In [23]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
# Acknowledgement to
# https://github.com/kuangliu/pytorch-cifar,
# https://github.com/BIGBALLON/CIFAR-ZOO,


''' Swish activation '''
class Swish(nn.Module): # Swish(x) = x*σ(x)
    def __init__(self):
        super().__init__()

    def forward(self, input):
        return input * torch.sigmoid(input)


''' MLP '''
class MLP(nn.Module):
    def __init__(self, channel, num_classes):
        super(MLP, self).__init__()
        self.fc_1 = nn.Linear(28*28*1 if channel==1 else 32*32*3, 128)
        self.fc_2 = nn.Linear(128, 128)
        self.fc_3 = nn.Linear(128, num_classes)

    def forward(self, x):
        out = x.view(x.size(0), -1)
        out = F.relu(self.fc_1(out))
        out = F.relu(self.fc_2(out))
        out = self.fc_3(out)
        return out



''' ConvNet '''
class ConvNet(nn.Module):
    def __init__(self, channel, num_classes, net_width, net_depth, net_act, net_nor
        super(ConvNet, self).__init__()

        self.features, shape_feat = self._make_layers(channel, net_width, net_depth
        num_feat = shape_feat[0]*shape_feat[1]*shape_feat[2]
        self.classifier = nn.Linear(num_feat, num_classes)

    def forward(self, x):
        out = self.features(x)
        out = out.view(out.size(0), -1)
        out = self.classifier(out)
        return out

    def embed(self, x):
        out = self.features(x)
        out = out.view(out.size(0), -1)
        return out
```

```
    def _get_activation(self, net_act):
        if net_act == 'sigmoid':
            return nn.Sigmoid()
        elif net_act == 'relu':
            return nn.ReLU(inplace=True)
        elif net_act == 'leakyrelu':
            return nn.LeakyReLU(negative_slope=0.01)
        elif net_act == 'swish':
            return Swish()
        else:
            exit('unknown activation function: %s'%net_act)

    def _get_pooling(self, net_pooling):
        if net_pooling == 'maxpooling':
            return nn.MaxPool2d(kernel_size=2, stride=2)
        elif net_pooling == 'avgpooling':
            return nn.AvgPool2d(kernel_size=2, stride=2)
        elif net_pooling == 'none':
            return None
        else:
            exit('unknown net_pooling: %s'%net_pooling)

    def _get_normlayer(self, net_norm, shape_feat):
        # shape_feat = (c*h*w)
        if net_norm == 'batchnorm':
            return nn.BatchNorm2d(shape_feat[0], affine=True)
        elif net_norm == 'layernorm':
            return nn.LayerNorm(shape_feat, elementwise_affine=True)
        elif net_norm == 'instancenorm':
            return nn.GroupNorm(shape_feat[0], shape_feat[0], affine=True)
        elif net_norm == 'groupnorm':
            return nn.GroupNorm(4, shape_feat[0], affine=True)
        elif net_norm == 'none':
            return None
        else:
            exit('unknown net_norm: %s'%net_norm)

    def _make_layers(self, channel, net_width, net_depth, net_norm, net_act, net_po
        layers = []
        in_channels = channel
        if im_size[0] == 28:
            im_size = (32, 32)
        shape_feat = [in_channels, im_size[0], im_size[1]]
        for d in range(net_depth):
            layers += [nn.Conv2d(in_channels, net_width, kernel_size=3, padding=3 i
            shape_feat[0] = net_width
            if net_norm != 'none':
                layers += [self._get_normlayer(net_norm, shape_feat)]
            layers += [self._get_activation(net_act)]
            in_channels = net_width
            if net_pooling != 'none':
                layers += [self._get_pooling(net_pooling)]
                shape_feat[1] //= 2
                shape_feat[2] //= 2

        return nn.Sequential(*layers), shape_feat
```

```
''' LeNet '''
class LeNet(nn.Module):
    def __init__(self, channel, num_classes):
        super(LeNet, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(channel, 6, kernel_size=5, padding=2 if channel==1 else 0),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(6, 16, kernel_size=5),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
        self.fc_1 = nn.Linear(16 * 5 * 5, 120)
        self.fc_2 = nn.Linear(120, 84)
        self.fc_3 = nn.Linear(84, num_classes)

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc_1(x))
        x = F.relu(self.fc_2(x))
        x = self.fc_3(x)
        return x


''' AlexNet '''
class AlexNet(nn.Module):
    def __init__(self, channel, num_classes):
        super(AlexNet, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(channel, 128, kernel_size=5, stride=1, padding=4 if channel==
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(128, 192, kernel_size=5, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(192, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 192, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(192, 192, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
        self.fc = nn.Linear(192 * 4 * 4, num_classes)

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x
```

```
    def embed(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)
        return x


''' AlexNetBN '''
class AlexNetBN(nn.Module):
    def __init__(self, channel, num_classes):
        super(AlexNetBN, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(channel, 128, kernel_size=5, stride=1, padding=4 if channel==
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(128, 192, kernel_size=5, padding=2),
            nn.BatchNorm2d(192),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(192, 256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 192, kernel_size=3, padding=1),
            nn.BatchNorm2d(192),
            nn.ReLU(inplace=True),
            nn.Conv2d(192, 192, kernel_size=3, padding=1),
            nn.BatchNorm2d(192),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
        self.fc = nn.Linear(192 * 4 * 4, num_classes)

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x

    def embed(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)
        return x


''' VGG '''
cfg_vgg = {
    'VGG11': [64, 'M', 128, 'M', 256, 256, 'M', 512, 512, 'M', 512, 512, 'M'],
    'VGG13': [64, 64, 'M', 128, 128, 'M', 256, 256, 'M', 512, 512, 'M', 512, 512, '
    'VGG16': [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 'M', 512, 512, 512, 'M', 5
    'VGG19': [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 256, 'M', 512, 512, 512, 5
}
class VGG(nn.Module):
    def __init__(self, vgg_name, channel, num_classes, norm='instancenorm'):
        super(VGG, self).__init__()
        self.channel = channel
        self.features = self._make_layers(cfg_vgg[vgg_name], norm)
```

```
        self.classifier = nn.Linear(512 if vgg_name != 'VGGS' else 128, num_classes

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)
        x = self.classifier(x)
        return x

    def embed(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)
        return x

    def _make_layers(self, cfg, norm):
        layers = []
        in_channels = self.channel
        for ic, x in enumerate(cfg):
            if x == 'M':
                layers += [nn.MaxPool2d(kernel_size=2, stride=2)]
            else:
                layers += [nn.Conv2d(in_channels, x, kernel_size=3, padding=3 if se
                           nn.GroupNorm(x, x, affine=True) if norm=='instancenorm'
                           nn.ReLU(inplace=True)]
                in_channels = x
        layers += [nn.AvgPool2d(kernel_size=1, stride=1)]
        return nn.Sequential(*layers)


def VGG11(channel, num_classes):
    return VGG('VGG11', channel, num_classes)
def VGG11BN(channel, num_classes):
    return VGG('VGG11', channel, num_classes, norm='batchnorm')
def VGG13(channel, num_classes):
    return VGG('VGG13', channel, num_classes)
def VGG16(channel, num_classes):
    return VGG('VGG16', channel, num_classes)
def VGG19(channel, num_classes):
    return VGG('VGG19', channel, num_classes)


''' ResNet_AP '''
# The conv(stride=2) is replaced by conv(stride=1) + avgpool(kernel_size=2, stride=

class BasicBlock_AP(nn.Module):
    expansion = 1

    def __init__(self, in_planes, planes, stride=1, norm='instancenorm'):
        super(BasicBlock_AP, self).__init__()
        self.norm = norm
        self.stride = stride
        self.conv1 = nn.Conv2d(in_planes, planes, kernel_size=3, stride=1, padding=
        self.bn1 = nn.GroupNorm(planes, planes, affine=True) if self.norm == 'insta
        self.conv2 = nn.Conv2d(planes, planes, kernel_size=3, stride=1, padding=1,
        self.bn2 = nn.GroupNorm(planes, planes, affine=True) if self.norm == 'insta

        self.shortcut = nn.Sequential()
```

```
        if stride != 1 or in_planes != self.expansion * planes:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_planes, self.expansion * planes, kernel_size=1, stride
                nn.AvgPool2d(kernel_size=2, stride=2), # modification
                nn.GroupNorm(self.expansion * planes, self.expansion * planes, affi
            )

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        if self.stride != 1: # modification
            out = F.avg_pool2d(out, kernel_size=2, stride=2)
        out = self.bn2(self.conv2(out))
        out += self.shortcut(x)
        out = F.relu(out)
        return out


class Bottleneck_AP(nn.Module):
    expansion = 4

    def __init__(self, in_planes, planes, stride=1, norm='instancenorm'):
        super(Bottleneck_AP, self).__init__()
        self.norm = norm
        self.stride = stride
        self.conv1 = nn.Conv2d(in_planes, planes, kernel_size=1, bias=False)
        self.bn1 = nn.GroupNorm(planes, planes, affine=True) if self.norm == 'insta
        self.conv2 = nn.Conv2d(planes, planes, kernel_size=3, stride=1, padding=1,
        self.bn2 = nn.GroupNorm(planes, planes, affine=True) if self.norm == 'insta
        self.conv3 = nn.Conv2d(planes, self.expansion * planes, kernel_size=1, bias
        self.bn3 = nn.GroupNorm(self.expansion * planes, self.expansion * planes, a

        self.shortcut = nn.Sequential()
        if stride != 1 or in_planes != self.expansion * planes:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_planes, self.expansion * planes, kernel_size=1, stride
                nn.AvgPool2d(kernel_size=2, stride=2),  # modification
                nn.GroupNorm(self.expansion * planes, self.expansion * planes, affi
            )

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = F.relu(self.bn2(self.conv2(out)))
        if self.stride != 1: # modification
            out = F.avg_pool2d(out, kernel_size=2, stride=2)
        out = self.bn3(self.conv3(out))
        out += self.shortcut(x)
        out = F.relu(out)
        return out


class ResNet_AP(nn.Module):
    def __init__(self, block, num_blocks, channel=3, num_classes=10, norm='instance
        super(ResNet_AP, self).__init__()
        self.in_planes = 64
        self.norm = norm
```

```
        self.conv1 = nn.Conv2d(channel, 64, kernel_size=3, stride=1, padding=1, bia
        self.bn1 = nn.GroupNorm(64, 64, affine=True) if self.norm == 'instancenorm'
        self.layer1 = self._make_layer(block, 64, num_blocks[0], stride=1)
        self.layer2 = self._make_layer(block, 128, num_blocks[1], stride=2)
        self.layer3 = self._make_layer(block, 256, num_blocks[2], stride=2)
        self.layer4 = self._make_layer(block, 512, num_blocks[3], stride=2)
        self.classifier = nn.Linear(512 * block.expansion * 3 * 3 if channel==1 els

    def _make_layer(self, block, planes, num_blocks, stride):
        strides = [stride] + [1] * (num_blocks - 1)
        layers = []
        for stride in strides:
            layers.append(block(self.in_planes, planes, stride, self.norm))
            self.in_planes = planes * block.expansion
        return nn.Sequential(*layers)

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = F.avg_pool2d(out, kernel_size=1, stride=1) # modification
        out = out.view(out.size(0), -1)
        out = self.classifier(out)
        return out

    def embed(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = F.avg_pool2d(out, kernel_size=1, stride=1) # modification
        out = out.view(out.size(0), -1)
        return out

def ResNet18BN_AP(channel, num_classes):
    return ResNet_AP(BasicBlock_AP, [2,2,2,2], channel=channel, num_classes=num_cla

def ResNet18_AP(channel, num_classes):
    return ResNet_AP(BasicBlock_AP, [2,2,2,2], channel=channel, num_classes=num_cla


''' ResNet '''

class BasicBlock(nn.Module):
    expansion = 1

    def __init__(self, in_planes, planes, stride=1, norm='instancenorm'):
        super(BasicBlock, self).__init__()
        self.norm = norm
        self.conv1 = nn.Conv2d(in_planes, planes, kernel_size=3, stride=stride, pad
        self.bn1 = nn.GroupNorm(planes, planes, affine=True) if self.norm == 'insta
        self.conv2 = nn.Conv2d(planes, planes, kernel_size=3, stride=1, padding=1,
        self.bn2 = nn.GroupNorm(planes, planes, affine=True) if self.norm == 'insta
```

```
        self.shortcut = nn.Sequential()
        if stride != 1 or in_planes != self.expansion*planes:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_planes, self.expansion*planes, kernel_size=1, stride=s
                nn.GroupNorm(self.expansion*planes, self.expansion*planes, affine=T
            )

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += self.shortcut(x)
        out = F.relu(out)
        return out


class Bottleneck(nn.Module):
    expansion = 4

    def __init__(self, in_planes, planes, stride=1, norm='instancenorm'):
        super(Bottleneck, self).__init__()
        self.norm = norm
        self.conv1 = nn.Conv2d(in_planes, planes, kernel_size=1, bias=False)
        self.bn1 = nn.GroupNorm(planes, planes, affine=True) if self.norm == 'insta
        self.conv2 = nn.Conv2d(planes, planes, kernel_size=3, stride=stride, paddin
        self.bn2 = nn.GroupNorm(planes, planes, affine=True) if self.norm == 'insta
        self.conv3 = nn.Conv2d(planes, self.expansion*planes, kernel_size=1, bias=F
        self.bn3 = nn.GroupNorm(self.expansion*planes, self.expansion*planes, affin

        self.shortcut = nn.Sequential()
        if stride != 1 or in_planes != self.expansion*planes:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_planes, self.expansion*planes, kernel_size=1, stride=s
                nn.GroupNorm(self.expansion*planes, self.expansion*planes, affine=T
            )

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = F.relu(self.bn2(self.conv2(out)))
        out = self.bn3(self.conv3(out))
        out += self.shortcut(x)
        out = F.relu(out)
        return out


class ResNet(nn.Module):
    def __init__(self, block, num_blocks, channel=3, num_classes=10, norm='instance
        super(ResNet, self).__init__()
        self.in_planes = 64
        self.norm = norm

        self.conv1 = nn.Conv2d(channel, 64, kernel_size=3, stride=1, padding=1, bia
        self.bn1 = nn.GroupNorm(64, 64, affine=True) if self.norm == 'instancenorm'
        self.layer1 = self._make_layer(block, 64, num_blocks[0], stride=1)
        self.layer2 = self._make_layer(block, 128, num_blocks[1], stride=2)
        self.layer3 = self._make_layer(block, 256, num_blocks[2], stride=2)
```

```
        self.layer4 = self._make_layer(block, 512, num_blocks[3], stride=2)
        self.classifier = nn.Linear(512*block.expansion, num_classes)

    def _make_layer(self, block, planes, num_blocks, stride):
        strides = [stride] + [1]*(num_blocks-1)
        layers = []
        for stride in strides:
            layers.append(block(self.in_planes, planes, stride, self.norm))
            self.in_planes = planes * block.expansion
        return nn.Sequential(*layers)

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = F.avg_pool2d(out, 4)
        out = out.view(out.size(0), -1)
        out = self.classifier(out)
        return out

    def embed(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = F.avg_pool2d(out, 4)
        out = out.view(out.size(0), -1)
        return out


def ResNet18BN(channel, num_classes):
    return ResNet(BasicBlock, [2,2,2,2], channel=channel, num_classes=num_classes,

def ResNet18(channel, num_classes):
    return ResNet(BasicBlock, [2,2,2,2], channel=channel, num_classes=num_classes)

def ResNet34(channel, num_classes):
    return ResNet(BasicBlock, [3,4,6,3], channel=channel, num_classes=num_classes)

def ResNet50(channel, num_classes):
    return ResNet(Bottleneck, [3,4,6,3], channel=channel, num_classes=num_classes)

def ResNet101(channel, num_classes):
    return ResNet(Bottleneck, [3,4,23,3], channel=channel, num_classes=num_classes)

def ResNet152(channel, num_classes):
    return ResNet(Bottleneck, [3,8,36,3], channel=channel, num_classes=num_classes)
```

In [24]:

```
import time
import os
import numpy as np
import torch
import torch.nn as nn
```

