# Map random Gaussian noise to latent space using a conditional latent
generator
latent_data = []
label_data = []
with torch.no_grad():
    for images, labels in mnist_loader:
        images = images.to(device, non_blocking=True)
        condition = F.one_hot((labels == 4).long(),
num_classes=2).float().to(device)
        mu, _ = model_mnist.encode(images, condition)
        latent_data.append(mu.cpu())
        label_data.append(labels.cpu())
latent_data = torch.cat(latent_data, dim=0)
label_data = torch.cat(label_data, dim=0)
labels_onehot = F.one_hot((label_data == 4).long(),
num_classes=2).float()
conditional_dataset = TensorDataset(latent_data, labels_onehot)
conditional_loader = DataLoader(conditional_dataset, batch_size=128,
shuffle=True)

class ConditionalLatentGenerator(nn.Module):
    def __init__(self, latent_dim=16, noise_dim=16, num_classes=2):
        super(ConditionalLatentGenerator, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(noise_dim + num_classes, 64),
            nn.ReLU(),
            nn.Linear(64, latent_dim)
        )
    def forward(self, noise, label):
        x = torch.cat([noise, label], dim=1)
        return self.net(x)

cond_latent_generator = ConditionalLatentGenerator(latent_dim=16,
noise_dim=16, num_classes=2).to(device)
optimizer_cond = optim.Adam(cond_latent_generator.parameters(), lr=1e-
3)
criterion_cond = nn.MSELoss()

num_epochs_cond = 5
for epoch in range(num_epochs_cond):
    cond_latent_generator.train()
    running_loss = 0.0
    total_samples = 0
    for latent_batch, label_batch in tqdm(conditional_loader,
desc=f"Cond Gen Epoch {epoch+1}/{num_epochs_cond}", leave=False):
        latent_batch = latent_batch.to(device, non_blocking=True)
        label_batch = label_batch.to(device, non_blocking=True)
        noise = torch.randn(latent_batch.size(0), 16).to(device)
```

```
        gen_latent = cond_latent_generator(noise, label_batch)
        loss = criterion_cond(gen_latent, latent_batch)
        optimizer_cond.zero_grad()
        loss.backward()
        optimizer_cond.step()
        running_loss += loss.item()
        total_samples += 1
    epoch_loss = running_loss / total_samples
    print(f"Cond Gen Epoch {epoch+1}/{num_epochs_cond} Loss:
{epoch_loss:.4f}")

cond_latent_generator.eval()
with torch.no_grad():
    mixed_batch = 16
    noise = torch.randn(mixed_batch, 16).to(device)
    labels_mixed = torch.zeros(mixed_batch, 2).to(device)
    half = mixed_batch // 2
    labels_mixed[:half, 0] = 1
    labels_mixed[half:, 1] = 1
    gen_latent_mixed = cond_latent_generator(noise, labels_mixed)
    generated_cond_mixed = model_mnist.decode(gen_latent_mixed,
labels_mixed)

def plot_mnist_random(images, title="Images", n=8):
    images = images.detach().cpu().numpy()
    idx = np.random.choice(images.shape[0], n, replace=False)
    selected = images[idx]
    fig, axes = plt.subplots(1, n, figsize=(n*2, 2))
    for i in range(n):
        axes[i].imshow(selected[i].squeeze(), cmap='gray')
        axes[i].axis('off')
    plt.suptitle(title)
    plt.show()

plot_mnist_random(images, title="MNIST: Original", n=8)
plot_mnist_random(recon, title="MNIST: Reconstructed ", n=8)
plot_mnist_random(generated_ot, title="MNIST: OT Generated", n=8)
plot_mnist_random(generated_cond_mixed, title="MNIST: Cond Generator
(Noise-Mapped)", n=8)



class HEPAutoencoderSubset(Dataset):
    def __init__(self, h5_path, num_samples=3000):
        with h5py.File(h5_path, 'r') as f:
            data = f['X_jets'][:num_samples]
        self.data = torch.tensor(data, dtype=torch.float32).permute(0,
3, 1, 2)
        for i in range(len(self.data)):
            for c in range(self.data.shape[1]):
```

```
                channel = self.data[i, c]
                min_val = channel.min()
                max_val = channel.max()
                if max_val > min_val:
                    self.data[i, c] = (channel - min_val) / (max_val - 
min_val)

    def __len__(self):
        return self.data.shape[0]

    def __getitem__(self, idx):
        return self.data[idx]

HEP_PATH = '/kaggle/input/quark-gluon/quark-gluon_data-
set_n139306.hdf5'
try:
    hep_dataset = HEPAutoencoderSubset(HEP_PATH, num_samples=3000)
except FileNotFoundError:
    print(f"File not found at {HEP_PATH}. Using dummy data for 
demonstration.")
    dummy_data = torch.rand(3000, 3, 125, 125)
    class DummyDataset(Dataset):
        def __init__(self, data):
            self.data = data
        def __len__(self):
            return len(self.data)
        def __getitem__(self, idx):
            return self.data[idx]
    hep_dataset = DummyDataset(dummy_data)

class HEPConvAutoencoder(nn.Module):
    def __init__(self, latent_dim=64):
        super(HEPConvAutoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Conv2d(3, 32, 3, 2, 1),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.Conv2d(32, 64, 3, 2, 1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.Conv2d(64, 128, 3, 2, 1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.Flatten(),
            nn.Linear(128 * 16 * 16, latent_dim),
            nn.BatchNorm1d(latent_dim)
        )

        self.decoder_fc = nn.Sequential(
```

```
            nn.Linear(latent_dim, 128 * 16 * 16),
            nn.ReLU()
        )

        self.decoder_conv = nn.Sequential(
            nn.ConvTranspose2d(128, 64, 3, 2, 1, output_padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.ConvTranspose2d(64, 32, 3, 2, 1, output_padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.ConvTranspose2d(32, 3, 3, 2, 1, output_padding=1),
            nn.Sigmoid()
        )

    def encode(self, x):
        return self.encoder(x)

    def decode(self, latent):
        x_fc = self.decoder_fc(latent)
        x_fc = x_fc.view(-1, 128, 16, 16)
        return self.decoder_conv(x_fc)[:, :, :125, :125]

    def forward(self, x):
        latent = self.encode(x)
        recon = self.decode(latent)
        return recon, latent

hep_model = HEPConvAutoencoder(latent_dim=64).to(device)
hep_optimizer = optim.Adam(hep_model.parameters(), lr=1e-3)
hep_criterion = nn.MSELoss(reduction='sum')
scaler = torch.cuda.amp.GradScaler() if device.type == 'cuda' else
None

from torch.utils.data import RandomSampler
num_batches_per_epoch = 1500
batch_size_hep = 32
num_samples_for_sampler = num_batches_per_epoch * batch_size_hep
sampler = RandomSampler(hep_dataset, replacement=True,
num_samples=num_samples_for_sampler)
hep_loader = DataLoader(
    hep_dataset,
    batch_size=batch_size_hep,
    sampler=sampler,
    num_workers=2,
    pin_memory=pin_memory
)

num_epochs_hep = 20
MAX_BATCHES_PER_EPOCH = 1500
```

```
for epoch in range(num_epochs_hep):
    hep_model.train()
    running_loss = 0.0
    total_samples = 0
    for i, images in enumerate(tqdm(hep_loader, desc=f"HEP Epoch
{epoch+1}/{num_epochs_hep}", leave=False)):
        if i >= MAX_BATCHES_PER_EPOCH:
            break
        images = images.to(device, non_blocking=True)
        hep_optimizer.zero_grad()
        if scaler:
            with torch.amp.autocast(device_type='cuda'):
                recon, latent = hep_model(images)
                loss = hep_criterion(recon, images)
            scaler.scale(loss).backward()
            scaler.step(hep_optimizer)
            scaler.update()
        else:
            recon, latent = hep_model(images)
            loss = hep_criterion(recon, images)
            loss.backward()
            hep_optimizer.step()
        batch_size = images.size(0)
        running_loss += loss.item() * batch_size
        total_samples += batch_size
    epoch_loss = running_loss / total_samples
    print(f"HEP Epoch {epoch+1}/{num_epochs_hep} Loss:
{epoch_loss:.4f}")