```
import torch.nn.functional as F
from torch.utils.data import Dataset
from torchvision import datasets, transforms
from scipy.ndimage.interpolation import rotate as scipyrotate
#from networks import MLP, ConvNet, LeNet, AlexNet, AlexNetBN, VGG11, VGG11BN, ResN

def get_dataset(dataset, data_path):
    if dataset == 'MNIST':
        channel = 1
        im_size = (28, 28)
        num_classes = 10
        mean = [0.1307]
        std = [0.3081]
        transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize
        dst_train = datasets.MNIST(data_path, train=True, download=True, transform=
        dst_test = datasets.MNIST(data_path, train=False, download=True, transform=
        class_names = [str(c) for c in range(num_classes)]

    elif dataset == 'FashionMNIST':
        channel = 1
        im_size = (28, 28)
        num_classes = 10
        mean = [0.2861]
        std = [0.3530]
        transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize
        dst_train = datasets.FashionMNIST(data_path, train=True, download=True, tra
        dst_test = datasets.FashionMNIST(data_path, train=False, download=True, tra
        class_names = dst_train.classes

    elif dataset == 'SVHN':
        channel = 3
        im_size = (32, 32)
        num_classes = 10
        mean = [0.4377, 0.4438, 0.4728]
        std = [0.1980, 0.2010, 0.1970]
        transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize
        dst_train = datasets.SVHN(data_path, split='train', download=True, transfor
        dst_test = datasets.SVHN(data_path, split='test', download=True, transform=
        class_names = [str(c) for c in range(num_classes)]

    elif dataset == 'CIFAR10':
        channel = 3
        im_size = (32, 32)
        num_classes = 10
        mean = [0.4914, 0.4822, 0.4465]
        std = [0.2023, 0.1994, 0.2010]
        transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize
        dst_train = datasets.CIFAR10(data_path, train=True, download=True, transfor
        dst_test = datasets.CIFAR10(data_path, train=False, download=True, transfor
        class_names = dst_train.classes

    elif dataset == 'CIFAR100':
        channel = 3
        im_size = (32, 32)
        num_classes = 100
        mean = [0.5071, 0.4866, 0.4409]
```

```
        std = [0.2673, 0.2564, 0.2762]
        transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize
        dst_train = datasets.CIFAR100(data_path, train=True, download=True, transfo
        dst_test = datasets.CIFAR100(data_path, train=False, download=True, transfo
        class_names = dst_train.classes

    elif dataset == 'TinyImageNet':
        channel = 3
        im_size = (64, 64)
        num_classes = 200
        mean = [0.485, 0.456, 0.406]
        std = [0.229, 0.224, 0.225]
        data = torch.load(os.path.join(data_path, 'tinyimagenet.pt'), map_location=

        class_names = data['classes']

        images_train = data['images_train']
        labels_train = data['labels_train']
        images_train = images_train.detach().float() / 255.0
        labels_train = labels_train.detach()
        for c in range(channel):
            images_train[:,c] = (images_train[:,c] - mean[c])/std[c]
        dst_train = TensorDataset(images_train, labels_train)  # no augmentation

        images_val = data['images_val']
        labels_val = data['labels_val']
        images_val = images_val.detach().float() / 255.0
        labels_val = labels_val.detach()

        for c in range(channel):
            images_val[:, c] = (images_val[:, c] - mean[c]) / std[c]

        dst_test = TensorDataset(images_val, labels_val)  # no augmentation

    else:
        exit('unknown dataset: %s'%dataset)


    testloader = torch.utils.data.DataLoader(dst_test, batch_size=256, shuffle=Fals
    return channel, im_size, num_classes, class_names, mean, std, dst_train, dst_te



class TensorDataset(Dataset):
    def __init__(self, images, labels): # images: n x c x h x w tensor
        self.images = images.detach().float()
        self.labels = labels.detach()

    def __getitem__(self, index):
        return self.images[index], self.labels[index]

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

```
def get_default_convnet_setting():
    net_width, net_depth, net_act, net_norm, net_pooling = 128, 3, 'relu', 'instanc
    return net_width, net_depth, net_act, net_norm, net_pooling



def get_network(model, channel, num_classes, im_size=(32, 32)):
    torch.random.manual_seed(int(time.time() * 1000) % 100000)
    net_width, net_depth, net_act, net_norm, net_pooling = get_default_convnet_sett

    if model == 'MLP':
        net = MLP(channel=channel, num_classes=num_classes)
    elif model == 'ConvNet':
        net = ConvNet(channel=channel, num_classes=num_classes, net_width=net_width
    elif model == 'LeNet':
        net = LeNet(channel=channel, num_classes=num_classes)
    elif model == 'AlexNet':
        net = AlexNet(channel=channel, num_classes=num_classes)
    elif model == 'AlexNetBN':
        net = AlexNetBN(channel=channel, num_classes=num_classes)
    elif model == 'VGG11':
        net = VGG11( channel=channel, num_classes=num_classes)
    elif model == 'VGG11BN':
        net = VGG11BN(channel=channel, num_classes=num_classes)
    elif model == 'ResNet18':
        net = ResNet18(channel=channel, num_classes=num_classes)
    elif model == 'ResNet18BN_AP':
        net = ResNet18BN_AP(channel=channel, num_classes=num_classes)
    elif model == 'ResNet18BN':
        net = ResNet18BN(channel=channel, num_classes=num_classes)

    elif model == 'ConvNetD1':
        net = ConvNet(channel=channel, num_classes=num_classes, net_width=net_width
    elif model == 'ConvNetD2':
        net = ConvNet(channel=channel, num_classes=num_classes, net_width=net_width
    elif model == 'ConvNetD3':
        net = ConvNet(channel=channel, num_classes=num_classes, net_width=net_width
    elif model == 'ConvNetD4':
        net = ConvNet(channel=channel, num_classes=num_classes, net_width=net_width
    elif model == 'ConvNetD7':
        net = ConvNet(channel=channel, num_classes=num_classes, net_width=net_width

    elif model == 'ConvNetW32':
        net = ConvNet(channel=channel, num_classes=num_classes, net_width=32, net_d
    elif model == 'ConvNetW64':
        net = ConvNet(channel=channel, num_classes=num_classes, net_width=64, net_d
    elif model == 'ConvNetW128':
        net = ConvNet(channel=channel, num_classes=num_classes, net_width=128, net_
    elif model == 'ConvNetW256':
        net = ConvNet(channel=channel, num_classes=num_classes, net_width=256, net_

    elif model == 'ConvNetAS':
        net = ConvNet(channel=channel, num_classes=num_classes, net_width=net_width
    elif model == 'ConvNetAR':
        net = ConvNet(channel=channel, num_classes=num_classes, net_width=net_width
    elif model == 'ConvNetAL':
```

```
        net = ConvNet(channel=channel, num_classes=num_classes, net_width=net_width
    elif model == 'ConvNetASwish':
        net = ConvNet(channel=channel, num_classes=num_classes, net_width=net_width
    elif model == 'ConvNetASwishBN':
        net = ConvNet(channel=channel, num_classes=num_classes, net_width=net_width

    elif model == 'ConvNetNN':
        net = ConvNet(channel=channel, num_classes=num_classes, net_width=net_width
    elif model == 'ConvNetBN':
        net = ConvNet(channel=channel, num_classes=num_classes, net_width=net_width
    elif model == 'ConvNetLN':
        net = ConvNet(channel=channel, num_classes=num_classes, net_width=net_width
    elif model == 'ConvNetIN':
        net = ConvNet(channel=channel, num_classes=num_classes, net_width=net_width
    elif model == 'ConvNetGN':
        net = ConvNet(channel=channel, num_classes=num_classes, net_width=net_width

    elif model == 'ConvNetNP':
        net = ConvNet(channel=channel, num_classes=num_classes, net_width=net_width
    elif model == 'ConvNetMP':
        net = ConvNet(channel=channel, num_classes=num_classes, net_width=net_width
    elif model == 'ConvNetAP':
        net = ConvNet(channel=channel, num_classes=num_classes, net_width=net_width

    else:
        net = None
        exit('unknown model: %s'%model)

    gpu_num = torch.cuda.device_count()
    if gpu_num>0:
        device = 'cuda'
        if gpu_num>1:
            net = nn.DataParallel(net)
    else:
        device = 'cpu'
    net = net.to(device)

    return net



def get_time():
    return str(time.strftime("[%Y-%m-%d %H:%M:%S]", time.localtime()))



def distance_wb(gwr, gws):
    shape = gwr.shape
    if len(shape) == 4: # conv, out*in*h*w
        gwr = gwr.reshape(shape[0], shape[1] * shape[2] * shape[3])
        gws = gws.reshape(shape[0], shape[1] * shape[2] * shape[3])
    elif len(shape) == 3:  # layernorm, C*h*w
        gwr = gwr.reshape(shape[0], shape[1] * shape[2])
        gws = gws.reshape(shape[0], shape[1] * shape[2])
    elif len(shape) == 2: # linear, out*in
        tmp = 'do nothing'
```

```
    elif len(shape) == 1: # batchnorm/instancenorm, C; groupnorm x, bias
        gwr = gwr.reshape(1, shape[0])
        gws = gws.reshape(1, shape[0])
        return torch.tensor(0, dtype=torch.float, device=gwr.device)

    dis_weight = torch.sum(1 - torch.sum(gwr * gws, dim=-1) / (torch.norm(gwr, dim=
    dis = dis_weight
    return dis



def match_loss(gw_syn, gw_real, args):
    dis = torch.tensor(0.0).to(args.device)

    if args.dis_metric == 'ours':
        for ig in range(len(gw_real)):
            gwr = gw_real[ig]
            gws = gw_syn[ig]
            dis += distance_wb(gwr, gws)

    elif args.dis_metric == 'mse':
        gw_real_vec = []
        gw_syn_vec = []
        for ig in range(len(gw_real)):
            gw_real_vec.append(gw_real[ig].reshape((-1)))
            gw_syn_vec.append(gw_syn[ig].reshape((-1)))
        gw_real_vec = torch.cat(gw_real_vec, dim=0)
        gw_syn_vec = torch.cat(gw_syn_vec, dim=0)
        dis = torch.sum((gw_syn_vec - gw_real_vec)**2)

    elif args.dis_metric == 'cos':
        gw_real_vec = []
        gw_syn_vec = []
        for ig in range(len(gw_real)):
            gw_real_vec.append(gw_real[ig].reshape((-1)))
            gw_syn_vec.append(gw_syn[ig].reshape((-1)))
        gw_real_vec = torch.cat(gw_real_vec, dim=0)
        gw_syn_vec = torch.cat(gw_syn_vec, dim=0)
        dis = 1 - torch.sum(gw_real_vec * gw_syn_vec, dim=-1) / (torch.norm(gw_real

    else:
        exit('unknown distance function: %s'%args.dis_metric)

    return dis



def get_loops(ipc):
    # Get the two hyper-parameters of outer-loop and inner-loop.
    # The following values are empirically good.
    if ipc == 1:
        outer_loop, inner_loop = 1, 1
    elif ipc == 10:
        outer_loop, inner_loop = 10, 50
    elif ipc == 20:
        outer_loop, inner_loop = 20, 25
```

```
    elif ipc == 30:
        outer_loop, inner_loop = 30, 20
    elif ipc == 40:
        outer_loop, inner_loop = 40, 15
    elif ipc == 50:
        outer_loop, inner_loop = 50, 10
    else:
        outer_loop, inner_loop = 0, 0
        exit('loop hyper-parameters are not defined for %d ipc'%ipc)
    return outer_loop, inner_loop



def epoch(mode, dataloader, net, optimizer, criterion, args, aug):
    loss_avg, acc_avg, num_exp = 0, 0, 0
    net = net.to(args.device)
    criterion = criterion.to(args.device)

    if mode == 'train':
        net.train()
    else:
        net.eval()

    for i_batch, datum in enumerate(dataloader):

        img = datum[0].float().to(args.device)
        #print(datum, img)
        #datum[0] = torch.tensor(datum[0])
        if aug:
            if args.dsa:
                img = DiffAugment(img, args.dsa_strategy, param=args.dsa_param)
            else:
                img = augment(img, args.dc_aug_param, device=args.device)
        # Ensure `lab` is 1D by adding an extra dimension if it's a scalar
        lab = datum[1].to(args.device).long()
        if lab.ndim == 0:  # If lab is a scalar
            lab = lab.unsqueeze(0)
        #print(lab)
        #lab = lab.unsqueeze(0)
        if img.ndim == 3:  # If shape is [128, 32,S 32]
            img = img.unsqueeze(0)


        n_b = lab.shape[0]

        output = net(img)
        loss = criterion(output, lab)
        acc = np.sum(np.equal(np.argmax(output.cpu().data.numpy(), axis=-1), lab.cp

        loss_avg += loss.item()*n_b
        acc_avg += acc
        num_exp += n_b

        if mode == 'train':
            optimizer.zero_grad()
            loss.backward()
```

```
            optimizer.step()

    loss_avg /= num_exp
    acc_avg /= num_exp

    return loss_avg, acc_avg



def evaluate_synset(it_eval, net, images_train, labels_train, testloader, args):
    net = net.to(args.device)
    images_train = images_train.to(args.device)
    labels_train = labels_train.to(args.device)
    lr = float(args.lr_net)
    Epoch = int(args.epoch_eval_train)
    lr_schedule = [Epoch//2+1]
    optimizer = torch.optim.SGD(net.parameters(), lr=lr, momentum=0.9, weight_decay
    criterion = nn.CrossEntropyLoss().to(args.device)

    dst_train = TensorDataset(images_train, labels_train)
    trainloader = torch.utils.data.DataLoader(dst_train, batch_size=args.batch_trai

    start = time.time()
    for ep in range(Epoch+1):
        loss_train, acc_train = epoch('train', trainloader, net, optimizer, criteri
        if ep in lr_schedule:
            lr *= 0.1
            optimizer = torch.optim.SGD(net.parameters(), lr=lr, momentum=0.9, weig

    time_train = time.time() - start
    loss_test, acc_test = epoch('test', testloader, net, optimizer, criterion, args
    print('%s Evaluate_%02d: epoch = %04d train time = %d s train loss = %.6f train

    return net, acc_train, acc_test



def augment(images, dc_aug_param, device):
    # This can be sped up in the future.

    if dc_aug_param != None and dc_aug_param['strategy'] != 'none':
        scale = dc_aug_param['scale']
        crop = dc_aug_param['crop']
        rotate = dc_aug_param['rotate']
        noise = dc_aug_param['noise']
        strategy = dc_aug_param['strategy']

        shape = images.shape
        mean = []
        for c in range(shape[1]):
            mean.append(float(torch.mean(images[:,c])))

        def cropfun(i):
            im_ = torch.zeros(shape[1],shape[2]+crop*2,shape[3]+crop*2, dtype=torch
            for c in range(shape[1]):
                im_[c] = mean[c]
```

```
            im_[:, crop:crop+shape[2], crop:crop+shape[3]] = images[i]
            r, c = np.random.permutation(crop*2)[0], np.random.permutation(crop*2)[
            images[i] = im_[:, r:r+shape[2], c:c+shape[3]]

        def scalefun(i):
            h = int((np.random.uniform(1 - scale, 1 + scale)) * shape[2])
            w = int((np.random.uniform(1 - scale, 1 + scale)) * shape[2])
            tmp = F.interpolate(images[i:i + 1], [h, w], )[0]
            mhw = max(h, w, shape[2], shape[3])
            im_ = torch.zeros(shape[1], mhw, mhw, dtype=torch.float, device=device)
            r = int((mhw - h) / 2)
            c = int((mhw - w) / 2)
            im_[:, r:r + h, c:c + w] = tmp
            r = int((mhw - shape[2]) / 2)
            c = int((mhw - shape[3]) / 2)
            images[i] = im_[:, r:r + shape[2], c:c + shape[3]]

        def rotatefun(i):
            im_ = scipyrotate(images[i].cpu().data.numpy(), angle=np.random.randint
            r = int((im_.shape[-2] - shape[-2]) / 2)
            c = int((im_.shape[-1] - shape[-1]) / 2)
            images[i] = torch.tensor(im_[:, r:r + shape[-2], c:c + shape[-1]], dtyp

        def noisefun(i):
            images[i] = images[i] + noise * torch.randn(shape[1:], dtype=torch.floa


        augs = strategy.split('_')

        for i in range(shape[0]):
            choice = np.random.permutation(augs)[0] # randomly implement one augmen
            if choice == 'crop':
                cropfun(i)
            elif choice == 'scale':
                scalefun(i)
            elif choice == 'rotate':
                rotatefun(i)
            elif choice == 'noise':
                noisefun(i)

    return images



def get_daparam(dataset, model, model_eval, ipc):
    # We find that augmentation doesn't always benefit the performance.
    # So we do augmentation for some of the settings.

    dc_aug_param = dict()
    dc_aug_param['crop'] = 4
    dc_aug_param['scale'] = 0.2
    dc_aug_param['rotate'] = 45
    dc_aug_param['noise'] = 0.001
    dc_aug_param['strategy'] = 'none'

    if dataset == 'MNIST':
```

```
        dc_aug_param['strategy'] = 'crop_scale_rotate'

    if model_eval in ['ConvNetBN']: # Data augmentation makes model training with B
        dc_aug_param['strategy'] = 'crop_noise'

    return dc_aug_param


def get_eval_pool(eval_mode, model, model_eval):
    if eval_mode == 'M': # multiple architectures
        model_eval_pool = ['MLP', 'ConvNet', 'LeNet', 'AlexNet', 'VGG11', 'ResNet18
    elif eval_mode == 'B':  # multiple architectures with BatchNorm for DM experime
        model_eval_pool = ['ConvNetBN', 'ConvNetASwishBN', 'AlexNetBN', 'VGG11BN',
    elif eval_mode == 'W': # ablation study on network width
        model_eval_pool = ['ConvNetW32', 'ConvNetW64', 'ConvNetW128', 'ConvNetW256'
    elif eval_mode == 'D': # ablation study on network depth
        model_eval_pool = ['ConvNetD1', 'ConvNetD2', 'ConvNetD3', 'ConvNetD4']
    elif eval_mode == 'A': # ablation study on network activation function
        model_eval_pool = ['ConvNetAS', 'ConvNetAR', 'ConvNetAL', 'ConvNetASwish']
    elif eval_mode == 'P': # ablation study on network pooling layer
        model_eval_pool = ['ConvNetNP', 'ConvNetMP', 'ConvNetAP']
    elif eval_mode == 'N': # ablation study on network normalization layer
        model_eval_pool = ['ConvNetNN', 'ConvNetBN', 'ConvNetLN', 'ConvNetIN', 'Con
    elif eval_mode == 'S': # itself
        if 'BN' in model:
            print('Attention: Here I will replace BN with IN in evaluation, as the
        model_eval_pool = [model[:model.index('BN')]] if 'BN' in model else [model]
    elif eval_mode == 'SS':  # itself
        model_eval_pool = [model]
    else:
        model_eval_pool = [model_eval]
    return model_eval_pool


class ParamDiffAug():
    def __init__(self):
        self.aug_mode = 'S' #'multiple or single'
        self.prob_flip = 0.5
        self.ratio_scale = 1.2
        self.ratio_rotate = 15.0
        self.ratio_crop_pad = 0.125
        self.ratio_cutout = 0.5 # the size would be 0.5x0.5
        self.brightness = 1.0
        self.saturation = 2.0
        self.contrast = 0.5


def set_seed_DiffAug(param):
    if param.latestseed == -1:
        return
    else:
        torch.random.manual_seed(param.latestseed)
        param.latestseed += 1


def DiffAugment(x, strategy='', seed = -1, param = None):
```

```
    if strategy == 'None' or strategy == 'none' or strategy == '':
        return x

    if seed == -1:
        param.Siamese = False
    else:
        param.Siamese = True

    param.latestseed = seed

    if strategy:
        if param.aug_mode == 'M': # original
            for p in strategy.split('_'):
                for f in AUGMENT_FNS[p]:
                    x = f(x, param)
        elif param.aug_mode == 'S':
            pbties = strategy.split('_')
            set_seed_DiffAug(param)
            p = pbties[torch.randint(0, len(pbties), size=(1,)).item()]
            for f in AUGMENT_FNS[p]:
                x = f(x, param)
        else:
            exit('unknown augmentation mode: %s'%param.aug_mode)
        x = x.contiguous()
    return x


# We implement the following differentiable augmentation strategies based on the co
def rand_scale(x, param):
    # x>1, max scale
    # sx, sy: (0, +oo), 1: orignial size, 0.5: enlarge 2 times
    ratio = param.ratio_scale
    set_seed_DiffAug(param)
    sx = torch.rand(x.shape[0]) * (ratio - 1.0/ratio) + 1.0/ratio
    set_seed_DiffAug(param)
    sy = torch.rand(x.shape[0]) * (ratio - 1.0/ratio) + 1.0/ratio
    theta = [[[sx[i], 0,  0],
            [0,  sy[i], 0],] for i in range(x.shape[0])]
    theta = torch.tensor(theta, dtype=torch.float)
    if param.Siamese: # Siamese augmentation:
        theta[:] = theta[0]
    grid = F.affine_grid(theta, x.shape).to(x.device)
    x = F.grid_sample(x, grid)
    return x


def rand_rotate(x, param): # [-180, 180], 90: anticlockwise 90 degree
    ratio = param.ratio_rotate
    set_seed_DiffAug(param)
    theta = (torch.rand(x.shape[0]) - 0.5) * 2 * ratio / 180 * float(np.pi)
    theta = [[[torch.cos(theta[i]), torch.sin(-theta[i]), 0],
        [torch.sin(theta[i]), torch.cos(theta[i]),  0],]  for i in range(x.shape[0]
    theta = torch.tensor(theta, dtype=torch.float)
    if param.Siamese: # Siamese augmentation:
        theta[:] = theta[0]
    grid = F.affine_grid(theta, x.shape).to(x.device)
```

```
    x = F.grid_sample(x, grid)
    return x


def rand_flip(x, param):
    prob = param.prob_flip
    set_seed_DiffAug(param)
    randf = torch.rand(x.size(0), 1, 1, 1, device=x.device)
    if param.Siamese: # Siamese augmentation:
        randf[:] = randf[0]
    return torch.where(randf < prob, x.flip(3), x)


def rand_brightness(x, param):
    ratio = param.brightness
    set_seed_DiffAug(param)
    randb = torch.rand(x.size(0), 1, 1, 1, dtype=x.dtype, device=x.device)
    if param.Siamese:  # Siamese augmentation:
        randb[:] = randb[0]
    x = x + (randb - 0.5)*ratio
    return x


def rand_saturation(x, param):
    ratio = param.saturation
    x_mean = x.mean(dim=1, keepdim=True)
    set_seed_DiffAug(param)
    rands = torch.rand(x.size(0), 1, 1, 1, dtype=x.dtype, device=x.device)
    if param.Siamese:  # Siamese augmentation:
        rands[:] = rands[0]
    x = (x - x_mean) * (rands * ratio) + x_mean
    return x


def rand_contrast(x, param):
    ratio = param.contrast
    x_mean = x.mean(dim=[1, 2, 3], keepdim=True)
    set_seed_DiffAug(param)
    randc = torch.rand(x.size(0), 1, 1, 1, dtype=x.dtype, device=x.device)
    if param.Siamese:  # Siamese augmentation:
        randc[:] = randc[0]
    x = (x - x_mean) * (randc + ratio) + x_mean
    return x


def rand_crop(x, param):
    # The image is padded on its surrounding and then cropped.
    ratio = param.ratio_crop_pad
    shift_x, shift_y = int(x.size(2) * ratio + 0.5), int(x.size(3) * ratio + 0.5)
    set_seed_DiffAug(param)
    translation_x = torch.randint(-shift_x, shift_x + 1, size=[x.size(0), 1, 1], de
    set_seed_DiffAug(param)
    translation_y = torch.randint(-shift_y, shift_y + 1, size=[x.size(0), 1, 1], de
    if param.Siamese:  # Siamese augmentation:
        translation_x[:] = translation_x[0]
        translation_y[:] = translation_y[0]
```

```
    grid_batch, grid_x, grid_y = torch.meshgrid(
        torch.arange(x.size(0), dtype=torch.long, device=x.device),
        torch.arange(x.size(2), dtype=torch.long, device=x.device),
        torch.arange(x.size(3), dtype=torch.long, device=x.device),
    )
    grid_x = torch.clamp(grid_x + translation_x + 1, 0, x.size(2) + 1)
    grid_y = torch.clamp(grid_y + translation_y + 1, 0, x.size(3) + 1)
    x_pad = F.pad(x, [1, 1, 1, 1, 0, 0, 0, 0])
    x = x_pad.permute(0, 2, 3, 1).contiguous()[grid_batch, grid_x, grid_y].permute(
    return x


def rand_cutout(x, param):
    ratio = param.ratio_cutout
    cutout_size = int(x.size(2) * ratio + 0.5), int(x.size(3) * ratio + 0.5)
    set_seed_DiffAug(param)
    offset_x = torch.randint(0, x.size(2) + (1 - cutout_size[0] % 2), size=[x.size(
    set_seed_DiffAug(param)
    offset_y = torch.randint(0, x.size(3) + (1 - cutout_size[1] % 2), size=[x.size(
    if param.Siamese:  # Siamese augmentation:
        offset_x[:] = offset_x[0]
        offset_y[:] = offset_y[0]
    grid_batch, grid_x, grid_y = torch.meshgrid(
        torch.arange(x.size(0), dtype=torch.long, device=x.device),
        torch.arange(cutout_size[0], dtype=torch.long, device=x.device),
        torch.arange(cutout_size[1], dtype=torch.long, device=x.device),
    )
    grid_x = torch.clamp(grid_x + offset_x - cutout_size[0] // 2, min=0, max=x.size
    grid_y = torch.clamp(grid_y + offset_y - cutout_size[1] // 2, min=0, max=x.size
    mask = torch.ones(x.size(0), x.size(2), x.size(3), dtype=x.dtype, device=x.devi
    mask[grid_batch, grid_x, grid_y] = 0
    x = x * mask.unsqueeze(1)
    return x


AUGMENT_FNS = {
    'color': [rand_brightness, rand_saturation, rand_contrast],
    'crop': [rand_crop],
    'cutout': [rand_cutout],
    'flip': [rand_flip],
    'scale': [rand_scale],
    'rotate': [rand_rotate],
}
```

```
C:\Users\rupin\AppData\Local\Temp\ipykernel_19572\760089213.py:9: DeprecationWarning
: Please import `rotate` from the `scipy.ndimage` namespace; the `scipy.ndimage.inte
rpolation` namespace is deprecated and will be removed in SciPy 2.0.0.
  from scipy.ndimage.interpolation import rotate as scipyrotate
```