hep_model.eval()
with torch.no_grad():
    for images in hep_loader:
        images = images.to(device, non_blocking=True)
        recon, latent = hep_model(images)
        pure_ot_latent = optimal_transport_mapping_general(latent,
reg=0.01, adjust_distribution=True, alpha=1.0)
        pure_ot_generated = hep_model.decode(pure_ot_latent)
        blended_ot_latent = optimal_transport_mapping_general(latent,
reg=0.01, adjust_distribution=True, alpha=0.5)
        blended_ot_generated = hep_model.decode(blended_ot_latent)
        random_latent = torch.randn_like(latent).to(device)
        random_latent = random_latent * latent.std(dim=0,
keepdim=True) + latent.mean(dim=0, keepdim=True)
        random_generated = hep_model.decode(random_latent)
        break

def plot_channels_scaled(image, title="Scaled Image Channels"):
    channels = image.detach().cpu().numpy()
    fig, axes = plt.subplots(1, 3, figsize=(12, 4))
```

```
    for i in range(3):
        c = channels[i]
        c = np.log1p(c * 100) / np.log1p(100)
        axes[i].imshow(c, cmap='viridis', vmin=0, vmax=1)
        axes[i].axis('off')
        axes[i].set_title(f'Channel {i}')
    plt.suptitle(title)
    plt.tight_layout()
    plt.show()

rand_idx = random.randint(0, images.size(0)-1)
plot_channels_scaled(images[rand_idx], title="HEP: Original Image")
plot_channels_scaled(recon[rand_idx], title="HEP: Reconstructed
Image")
plot_channels_scaled(pure_ot_generated[rand_idx], title="HEP: Pure OT
Generated Image")
plot_channels_scaled(blended_ot_generated[rand_idx], title="HEP:
Blended OT Generated Image")
plot_channels_scaled(random_generated[rand_idx], title="HEP: Noise-
Generated Image")

class HEPLatentGenerator(nn.Module):
    def __init__(self, noise_dim=32, latent_dim=64):
        super(HEPLatentGenerator, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(noise_dim, 128),
            nn.LeakyReLU(0.2),
            nn.Linear(128, 256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, latent_dim),
            nn.Tanh()
        )
    def forward(self, noise):
        return self.net(noise)

latent_gen = HEPLatentGenerator(noise_dim=32,
latent_dim=64).to(device)
latent_gen_optimizer = optim.Adam(latent_gen.parameters(), lr=1e-4)
latent_gen_criterion = nn.MSELoss()

hep_latents = []
hep_model.eval()
with torch.no_grad():
    for i, images in enumerate(hep_loader):
        if i >= 100:
            break
        images = images.to(device, non_blocking=True)
        _, latent = hep_model(images)
        hep_latents.append(latent.cpu())
hep_latents = torch.cat(hep_latents, dim=0)
```

```
latent_dataset = TensorDataset(hep_latents)
latent_loader = DataLoader(latent_dataset, batch_size=64,
shuffle=True)

num_epochs_latent = 10
for epoch in range(num_epochs_latent):
    latent_gen.train()
    running_loss = 0.0
    total_samples = 0
    for i, (latent_batch,) in enumerate(tqdm(latent_loader,
desc=f"Latent Gen Epoch {epoch+1}/{num_epochs_latent}", leave=False)):
        latent_batch = latent_batch.to(device, non_blocking=True)
        noise = torch.randn(latent_batch.size(0), 32).to(device)
        latent_gen_optimizer.zero_grad()
        gen_latent = latent_gen(noise)
        loss = latent_gen_criterion(gen_latent, latent_batch)
        loss.backward()
        latent_gen_optimizer.step()
        running_loss += loss.item() * latent_batch.size(0)
        total_samples += latent_batch.size(0)
    epoch_loss = running_loss / total_samples
    print(f"Latent Gen Epoch {epoch+1}/{num_epochs_latent} Loss:
{epoch_loss:.4f}")

latent_gen.eval()
hep_model.eval()
with torch.no_grad():
    noise = torch.randn(16, 32).to(device)
    gen_latent = latent_gen(noise)
    noise_generated = hep_model.decode(gen_latent)

    rand_idx = random.randint(0, noise_generated.size(0)-1)
    plot_channels_scaled(noise_generated[rand_idx], title="HEP: Noise-
Generated Image")
```

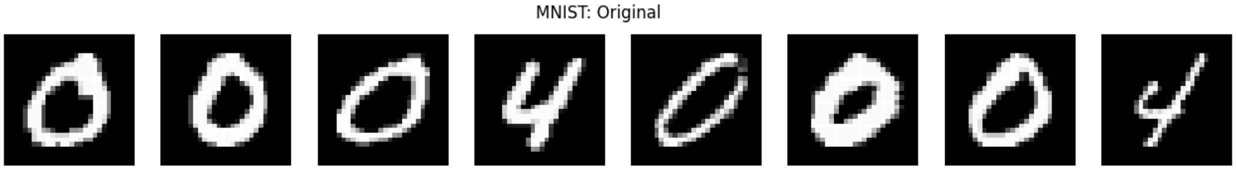
```
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 865.6/865.6 kB 15.2 MB/s eta
0:00:00a 0:00:01
/exdb/mnist/train-images-idx3-ubyte.gz
Failed to download (trying next):
HTTP Error 404: Not Found

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-
images-idx3-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-
images-idx3-ubyte.gz to ./data/MNIST/raw/train-images-idx3-ubyte.gz

100%|██████████| 9.91M/9.91M [00:00<00:00, 57.9MB/s]

Extracting ./data/MNIST/raw/train-images-idx3-ubyte.gz to
./data/MNIST/raw
```

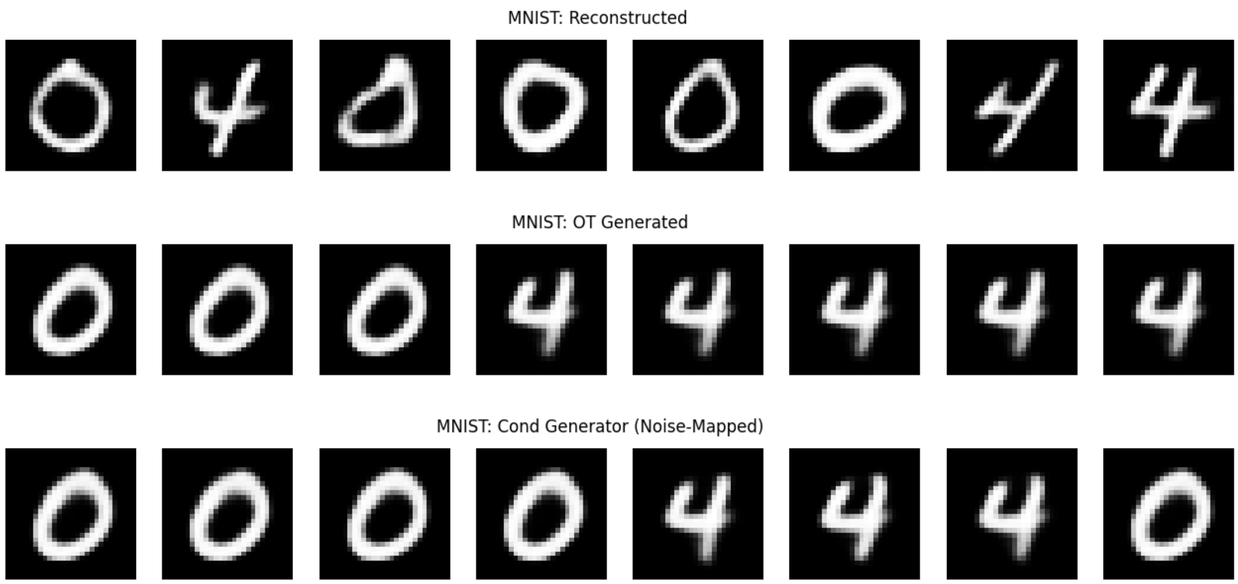
```
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-
ubyte.gz
Failed to download (trying next):
HTTP Error 404: Not Found