In [25]:
```
import torch
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset, random_split, ConcatDataset
import time
from torchvision import transforms
import matplotlib.pyplot as plt
from torchvision.utils import make_grid
```

```
import os
from PIL import Image
import pandas as pd
```

In [26]:
```
# MHIST LOADER

class MHISTDataset(Dataset):
    def __init__(self, annotations_file, img_dir, partition='train', transform=None

        self.img_labels = pd.read_csv(annotations_file)
        self.img_dir = img_dir
        self.transform = transform
        self.partition = partition

        self.img_labels = self.img_labels[self.img_labels['Partition'] == partition

        self.label_mapping = {'SSA': 0, 'HP': 1}

    def __len__(self):
        return len(self.img_labels)

    def __getitem__(self, idx):
        img_name = os.path.join(self.img_dir, self.img_labels.iloc[idx, 0])
        image = Image.open(img_name).convert("RGB")

        label_str = self.img_labels.iloc[idx, 1]
        label = self.label_mapping[label_str]

        if self.transform:
            image = self.transform(image)

        return image, label
```

In [27]:
```
class Args:
    def __init__(self):
        self.device ='cpu'
        self.lr_net = 0.01
        self.epoch_eval_train = 20
        self.batch_train = 128
        self.dsa = False
        self.dsa_strategy = ''
        self.dc_aug_param = None
        self.dis_metric = 'cos'

args = Args()
```

In [28]:
```
# Task 1 - Question 2 - Part a

img_dir = "C:/Users/rupin/Downloads/ECE1512_2024F_ProjectA_submission_files/submiss
annotations_file = r"C:\Users\rupin\Downloads\ECE1512_2024F_ProjectA_submission_fil

import torch
import torch.optim as optim
from torch.optim.lr_scheduler import CosineAnnealingLR
from torch.utils.data import DataLoader
```

```
from torchvision import transforms
import torch.nn.functional as F
from tqdm import tqdm
import time

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# Define the image transformations
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])

# Load MHIST Dataset
mhist_dataset = MHISTDataset(annotations_file=annotations_file, img_dir=img_dir, tr
batch_size = 128
num_classes = 2

# Split into training and testing sets
train_size = int(0.8 * len(mhist_dataset))
test_size = len(mhist_dataset) - train_size
train_dataset, test_dataset = torch.utils.data.random_split(mhist_dataset, [train_s

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

model_name = 'ConvNetW64'
channel = 3  # RGB images
im_size = (224, 224)  # Image size
model = get_network(model_name, channel, num_classes, im_size).to(device)

optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9, weight_decay=5e-4)
scheduler = CosineAnnealingLR(optimizer, T_max=20)
criterion = nn.CrossEntropyLoss()

def compute_flops(model, input_size):
    inputs = torch.randn(input_size).to(device)
    macs = profile_macs(model, inputs)
    flops = macs * 2
    return flops

def evaluate_model(model, test_loader):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in test_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    accuracy = 100 * correct / total
```

```
    return accuracy

epochs = 20
for epoch_num in range(epochs):
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0

    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()

        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

        running_loss += loss.item()
    scheduler.step()

    train_acc = 100 * correct / total
    print(f"Epoch {epoch_num + 1}/{epochs}, Loss: {running_loss:.4f}, Accuracy: {tr

test_accuracy = evaluate_model(model, test_loader)
print(f"Test Accuracy: {test_accuracy:.2f}%")
```

```
Using device: cpu
Epoch 1/20, Loss: 144.1497, Accuracy: 58.74%
Epoch 2/20, Loss: 45.0875, Accuracy: 64.77%
Epoch 3/20, Loss: 14.1904, Accuracy: 75.23%
Epoch 4/20, Loss: 5.9388, Accuracy: 82.76%
Epoch 5/20, Loss: 4.0381, Accuracy: 87.18%
Epoch 6/20, Loss: 3.0991, Accuracy: 91.15%
Epoch 7/20, Loss: 2.3857, Accuracy: 94.94%
Epoch 8/20, Loss: 2.0017, Accuracy: 96.44%
Epoch 9/20, Loss: 1.6930, Accuracy: 97.64%
Epoch 10/20, Loss: 1.4931, Accuracy: 98.16%
Epoch 11/20, Loss: 1.2817, Accuracy: 98.91%
Epoch 12/20, Loss: 1.1753, Accuracy: 99.25%
Epoch 13/20, Loss: 1.0687, Accuracy: 99.60%
Epoch 14/20, Loss: 1.0172, Accuracy: 99.83%
Epoch 15/20, Loss: 0.9650, Accuracy: 99.83%
Epoch 16/20, Loss: 0.9304, Accuracy: 100.00%
Epoch 17/20, Loss: 0.9088, Accuracy: 100.00%
Epoch 18/20, Loss: 0.8980, Accuracy: 100.00%
Epoch 19/20, Loss: 0.8838, Accuracy: 100.00%
Epoch 20/20, Loss: 0.8820, Accuracy: 100.00%
Test Accuracy: 71.72%
```

In [29]:
```
# Task 1 - Question 2 - Part b
```

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

img_dir = "C:/Users/rupin/Downloads/ECE1512_2024F_ProjectA_submission_files/submiss
annotations_file = r"C:\Users\rupin\Downloads\ECE1512_2024F_ProjectA_submission_fil

transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])

mhist_dataset = MHISTDataset(annotations_file=annotations_file, img_dir=img_dir, tr
batch_size = 128
real_data_loader = DataLoader(mhist_dataset, batch_size=batch_size, shuffle=True)

num_classes = 2
channel = 3
im_size = (224, 224)
synthetic_data = torch.randn(num_classes, channel, im_size[0], im_size[1], requires

model_name = 'ConvNetD7'
model = get_network(model_name, channel, num_classes, im_size).to(device)

K = 200
T = 10
eta_S = 0.1
lambda_param = 0.01

optimizer_syn = optim.SGD([synthetic_data], lr=eta_S)

class Args:
    def __init__(self, device, dis_metric='cos'):
        self.device = device
        self.dis_metric = dis_metric

args = Args(device=device)

for k in range(K):
    # Randomly initialize the model parameters
    model.apply(lambda m: m.reset_parameters() if hasattr(m, 'reset_parameters') el
    optimizer_model = optim.SGD(model.parameters(), lr=0.01)

    for t in range(T):
        optimizer_syn.zero_grad()
        optimizer_model.zero_grad()

        real_images, real_labels = next(iter(real_data_loader))
        real_images, real_labels = real_images.to(device), real_labels.to(device)

        indices = torch.randperm(real_images.size(0))[:2]
        real_images_subset = real_images[indices]
        real_labels_subset = real_labels[indices]

        outputs_real = model(real_images_subset)
        outputs_syn = model(synthetic_data)
```