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-
labels-idx1-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-
labels-idx1-ubyte.gz to ./data/MNIST/raw/train-labels-idx1-ubyte.gz

100%|██████████| 28.9k/28.9k [00:00<00:00, 1.67MB/s]

Extracting ./data/MNIST/raw/train-labels-idx1-ubyte.gz to
./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Failed to download (trying next):
HTTP Error 404: Not Found

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-
idx3-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-
idx3-ubyte.gz to ./data/MNIST/raw/t10k-images-idx3-ubyte.gz

100%|██████████| 1.65M/1.65M [00:00<00:00, 14.7MB/s]

Extracting ./data/MNIST/raw/t10k-images-idx3-ubyte.gz to
./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Failed to download (trying next):
HTTP Error 404: Not Found

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-
idx1-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-
idx1-ubyte.gz to ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz

100%|██████████| 4.54k/4.54k [00:00<00:00, 13.7MB/s]

Extracting ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz to
./data/MNIST/raw

Training MNIST CVAE...

Epoch 1/20: 100%|██████████| 92/92 [00:02<00:00, 42.23it/s]

Epoch 1/20 Loss: 0.3804

Epoch 2/20: 100%|██████████| 92/92 [00:01<00:00, 85.17it/s]
```

```
Epoch 2/20 Loss: 0.2513
Epoch 3/20: 100%|██████████| 92/92 [00:01<00:00, 82.96it/s]
Epoch 3/20 Loss: 0.1985
Epoch 4/20: 100%|██████████| 92/92 [00:01<00:00, 84.07it/s]
Epoch 4/20 Loss: 0.1521
Epoch 5/20: 100%|██████████| 92/92 [00:01<00:00, 83.90it/s]
Epoch 5/20 Loss: 0.1301
Epoch 6/20: 100%|██████████| 92/92 [00:01<00:00, 84.37it/s]
Epoch 6/20 Loss: 0.1200
Epoch 7/20: 100%|██████████| 92/92 [00:01<00:00, 84.29it/s]
Epoch 7/20 Loss: 0.1144
Epoch 8/20: 100%|██████████| 92/92 [00:01<00:00, 83.58it/s]
Epoch 8/20 Loss: 0.1108
Epoch 9/20: 100%|██████████| 92/92 [00:01<00:00, 85.74it/s]
Epoch 9/20 Loss: 0.1082
Epoch 10/20: 100%|██████████| 92/92 [00:01<00:00, 83.77it/s]
Epoch 10/20 Loss: 0.1063
Epoch 11/20: 100%|██████████| 92/92 [00:01<00:00, 82.96it/s]
Epoch 11/20 Loss: 0.1048
Epoch 12/20: 100%|██████████| 92/92 [00:01<00:00, 84.65it/s]
Epoch 12/20 Loss: 0.1036
Epoch 13/20: 100%|██████████| 92/92 [00:01<00:00, 85.42it/s]
Epoch 13/20 Loss: 0.1025
Epoch 14/20: 100%|██████████| 92/92 [00:01<00:00, 79.79it/s]
Epoch 14/20 Loss: 0.1017
Epoch 15/20: 100%|██████████| 92/92 [00:01<00:00, 83.77it/s]
Epoch 15/20 Loss: 0.1009
Epoch 16/20: 100%|██████████| 92/92 [00:01<00:00, 82.31it/s]
```