```
        loss = match_loss(outputs_syn, outputs_real, args) + lambda_param
        loss.backward()
        optimizer_syn.step()

    print(f"Iteration {k + 1}/{K}, SAM Loss: {loss.item()}")
```

```
Using device: cpu
Iteration 1/200, SAM Loss: 1.3139328956604004
Iteration 2/200, SAM Loss: 0.055756933987140656
Iteration 3/200, SAM Loss: 0.028593668714165688
Iteration 4/200, SAM Loss: 0.07843620330095291
Iteration 5/200, SAM Loss: 0.8124338984489441
Iteration 6/200, SAM Loss: 0.018386075273156166
Iteration 7/200, SAM Loss: 0.04344416409730911
Iteration 8/200, SAM Loss: 0.46725964546203613
Iteration 9/200, SAM Loss: 0.0379529669880867
Iteration 10/200, SAM Loss: 0.15229232609272003
Iteration 11/200, SAM Loss: 0.18234969675540924
Iteration 12/200, SAM Loss: 0.027177700772881508
Iteration 13/200, SAM Loss: 0.05029202252626419
Iteration 14/200, SAM Loss: 0.12217003852128983
Iteration 15/200, SAM Loss: 0.029774555936455727
Iteration 16/200, SAM Loss: 0.2619844675064087
Iteration 17/200, SAM Loss: 0.18472470343112946
Iteration 18/200, SAM Loss: 0.10635150223970413
Iteration 19/200, SAM Loss: 0.01598329283297062
Iteration 20/200, SAM Loss: 0.8956540822982788
Iteration 21/200, SAM Loss: 0.13103307783603668
Iteration 22/200, SAM Loss: 0.12565208971500397
Iteration 23/200, SAM Loss: 0.20528562366962433
Iteration 24/200, SAM Loss: 0.13512666523456573
Iteration 25/200, SAM Loss: 0.046334750950336456
Iteration 26/200, SAM Loss: 0.06733299046754837
Iteration 27/200, SAM Loss: 0.03697068244218826
Iteration 28/200, SAM Loss: 0.10944540053606033
Iteration 29/200, SAM Loss: 0.028074750676751137
Iteration 30/200, SAM Loss: 0.05312085896730423
Iteration 31/200, SAM Loss: 0.5923842191696167
Iteration 32/200, SAM Loss: 0.4121595025062561
Iteration 33/200, SAM Loss: 0.7208618521690369
Iteration 34/200, SAM Loss: 0.07303036004304886
Iteration 35/200, SAM Loss: 0.3168885111808777
Iteration 36/200, SAM Loss: 0.27961277961730957
Iteration 37/200, SAM Loss: 0.3464624881744385
Iteration 38/200, SAM Loss: 0.05675721913576126
Iteration 39/200, SAM Loss: 0.014644155278801918
Iteration 40/200, SAM Loss: 0.017499634996056557
Iteration 41/200, SAM Loss: 0.060796745121479034
Iteration 42/200, SAM Loss: 0.2036859542131424
Iteration 43/200, SAM Loss: 0.4300534129142761
Iteration 44/200, SAM Loss: 0.13352538645267487
Iteration 45/200, SAM Loss: 0.013445386663079262
Iteration 46/200, SAM Loss: 0.730274498462677
Iteration 47/200, SAM Loss: 0.0894930437207222
Iteration 48/200, SAM Loss: 0.07865387946367264
Iteration 49/200, SAM Loss: 0.017155298963189125
Iteration 50/200, SAM Loss: 0.03130365163087845
Iteration 51/200, SAM Loss: 0.9145160913467407
Iteration 52/200, SAM Loss: 0.29158735275268555
Iteration 53/200, SAM Loss: 0.16154684126377106
Iteration 54/200, SAM Loss: 0.488844096660614
Iteration 55/200, SAM Loss: 0.18342776596546173
```

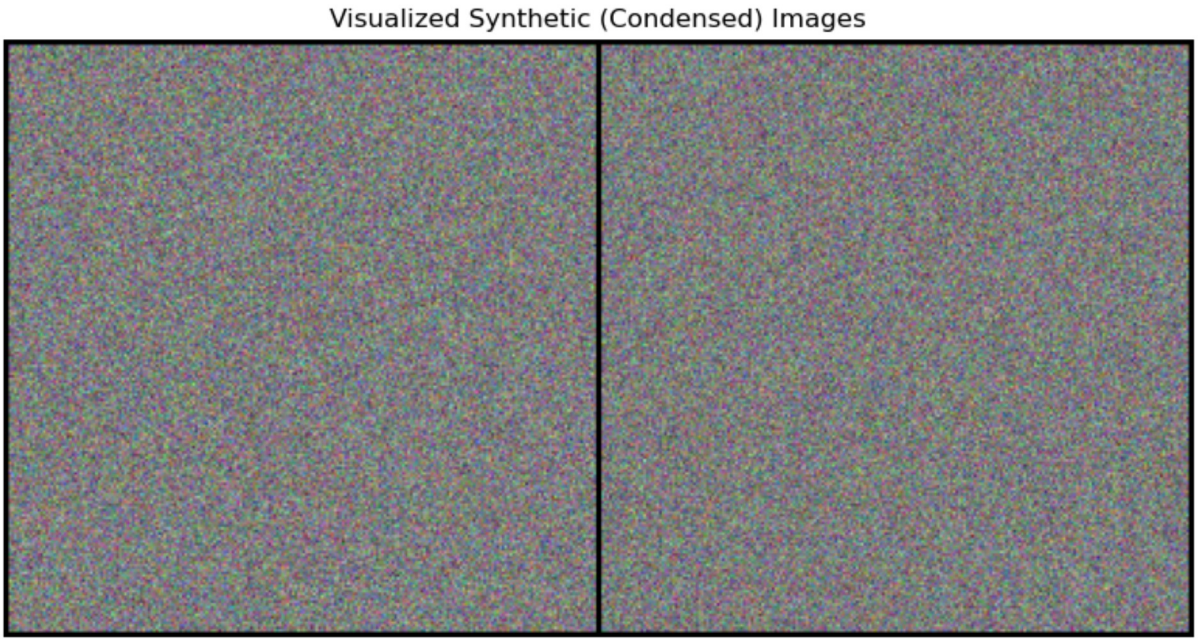
```
Iteration 56/200, SAM Loss: 0.556141197681427
Iteration 57/200, SAM Loss: 0.04204631596803665
Iteration 58/200, SAM Loss: 0.18076808750629425
Iteration 59/200, SAM Loss: 0.057893164455890656
Iteration 60/200, SAM Loss: 0.2028353363275528
Iteration 61/200, SAM Loss: 1.4177563190460205
Iteration 62/200, SAM Loss: 0.10621834546327591
Iteration 63/200, SAM Loss: 0.2961006760597229
Iteration 64/200, SAM Loss: 0.09224671870470047
Iteration 65/200, SAM Loss: 0.041900403797626495
Iteration 66/200, SAM Loss: 0.3097159266471863
Iteration 67/200, SAM Loss: 0.04921329766511917
Iteration 68/200, SAM Loss: 0.11981392651796341
Iteration 69/200, SAM Loss: 0.052866704761981964
Iteration 70/200, SAM Loss: 0.2298167496919632
Iteration 71/200, SAM Loss: 0.01836235262453556
Iteration 72/200, SAM Loss: 0.18103273212909698
Iteration 73/200, SAM Loss: 0.9353293776512146
Iteration 74/200, SAM Loss: 0.04379863291978836
Iteration 75/200, SAM Loss: 0.1429019719362259
Iteration 76/200, SAM Loss: 0.08967519551515579
Iteration 77/200, SAM Loss: 0.12837345898151398
Iteration 78/200, SAM Loss: 0.0851140096783638
Iteration 79/200, SAM Loss: 0.1856357604265213
Iteration 80/200, SAM Loss: 0.7789507508277893
Iteration 81/200, SAM Loss: 0.038852520287036896
Iteration 82/200, SAM Loss: 0.17865528166294098
Iteration 83/200, SAM Loss: 0.4380595088005066
Iteration 84/200, SAM Loss: 0.06497205048799515
Iteration 85/200, SAM Loss: 1.6494450569152832
Iteration 86/200, SAM Loss: 0.12168533354997635
Iteration 87/200, SAM Loss: 0.3360442519187927
Iteration 88/200, SAM Loss: 0.17928190529346466
Iteration 89/200, SAM Loss: 0.4882991313934326
Iteration 90/200, SAM Loss: 0.03550858050584793
Iteration 91/200, SAM Loss: 0.2750948667526245
Iteration 92/200, SAM Loss: 0.035171933472156525
Iteration 93/200, SAM Loss: 0.02189774252474308
Iteration 94/200, SAM Loss: 0.056023187935352325
Iteration 95/200, SAM Loss: 0.901598334312439
Iteration 96/200, SAM Loss: 0.08635527640581131
Iteration 97/200, SAM Loss: 0.037247784435749054
Iteration 98/200, SAM Loss: 0.04878927022218704
Iteration 99/200, SAM Loss: 0.7768771648406982
Iteration 100/200, SAM Loss: 0.03758091479539871
Iteration 101/200, SAM Loss: 0.06361640244722366
Iteration 102/200, SAM Loss: 0.1293358951807022
Iteration 103/200, SAM Loss: 0.12589241564273834
Iteration 104/200, SAM Loss: 0.04025036841630936
Iteration 105/200, SAM Loss: 1.40143620967865
Iteration 106/200, SAM Loss: 0.12335003167390823
Iteration 107/200, SAM Loss: 0.1777672916650772
Iteration 108/200, SAM Loss: 0.09422386437654495
Iteration 109/200, SAM Loss: 0.08362198621034622
Iteration 110/200, SAM Loss: 0.09149707108736038
Iteration 111/200, SAM Loss: 0.056076712906360626
```

```
Iteration 112/200, SAM Loss: 0.10251093655824661
Iteration 113/200, SAM Loss: 0.17539937794208527
Iteration 114/200, SAM Loss: 0.14913715422153473
Iteration 115/200, SAM Loss: 0.040385372936725616
Iteration 116/200, SAM Loss: 0.3145732283592224
Iteration 117/200, SAM Loss: 0.9944014549255371
Iteration 118/200, SAM Loss: 0.16574300825595856
Iteration 119/200, SAM Loss: 0.12929768860340118
Iteration 120/200, SAM Loss: 0.011812636628746986
Iteration 121/200, SAM Loss: 0.09460318833589554
Iteration 122/200, SAM Loss: 0.2634100317955017
Iteration 123/200, SAM Loss: 0.023177990689873695
Iteration 124/200, SAM Loss: 0.65901118516922
Iteration 125/200, SAM Loss: 0.1145550087094307
Iteration 126/200, SAM Loss: 0.01931900717318058
Iteration 127/200, SAM Loss: 0.2848992347717285
Iteration 128/200, SAM Loss: 0.08674246817827225
Iteration 129/200, SAM Loss: 0.03563458472490311
Iteration 130/200, SAM Loss: 0.15088321268558502
Iteration 131/200, SAM Loss: 0.05276186019182205
Iteration 132/200, SAM Loss: 0.20345087349414825
Iteration 133/200, SAM Loss: 0.1742432862520218
Iteration 134/200, SAM Loss: 0.23587454855442047
Iteration 135/200, SAM Loss: 0.0864834263920784
Iteration 136/200, SAM Loss: 0.18447668850421906
Iteration 137/200, SAM Loss: 0.12528397142887115
Iteration 138/200, SAM Loss: 0.043340571224689484
Iteration 139/200, SAM Loss: 0.2584289312362671
Iteration 140/200, SAM Loss: 0.17891110479831696
Iteration 141/200, SAM Loss: 0.12719793617725372
Iteration 142/200, SAM Loss: 0.09743870049715042
Iteration 143/200, SAM Loss: 0.12190795689821243
Iteration 144/200, SAM Loss: 0.019382664933800697
Iteration 145/200, SAM Loss: 0.6601561307907104
Iteration 146/200, SAM Loss: 0.0401153638958931
Iteration 147/200, SAM Loss: 0.1899731308221817
Iteration 148/200, SAM Loss: 0.18624962866306305
Iteration 149/200, SAM Loss: 1.2885663509368896
Iteration 150/200, SAM Loss: 0.07364512234926224
Iteration 151/200, SAM Loss: 0.10921234637498856
Iteration 152/200, SAM Loss: 0.06036479026079178
Iteration 153/200, SAM Loss: 0.042135901749134064
Iteration 154/200, SAM Loss: 0.08090157061815262
Iteration 155/200, SAM Loss: 0.06906045228242874
Iteration 156/200, SAM Loss: 0.23410262167453766
Iteration 157/200, SAM Loss: 0.2601439356803894
Iteration 158/200, SAM Loss: 0.37946122884750366
Iteration 159/200, SAM Loss: 0.1223728135228157
Iteration 160/200, SAM Loss: 0.05875105410814285
Iteration 161/200, SAM Loss: 0.05905861407518387
Iteration 162/200, SAM Loss: 0.04800880700349808
Iteration 163/200, SAM Loss: 0.17412038147449493
Iteration 164/200, SAM Loss: 0.05366212874650955
Iteration 165/200, SAM Loss: 0.2351330667734146
Iteration 166/200, SAM Loss: 0.24699975550174713
Iteration 167/200, SAM Loss: 0.09325189143419266
```

```
Iteration 168/200, SAM Loss: 0.021676549687981606
Iteration 169/200, SAM Loss: 0.0848841741681099
Iteration 170/200, SAM Loss: 0.015975603833794594
Iteration 171/200, SAM Loss: 0.03523535281419754
Iteration 172/200, SAM Loss: 0.09027958661317825
Iteration 173/200, SAM Loss: 0.01706922985613346
Iteration 174/200, SAM Loss: 0.31566476821899414
Iteration 175/200, SAM Loss: 0.021218368783593178
Iteration 176/200, SAM Loss: 0.06326103955507278
Iteration 177/200, SAM Loss: 0.3945666551589966
Iteration 178/200, SAM Loss: 0.03763110190629959
Iteration 179/200, SAM Loss: 0.028428802266716957
Iteration 180/200, SAM Loss: 0.17561109364032745
Iteration 181/200, SAM Loss: 0.20935262739658356
Iteration 182/200, SAM Loss: 0.16602130234241486
Iteration 183/200, SAM Loss: 0.15656007826328278
Iteration 184/200, SAM Loss: 0.2584623694419861
Iteration 185/200, SAM Loss: 0.06478668004274368
Iteration 186/200, SAM Loss: 1.2721495628356934
Iteration 187/200, SAM Loss: 0.17611391842365265
Iteration 188/200, SAM Loss: 0.16923590004444122
Iteration 189/200, SAM Loss: 0.236149862408638
Iteration 190/200, SAM Loss: 0.01703990437090397
Iteration 191/200, SAM Loss: 0.012336382642388344
Iteration 192/200, SAM Loss: 0.482740581035614
Iteration 193/200, SAM Loss: 0.11776978522539139
Iteration 194/200, SAM Loss: 0.16575898230075836
Iteration 195/200, SAM Loss: 0.07751632481813431
Iteration 196/200, SAM Loss: 0.13500483334064484
Iteration 197/200, SAM Loss: 0.09604383260011673
Iteration 198/200, SAM Loss: 0.23090530931949615
Iteration 199/200, SAM Loss: 0.03170258551836014
Iteration 200/200, SAM Loss: 0.01573331095278263
```

In [30]:

```python
# Task 1 - Question 2 - Part c
def visualize_synthetic_images(synthetic_data, num_classes=2):
    grid_img = make_grid(synthetic_data, nrow=num_classes, normalize=True, scale_ea

    plt.figure(figsize=(10, 5))
    plt.imshow(grid_img.permute(1, 2, 0).cpu().numpy())
    plt.title("Visualized Synthetic (Condensed) Images")
    plt.axis('off')
    plt.show()
visualize_synthetic_images(synthetic_data, num_classes=num_classes)
```

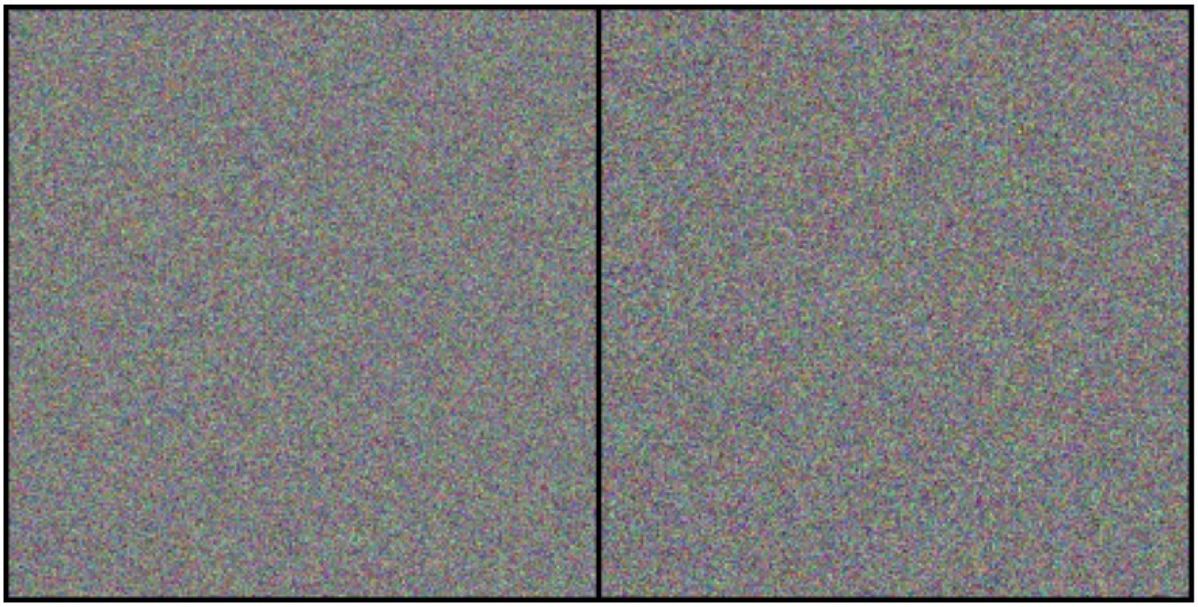


In [31]:
```
# Task 1 - Question 2 - Part d

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

img_dir = "C:/Users/rupin/Downloads/ECE1512_2024F_ProjectA_submission_files/submiss
annotations_file = r"C:\Users\rupin\Downloads\ECE1512_2024F_ProjectA_submission_fil

transform = transforms.Compose([
    transforms.Resize((224, 224)),  # Resize images to a fixed size (e.g., 224x224 
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])

mhist_dataset = MHISTDataset(annotations_file=annotations_file, img_dir=img_dir, tr
batch_size = 128
real_data_loader = DataLoader(mhist_dataset, batch_size=batch_size, shuffle=True)

num_classes = 2
channel = 3
im_size = (224, 224)

# Gaussian initialization for synthetic data
mean = 0.0
std = 1.0
synthetic_data = torch.normal(mean=mean, std=std, size=(num_classes, channel, im_si

def visualize_synthetic_images(synthetic_data, num_classes=2):

    grid_img = make_grid(synthetic_data.detach().cpu(), nrow=num_classes, normalize

    plt.figure(figsize=(10, 5))
    plt.imshow(grid_img.permute(1, 2, 0).numpy())
    plt.title("Visualized Synthetic (Condensed) Images")
    plt.axis('off')
```

```
    plt.show()

model_name = 'ConvNetD3'
model = get_network(model_name, channel, num_classes, im_size).to(device)


K = 200
T = 10
eta_S = 0.1
lambda_param = 0.01

optimizer_syn = optim.SGD([synthetic_data], lr=eta_S)

class Args:
    def __init__(self, device, dis_metric='cos'):
        self.device = device
        self.dis_metric = dis_metric

args = Args(device=device)

for k in range(K):
    model.apply(lambda m: m.reset_parameters() if hasattr(m, 'reset_parameters') el
    optimizer_model = optim.SGD(model.parameters(), lr=0.01)

    for t in range(T):
        optimizer_syn.zero_grad()
        optimizer_model.zero_grad()

        real_images, real_labels = next(iter(real_data_loader))
        real_images, real_labels = real_images.to(device), real_labels.to(device)

        indices = torch.randperm(real_images.size(0))[:2]
        real_images_subset = real_images[indices]
        real_labels_subset = real_labels[indices]

        outputs_real = model(real_images_subset)
        outputs_syn = model(synthetic_data)

        loss = match_loss(outputs_syn, outputs_real, args) + lambda_param
        loss.backward()
        optimizer_syn.step()

    print(f"Iteration {k + 1}/{K}, SAM Loss: {loss.item()}")

    if (k + 1) % 5 == 0:
        visualize_synthetic_images(synthetic_data, num_classes=num_classes)
```

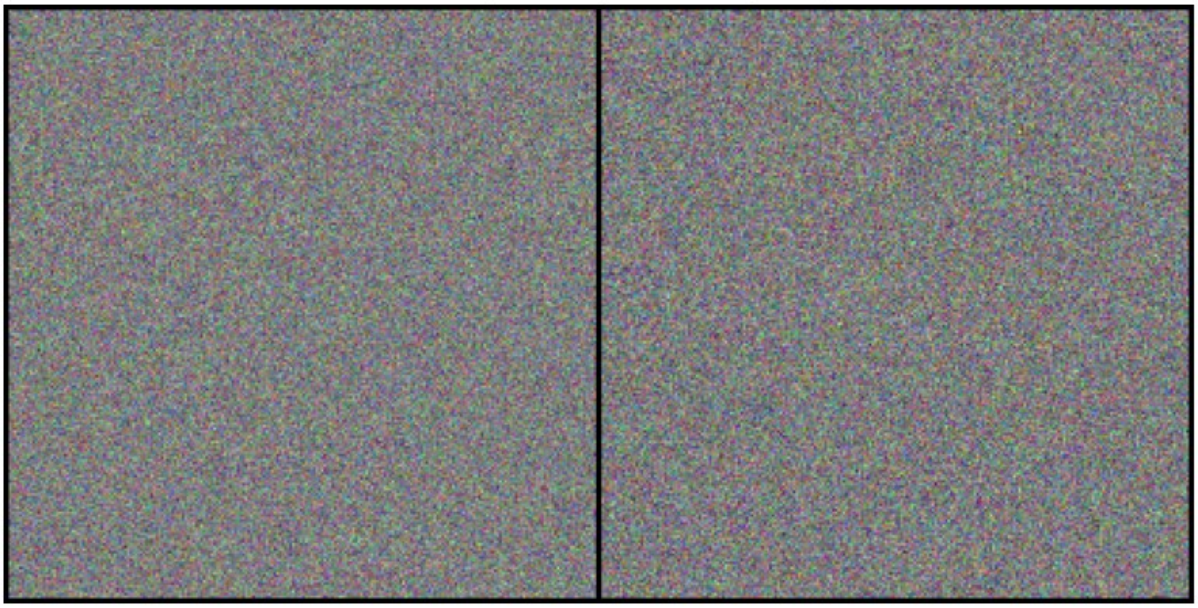
```
Using device: cpu
Iteration 1/200, SAM Loss: 0.9179607033729553
Iteration 2/200, SAM Loss: 0.1470094472169876
Iteration 3/200, SAM Loss: 0.5626929998397827
Iteration 4/200, SAM Loss: 0.5002634525299072
Iteration 5/200, SAM Loss: 0.9048941135406494
```

Visualized Synthetic (Condensed) Images



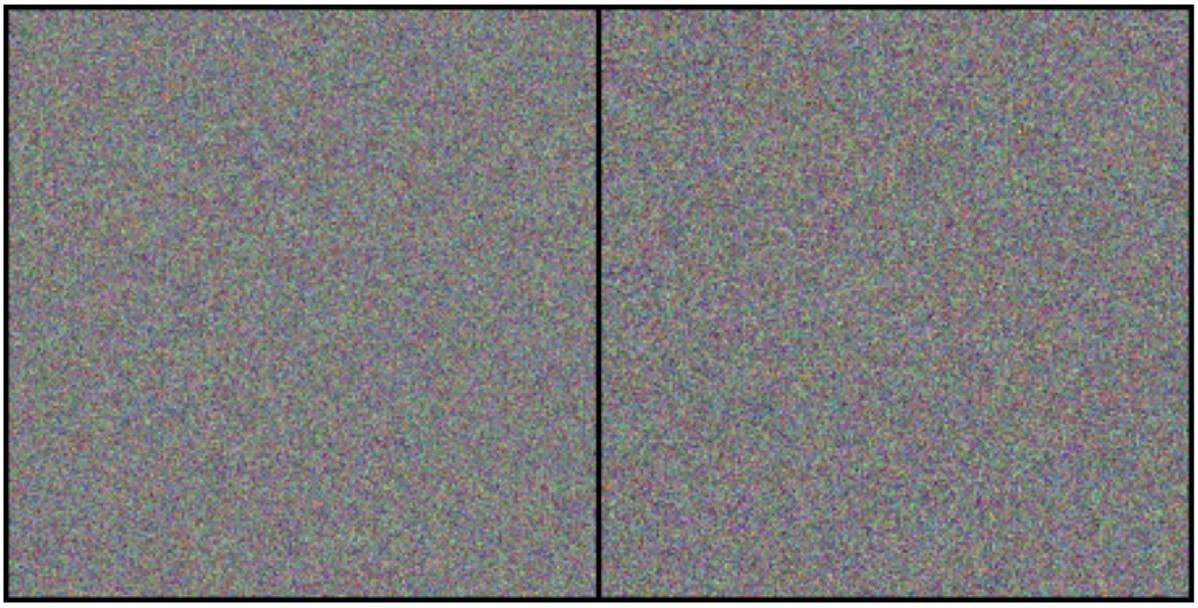
```
Iteration 6/200, SAM Loss: 1.8179564476013184
Iteration 7/200, SAM Loss: 0.08841288834810257
Iteration 8/200, SAM Loss: 0.2776220440864563
Iteration 9/200, SAM Loss: 0.2009405642747879
Iteration 10/200, SAM Loss: 1.1365973949432373
```

Visualized Synthetic (Condensed) Images



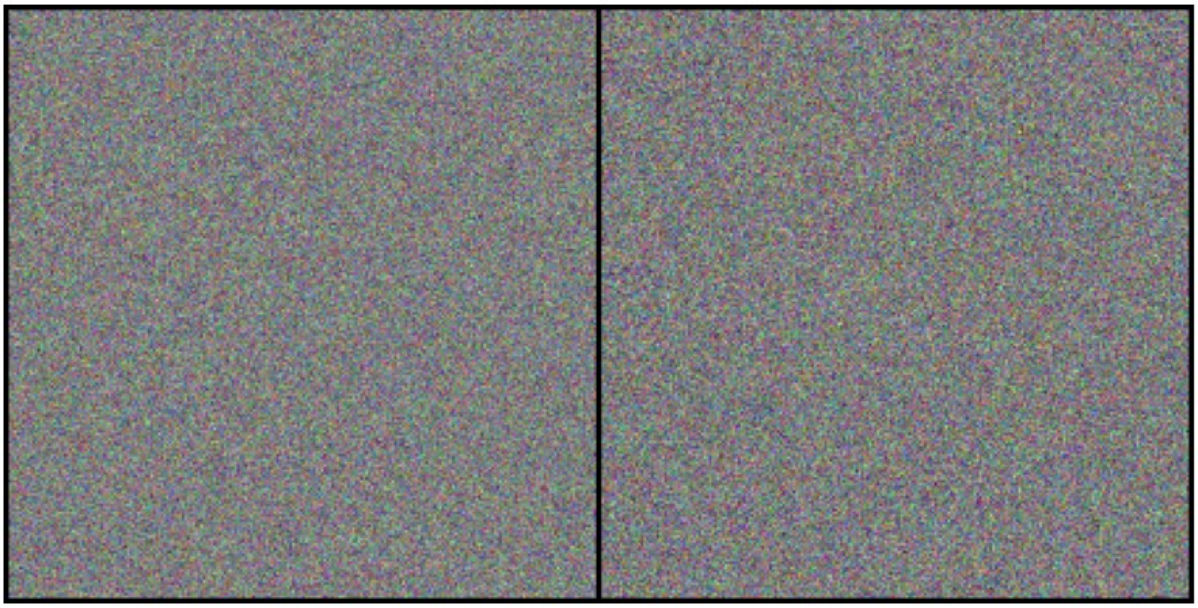
```
Iteration 11/200, SAM Loss: 0.6865944862365723
Iteration 12/200, SAM Loss: 0.35522574186325073
Iteration 13/200, SAM Loss: 0.2988209128379822
Iteration 14/200, SAM Loss: 0.849730908870697
Iteration 15/200, SAM Loss: 0.24848605692386627
```

Visualized Synthetic (Condensed) Images



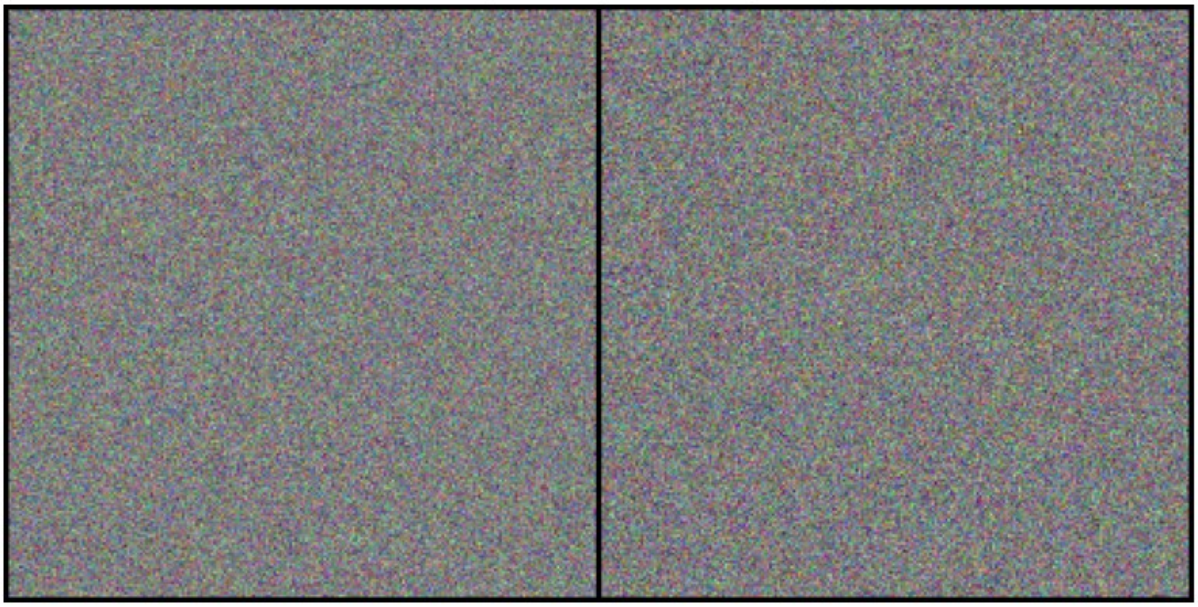
```
Iteration 16/200, SAM Loss: 0.3352375030517578
Iteration 17/200, SAM Loss: 0.19285400211811066
Iteration 18/200, SAM Loss: 0.09616471081972122
Iteration 19/200, SAM Loss: 0.4356839656829834
Iteration 20/200, SAM Loss: 0.9015045762062073
```

Visualized Synthetic (Condensed) Images



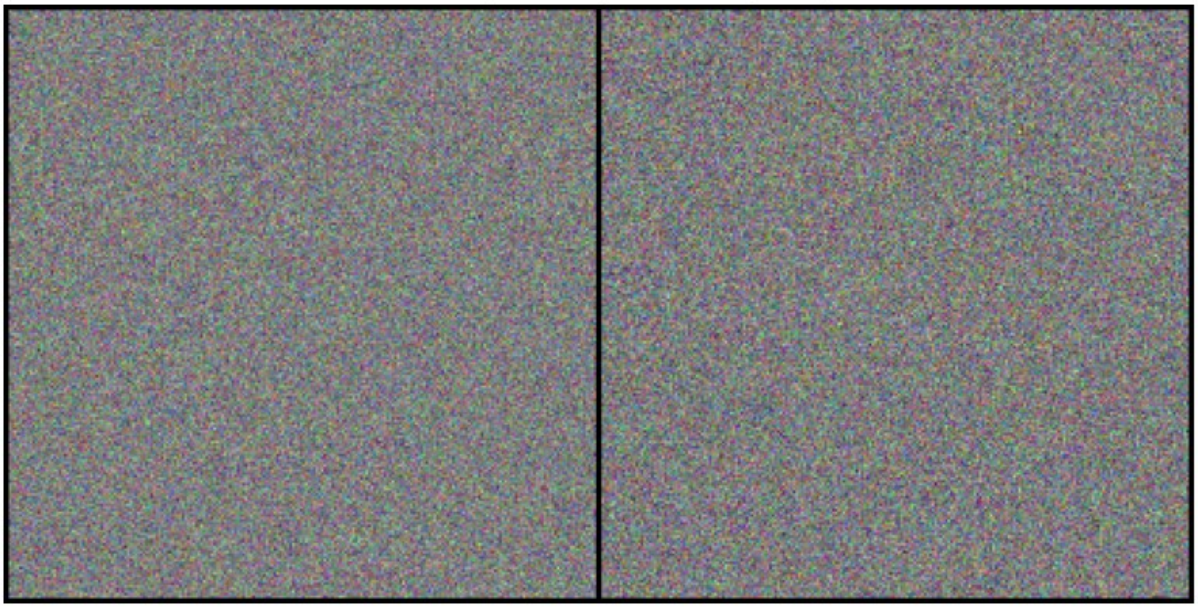
```
Iteration 21/200, SAM Loss: 0.055659420788288116
Iteration 22/200, SAM Loss: 0.17004425823688507
Iteration 23/200, SAM Loss: 0.34715574979782104
Iteration 24/200, SAM Loss: 0.10108990222215652
Iteration 25/200, SAM Loss: 1.3104461431503296
```

Visualized Synthetic (Condensed) Images



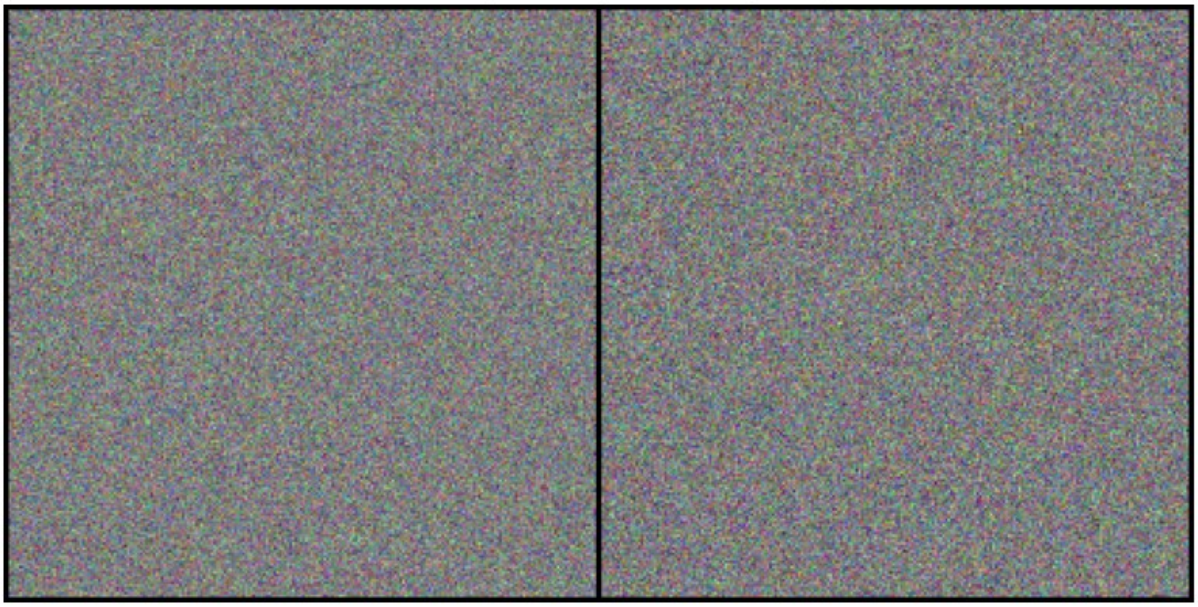
```
Iteration 26/200, SAM Loss: 0.2708700895309448
Iteration 27/200, SAM Loss: 0.624534010887146
Iteration 28/200, SAM Loss: 0.28962600231170654
Iteration 29/200, SAM Loss: 1.2376420497894287
Iteration 30/200, SAM Loss: 0.04275406152009964
```

Visualized Synthetic (Condensed) Images



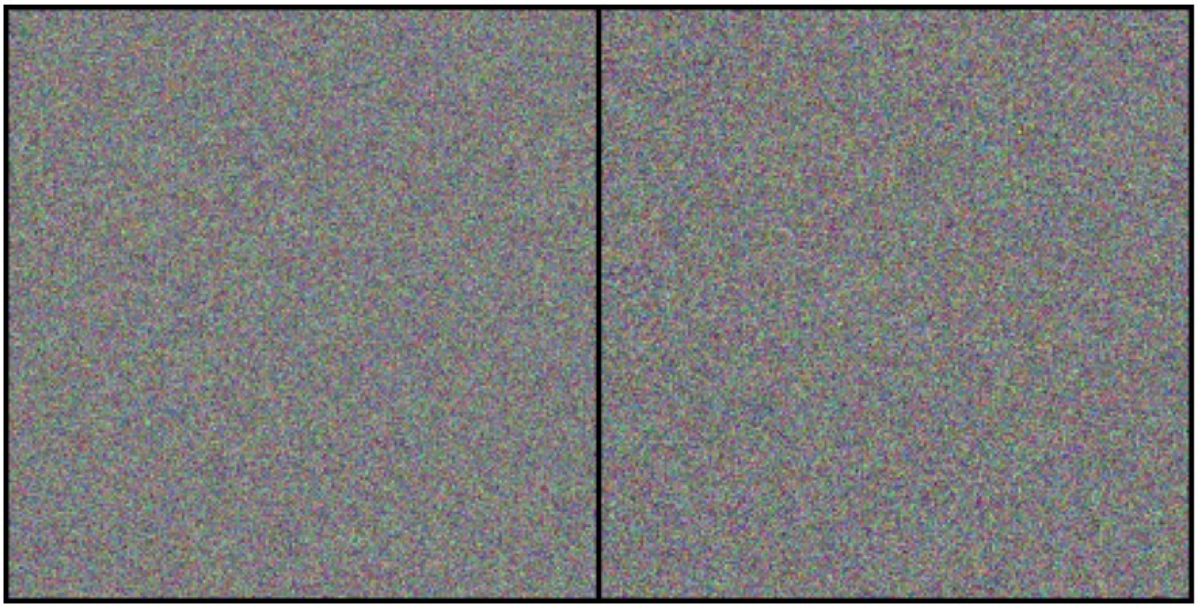
```
Iteration 31/200, SAM Loss: 0.3156421184539795
Iteration 32/200, SAM Loss: 1.8774863481521606
Iteration 33/200, SAM Loss: 0.19170619547367096
Iteration 34/200, SAM Loss: 0.13342483341693878
Iteration 35/200, SAM Loss: 0.07021690160036087
```

Visualized Synthetic (Condensed) Images



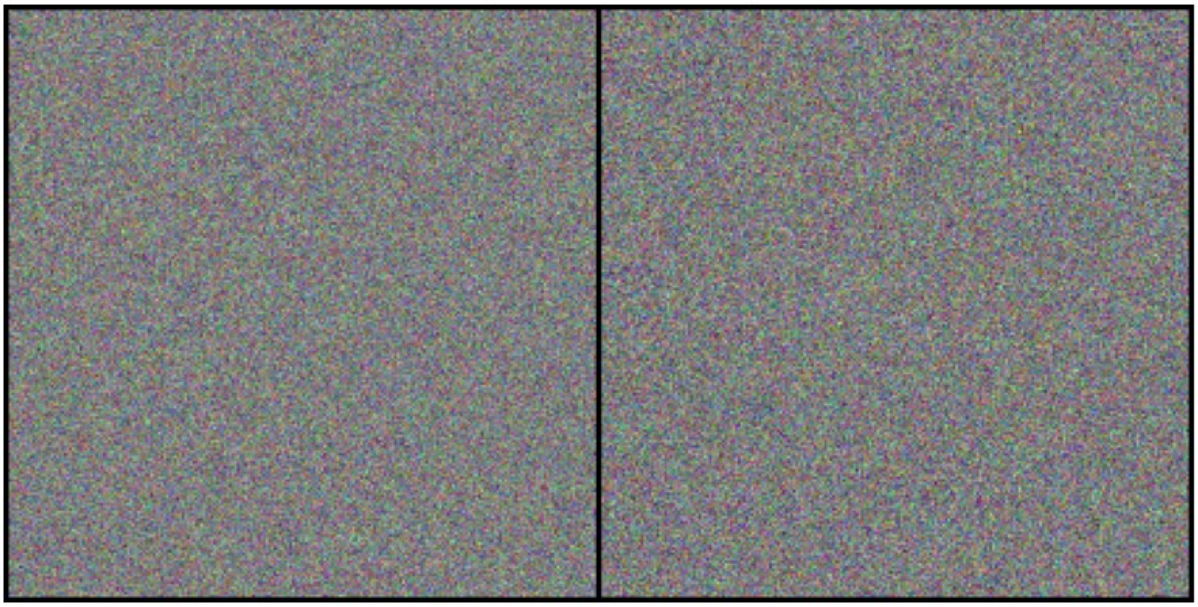
```
Iteration 36/200, SAM Loss: 0.20692963898181915
Iteration 37/200, SAM Loss: 0.4833504557609558
Iteration 38/200, SAM Loss: 0.4885404706001282
Iteration 39/200, SAM Loss: 0.2422472983598709
Iteration 40/200, SAM Loss: 1.0475369691848755
```

Visualized Synthetic (Condensed) Images



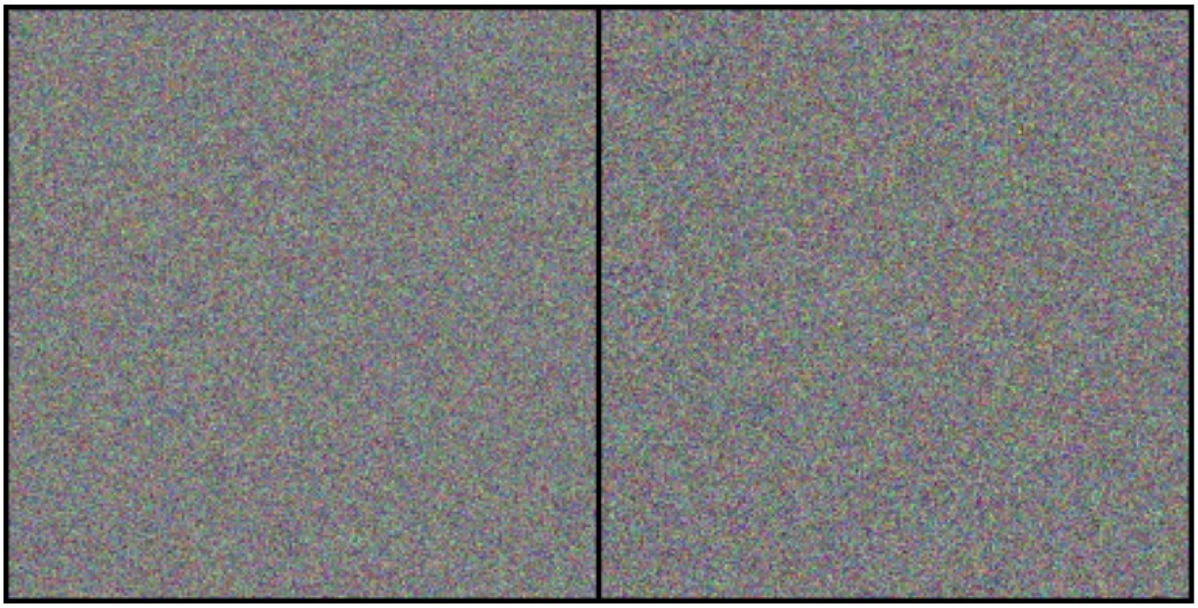
```
Iteration 41/200, SAM Loss: 1.2698590755462646
Iteration 42/200, SAM Loss: 0.7923212051391602
Iteration 43/200, SAM Loss: 0.12087655812501907
Iteration 44/200, SAM Loss: 1.7781665325164795
Iteration 45/200, SAM Loss: 0.23434562981128693
```

Visualized Synthetic (Condensed) Images



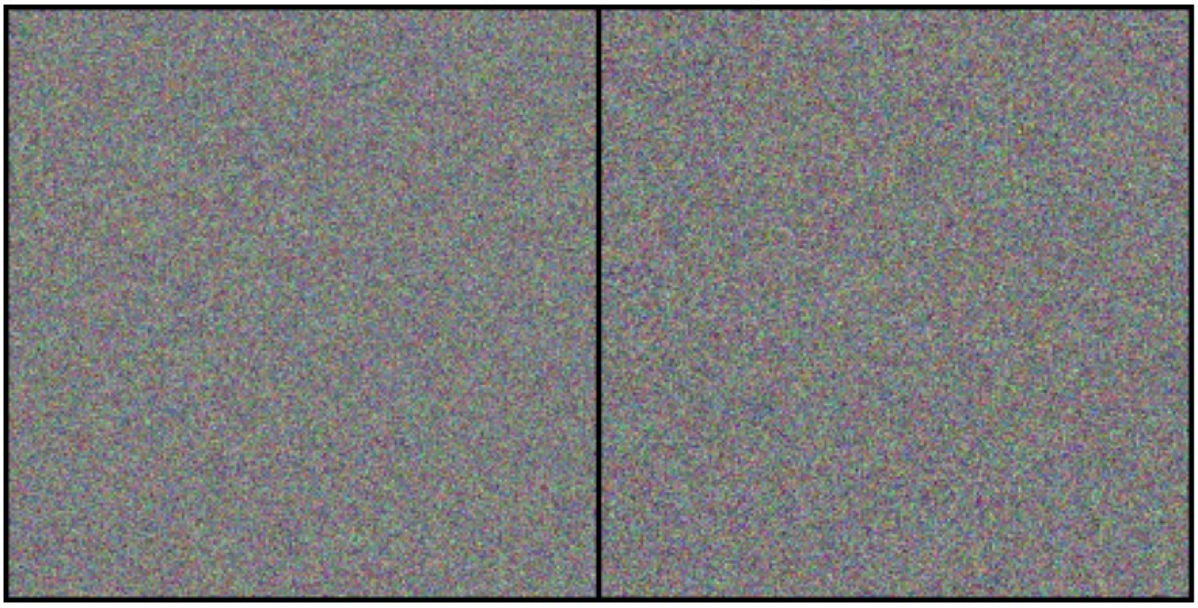
```
Iteration 46/200, SAM Loss: 0.5981769561767578
Iteration 47/200, SAM Loss: 0.8074577450752258
Iteration 48/200, SAM Loss: 0.779705822467804
Iteration 49/200, SAM Loss: 1.1527538299560547
Iteration 50/200, SAM Loss: 1.2970821857452393
```

Visualized Synthetic (Condensed) Images



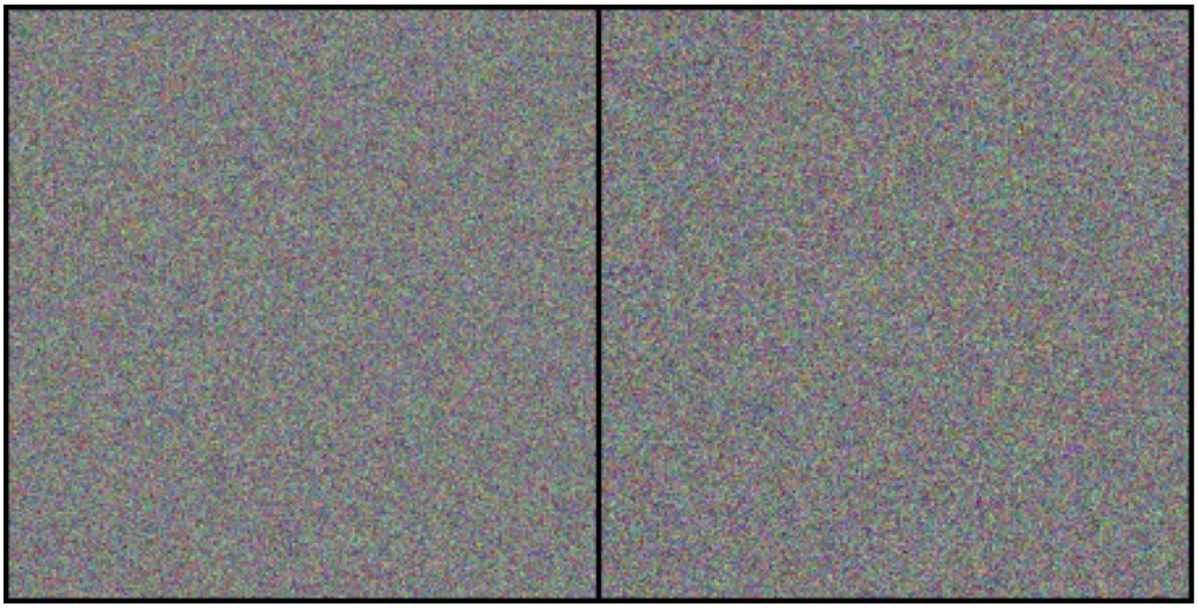
```
Iteration 51/200, SAM Loss: 0.929925799369812
Iteration 52/200, SAM Loss: 0.6338505148887634