```
Epoch 16/20 Loss: 0.1002

Epoch 17/20: 100%|██████████| 92/92 [00:01<00:00, 87.64it/s]

Epoch 17/20 Loss: 0.0995

Epoch 18/20: 100%|██████████| 92/92 [00:01<00:00, 86.98it/s]

Epoch 18/20 Loss: 0.0990

Epoch 19/20: 100%|██████████| 92/92 [00:01<00:00, 87.52it/s]

Epoch 19/20 Loss: 0.0985

Epoch 20/20: 100%|██████████| 92/92 [00:01<00:00, 83.14it/s]

Epoch 20/20 Loss: 0.0981


/usr/local/lib/python3.10/dist-packages/ot/bregman/_sinkhorn.py:631:
RuntimeWarning: divide by zero encountered in divide
  v = b / KtransposeU
/usr/local/lib/python3.10/dist-packages/ot/bregman/_sinkhorn.py:643:
UserWarning: Warning: numerical errors at iteration 0
  warnings.warn("Warning: numerical errors at iteration %d" % ii)


Cond Gen Epoch 1/5 Loss: 1.8184



Cond Gen Epoch 2/5 Loss: 1.5720



Cond Gen Epoch 3/5 Loss: 1.4755



Cond Gen Epoch 4/5 Loss: 1.4669



Cond Gen Epoch 5/5 Loss: 1.4640
```

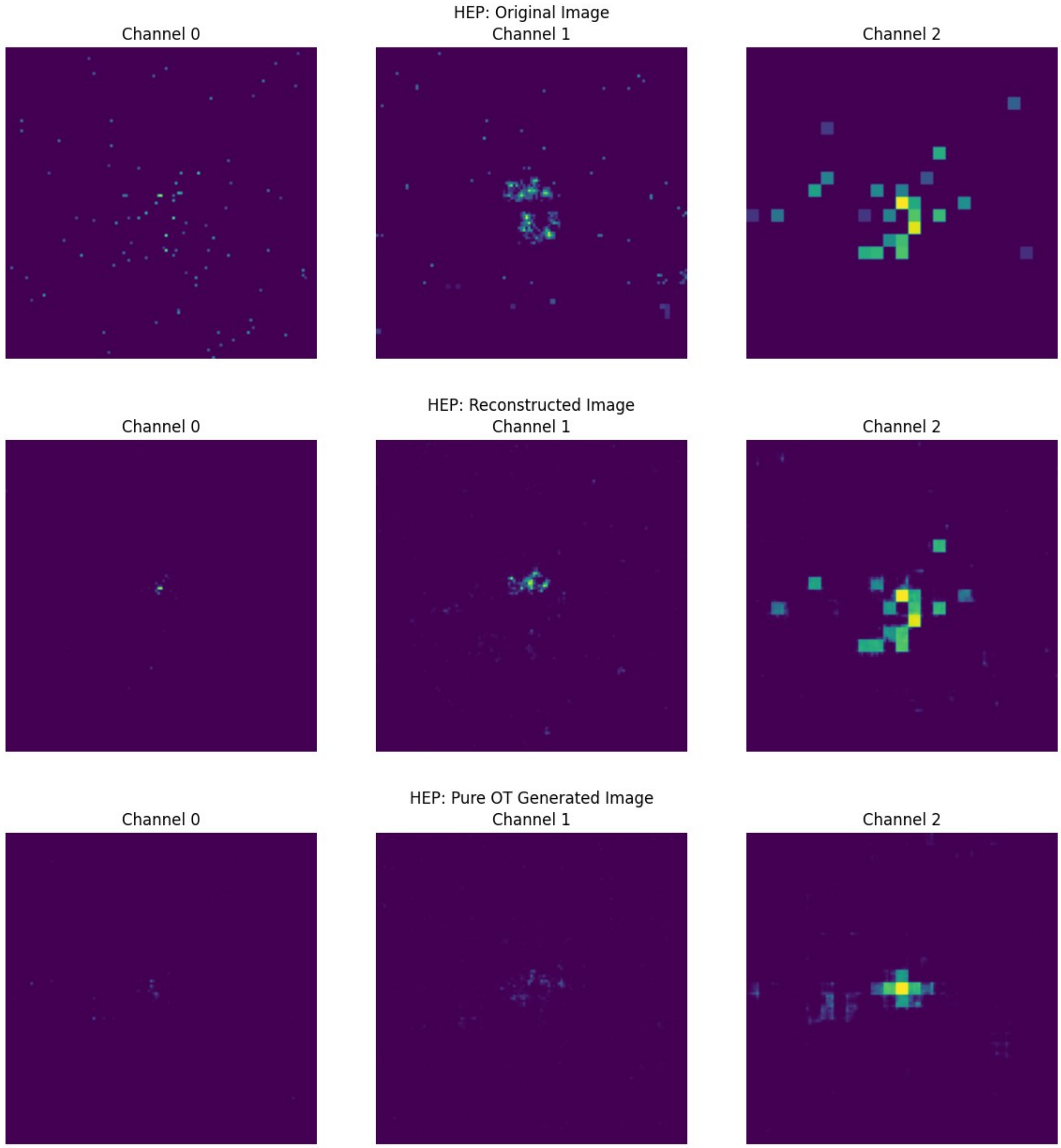
MNIST: Original

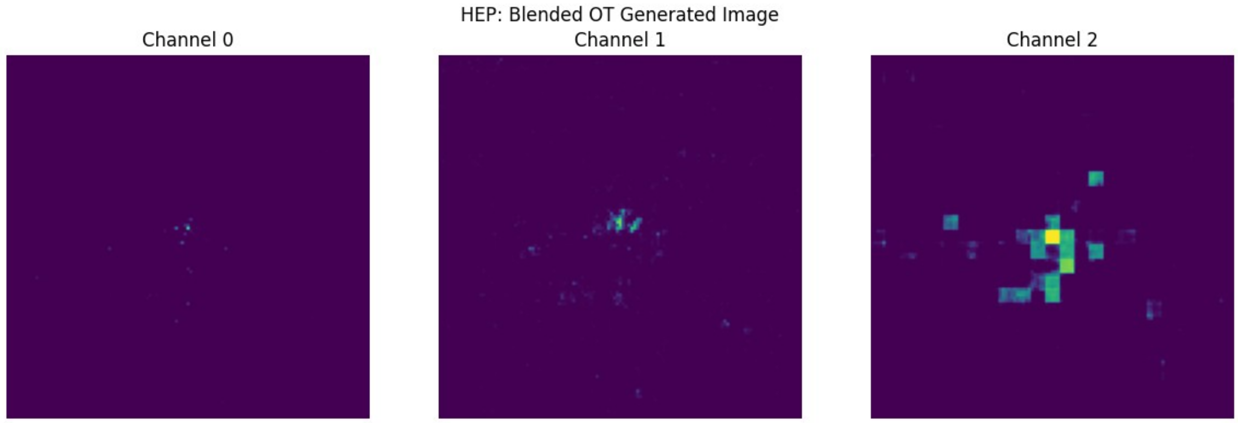


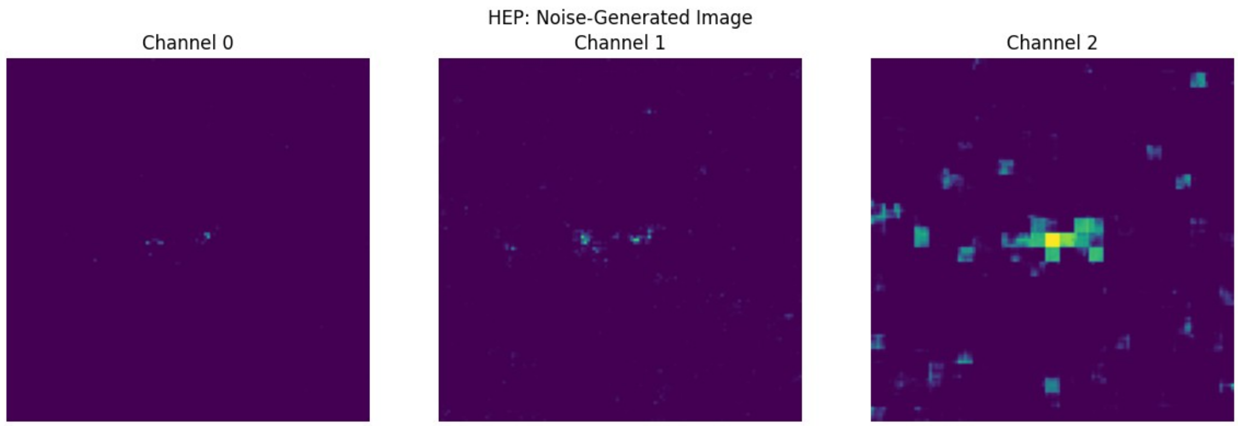