Iteration 53/200, SAM Loss: 0.5299016237258911
Iteration 54/200, SAM Loss: 0.8199635744094849
Iteration 55/200, SAM Loss: 0.7014877796173096
```

Visualized Synthetic (Condensed) Images



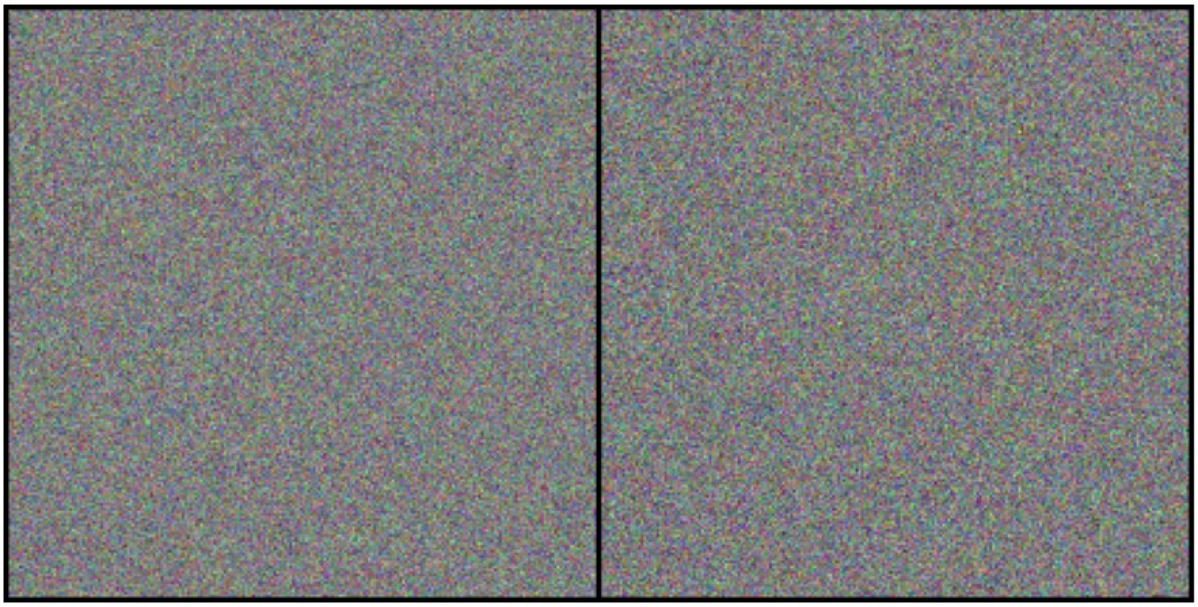
```
Iteration 56/200, SAM Loss: 0.48689669370651245
Iteration 57/200, SAM Loss: 0.8280671834945679
Iteration 58/200, SAM Loss: 0.28113895654678345
Iteration 59/200, SAM Loss: 0.8929292559623718
Iteration 60/200, SAM Loss: 0.1844901591539383
```

Visualized Synthetic (Condensed) Images



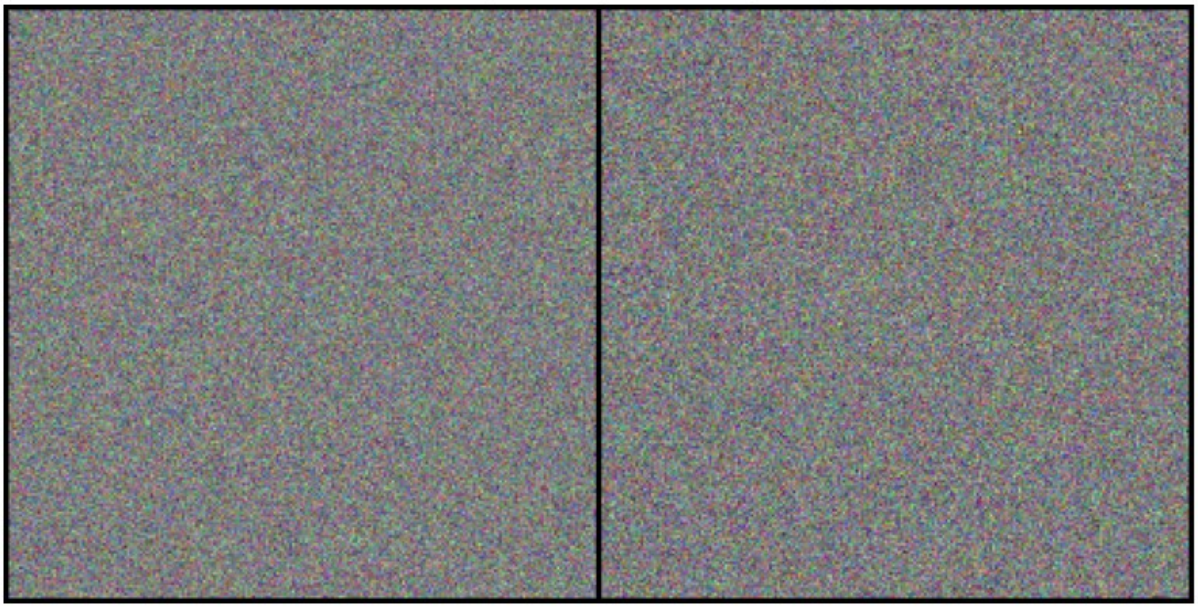
```
Iteration 61/200, SAM Loss: 1.4877256155014038
Iteration 62/200, SAM Loss: 0.2398625761270523
Iteration 63/200, SAM Loss: 0.05707252770662308
Iteration 64/200, SAM Loss: 0.2723240852355957
Iteration 65/200, SAM Loss: 0.05459136515855789
```

Visualized Synthetic (Condensed) Images



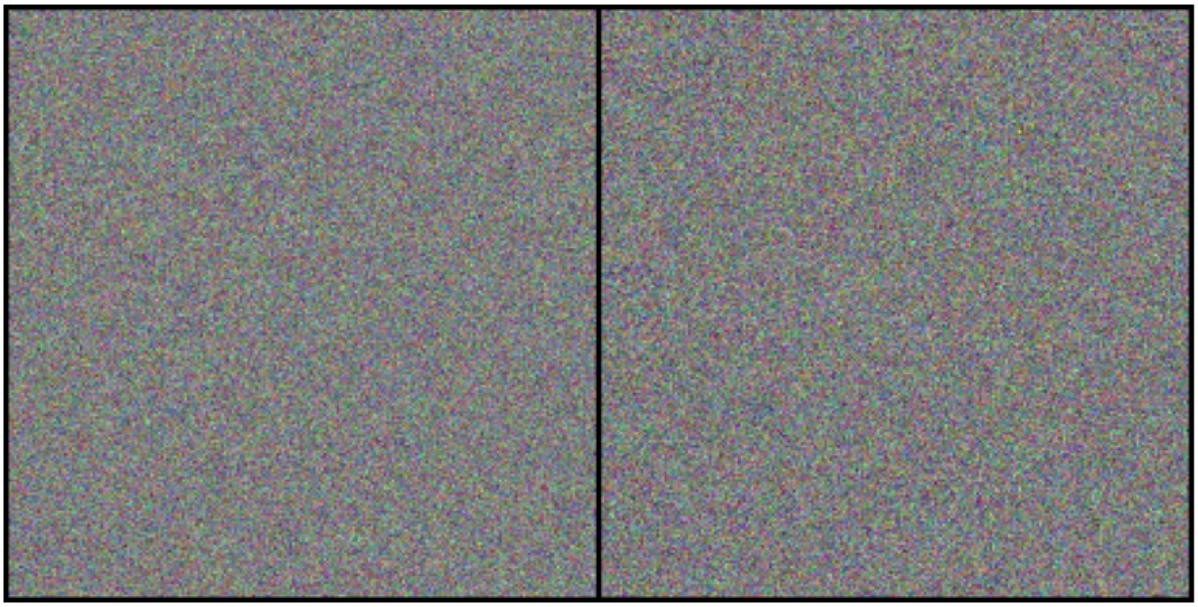
```
Iteration 66/200, SAM Loss: 1.4643038511276245
Iteration 67/200, SAM Loss: 0.13547058403491974
Iteration 68/200, SAM Loss: 1.4923608303070068
Iteration 69/200, SAM Loss: 0.4486526846885681
Iteration 70/200, SAM Loss: 1.1434181928634644
```

Visualized Synthetic (Condensed) Images



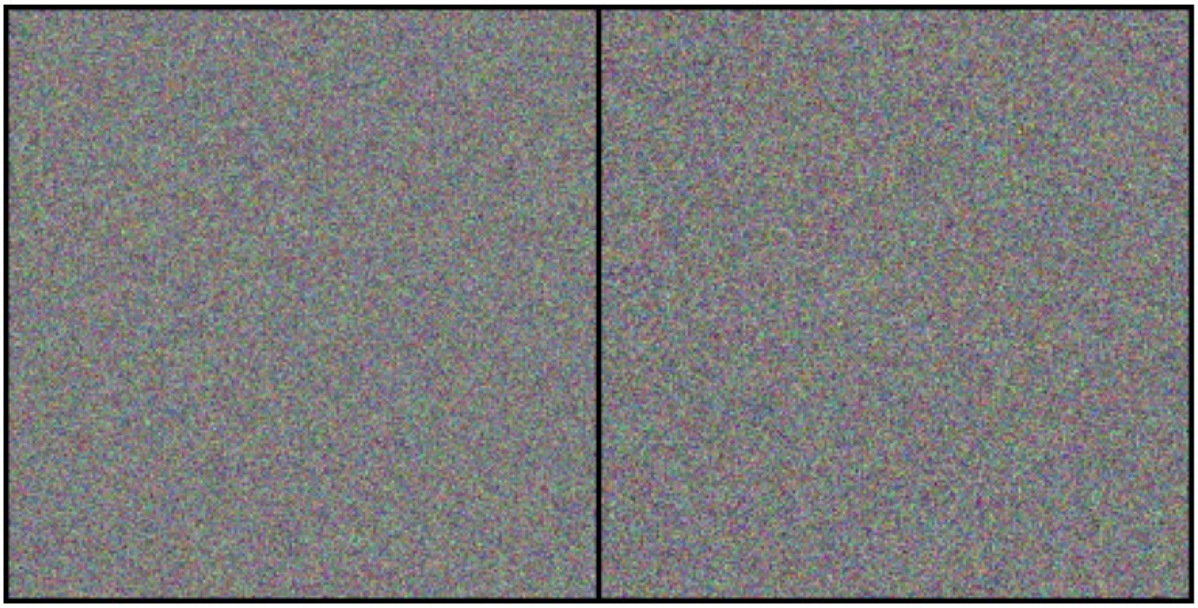
```
Iteration 71/200, SAM Loss: 0.36154747009277344
Iteration 72/200, SAM Loss: 0.2010027915239334
Iteration 73/200, SAM Loss: 1.9803974628448486
Iteration 74/200, SAM Loss: 0.04549182206392288
Iteration 75/200, SAM Loss: 0.10815901309251785
```

Visualized Synthetic (Condensed) Images



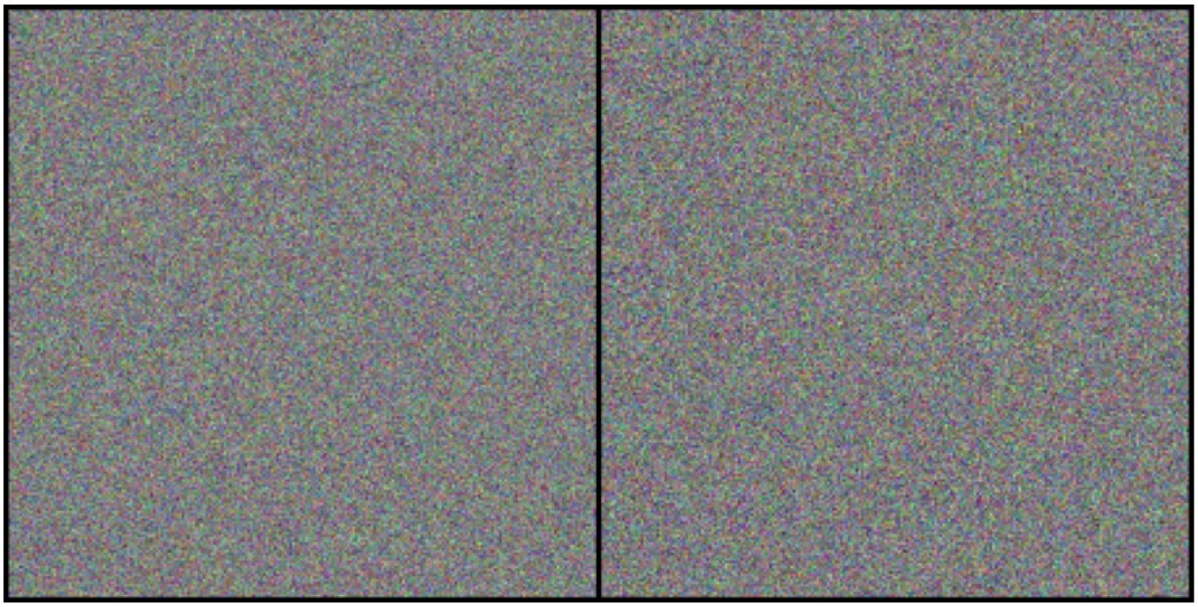
```
Iteration 76/200, SAM Loss: 1.8181475400924683
Iteration 77/200, SAM Loss: 0.44463932514190674
Iteration 78/200, SAM Loss: 0.5830071568489075
Iteration 79/200, SAM Loss: 1.0093008279800415
Iteration 80/200, SAM Loss: 0.34474319219589233
```

Visualized Synthetic (Condensed) Images



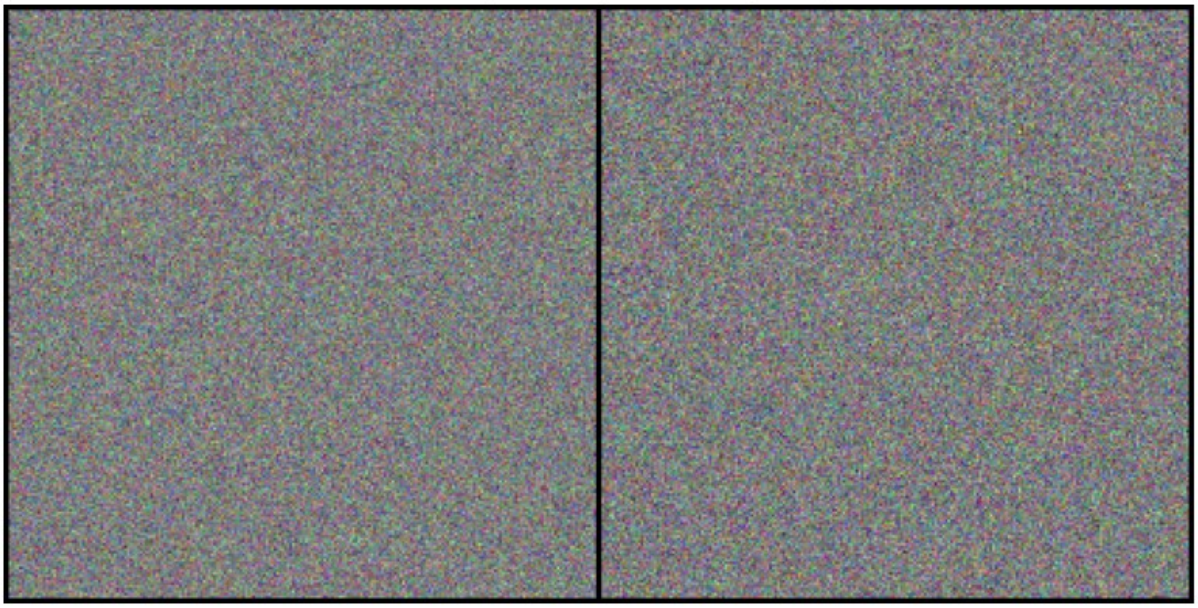
```
Iteration 81/200, SAM Loss: 1.2075139284133911
Iteration 82/200, SAM Loss: 0.436687707901001
Iteration 83/200, SAM Loss: 0.5907412767410278
Iteration 84/200, SAM Loss: 0.12464792281389236
Iteration 85/200, SAM Loss: 0.5070674419403076
```

Visualized Synthetic (Condensed) Images



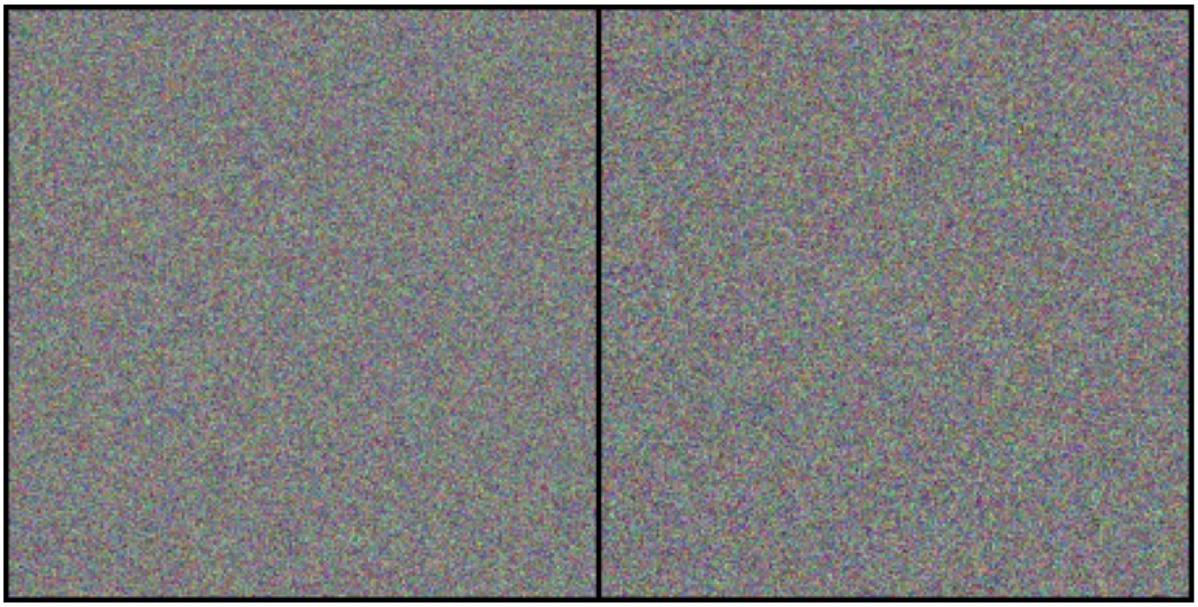
```
Iteration 86/200, SAM Loss: 0.2898673415184021
Iteration 87/200, SAM Loss: 0.31589192152023315
Iteration 88/200, SAM Loss: 0.20805145800113678
Iteration 89/200, SAM Loss: 0.27298516035079956
Iteration 90/200, SAM Loss: 0.2989266514778137
```

Visualized Synthetic (Condensed) Images



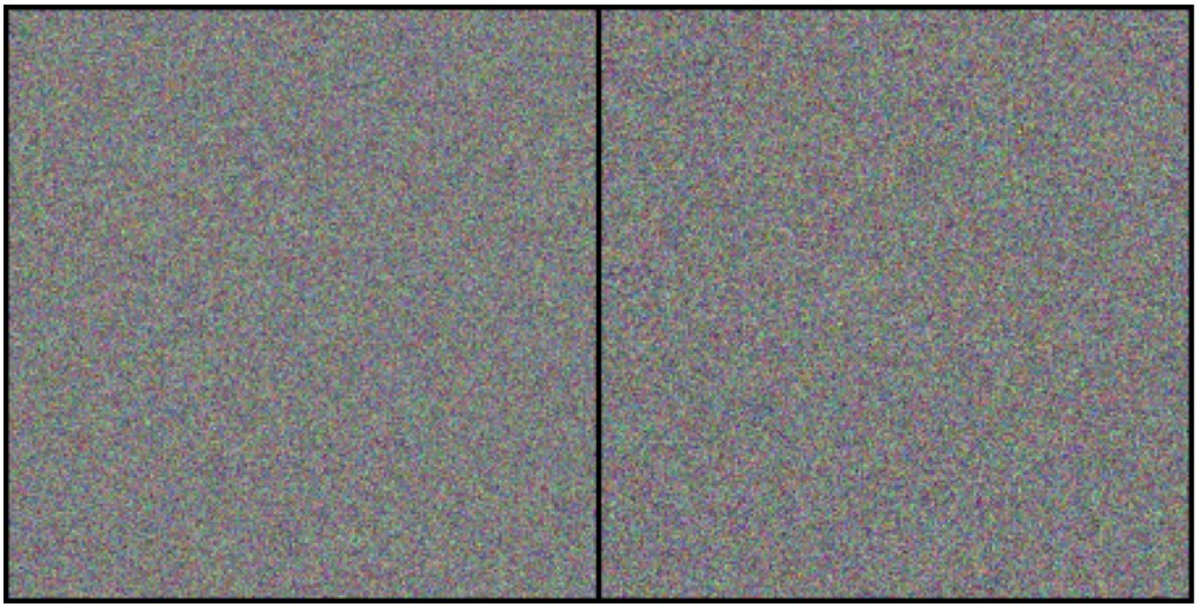
```
Iteration 91/200, SAM Loss: 0.20211602747440338
Iteration 92/200, SAM Loss: 1.1478629112243652
Iteration 93/200, SAM Loss: 0.5389487743377686
Iteration 94/200, SAM Loss: 0.7608807682991028
Iteration 95/200, SAM Loss: 0.4418078660964966
```

Visualized Synthetic (Condensed) Images



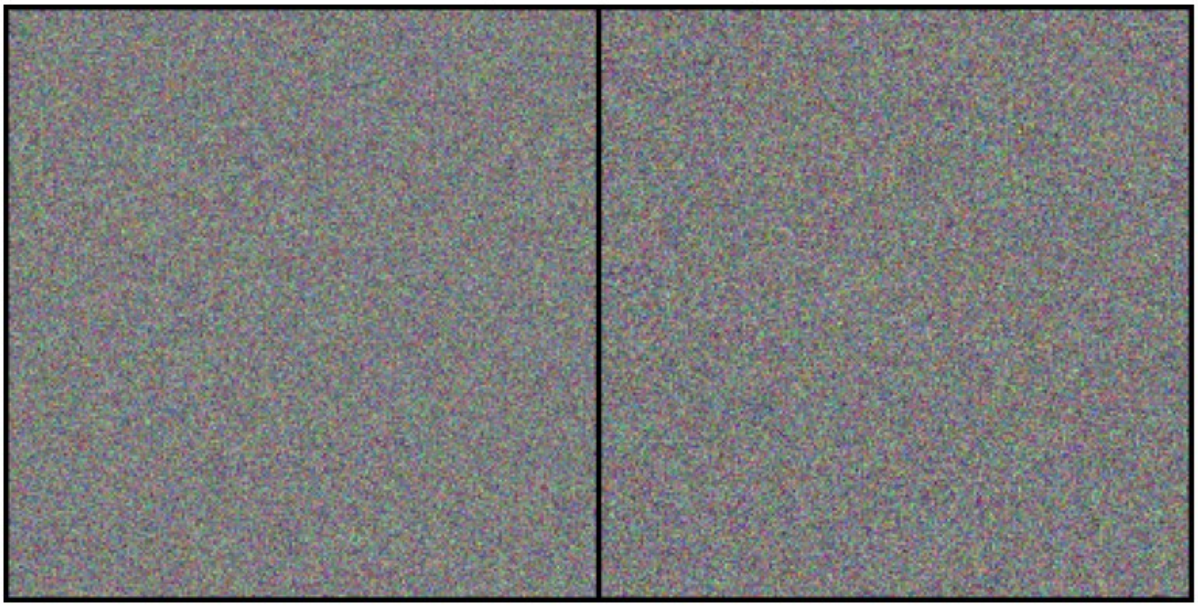
```
Iteration 96/200, SAM Loss: 0.5567556619644165
Iteration 97/200, SAM Loss: 0.08664382249116898
Iteration 98/200, SAM Loss: 0.9840946197509766
Iteration 99/200, SAM Loss: 0.9028469324111938
Iteration 100/200, SAM Loss: 1.974325180053711
```

Visualized Synthetic (Condensed) Images



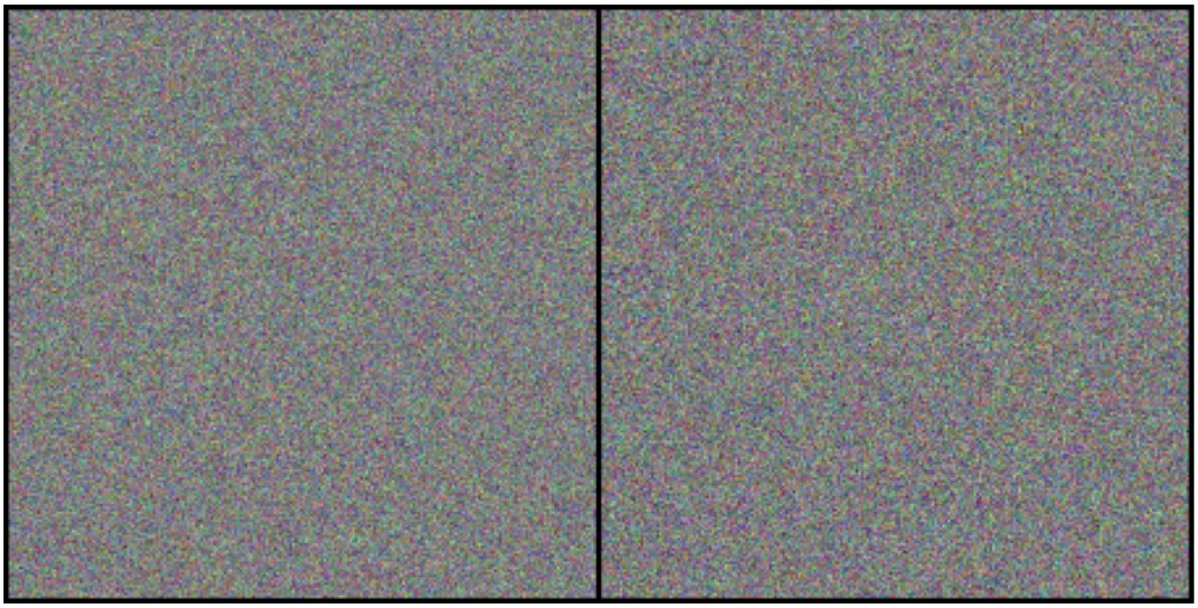
```
Iteration 101/200, SAM Loss: 0.05050665885210037
Iteration 102/200, SAM Loss: 0.01140511967241764
Iteration 103/200, SAM Loss: 0.06415707617998123
Iteration 104/200, SAM Loss: 0.32100796699523926
Iteration 105/200, SAM Loss: 0.19438536465168
```

Visualized Synthetic (Condensed) Images

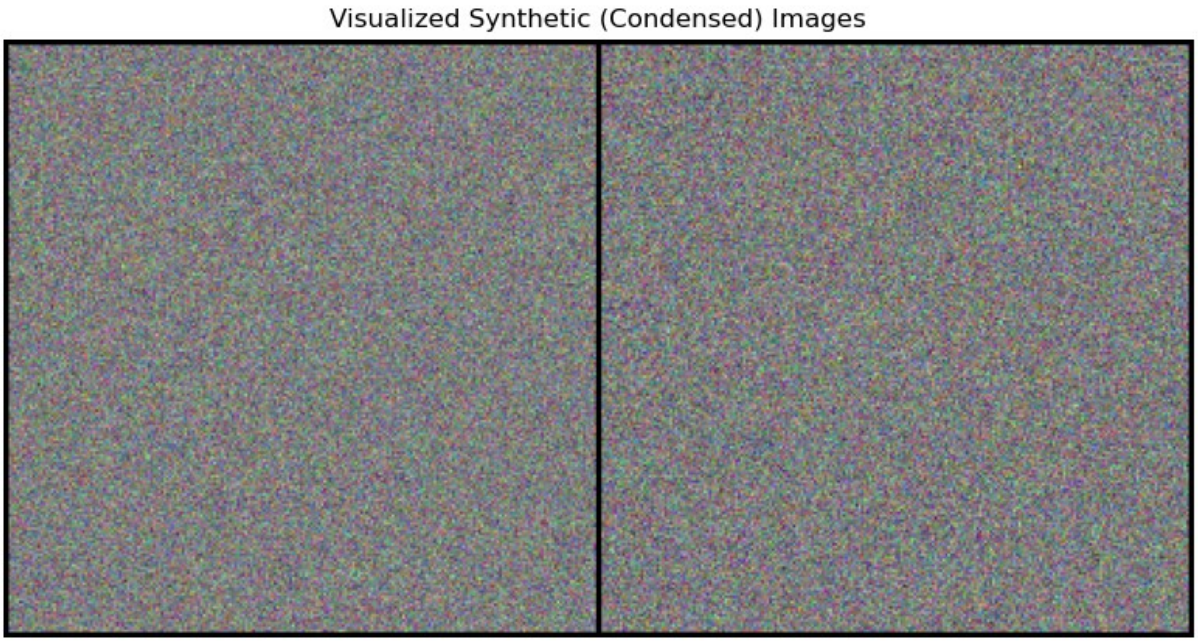