```
<ipython-input-1-fe9d46db2f98>:339: FutureWarning:
`torch.cuda.amp.GradScaler(args...)` is deprecated. Please use
`torch.amp.GradScaler('cuda', args...)` instead.
  scaler = torch.cuda.amp.GradScaler() if device.type == 'cuda' else
None


HEP Epoch 1/20 Loss: 13059.3396



HEP Epoch 2/20 Loss: 342.7961



HEP Epoch 3/20 Loss: 242.7392



HEP Epoch 4/20 Loss: 209.3069



HEP Epoch 5/20 Loss: 194.3286



HEP Epoch 6/20 Loss: 183.1492



HEP Epoch 7/20 Loss: 175.1301
```

```
HEP Epoch 8/20 Loss: 165.5424

HEP Epoch 9/20 Loss: 155.5915

HEP Epoch 10/20 Loss: 145.2767

HEP Epoch 11/20 Loss: 135.6450

HEP Epoch 12/20 Loss: 127.5305

HEP Epoch 13/20 Loss: 120.5782

HEP Epoch 14/20 Loss: 113.9880

HEP Epoch 15/20 Loss: 108.7008

HEP Epoch 16/20 Loss: 104.6929

HEP Epoch 17/20 Loss: 101.7539

HEP Epoch 18/20 Loss: 99.6971

HEP Epoch 19/20 Loss: 96.8082

HEP Epoch 20/20 Loss: 96.2557
```


HEP: Original Image
Channel 0
Channel 1
Channel 2
HEP: Reconstructed Image
Channel 0
Channel 1
Channel 2
HEP: Pure OT Generated Image
Channel 0
Channel 1
Channel 2






```
Latent Gen Epoch 1/10 Loss: 0.7731

Latent Gen Epoch 2/10 Loss: 0.7272

Latent Gen Epoch 3/10 Loss: 0.7118

Latent Gen Epoch 4/10 Loss: 0.7070

Latent Gen Epoch 5/10 Loss: 0.7057

Latent Gen Epoch 6/10 Loss: 0.7050
```

```
Latent Gen Epoch 7/10 Loss: 0.7043

Latent Gen Epoch 8/10 Loss: 0.7039

Latent Gen Epoch 9/10 Loss: 0.7038

Latent Gen Epoch 10/10 Loss: 0.7037
```