```
Iteration 106/200, SAM Loss: 0.4670848846435547
Iteration 107/200, SAM Loss: 0.36153435707092285
Iteration 108/200, SAM Loss: 0.08046186715364456
Iteration 109/200, SAM Loss: 0.7123156189918518
Iteration 110/200, SAM Loss: 1.3961918354034424
```

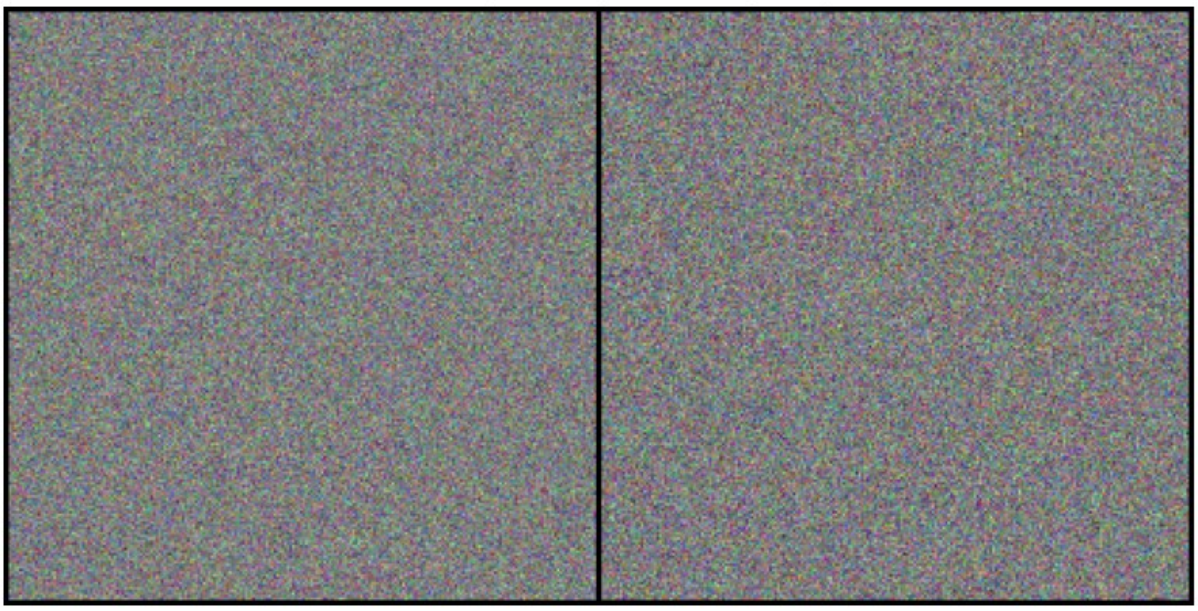
Visualized Synthetic (Condensed) Images



```
Iteration 111/200, SAM Loss: 0.08320791274309158
Iteration 112/200, SAM Loss: 0.6007540225982666
Iteration 113/200, SAM Loss: 0.181448832154274
Iteration 114/200, SAM Loss: 0.1727975755929947
Iteration 115/200, SAM Loss: 1.3906774520874023
```

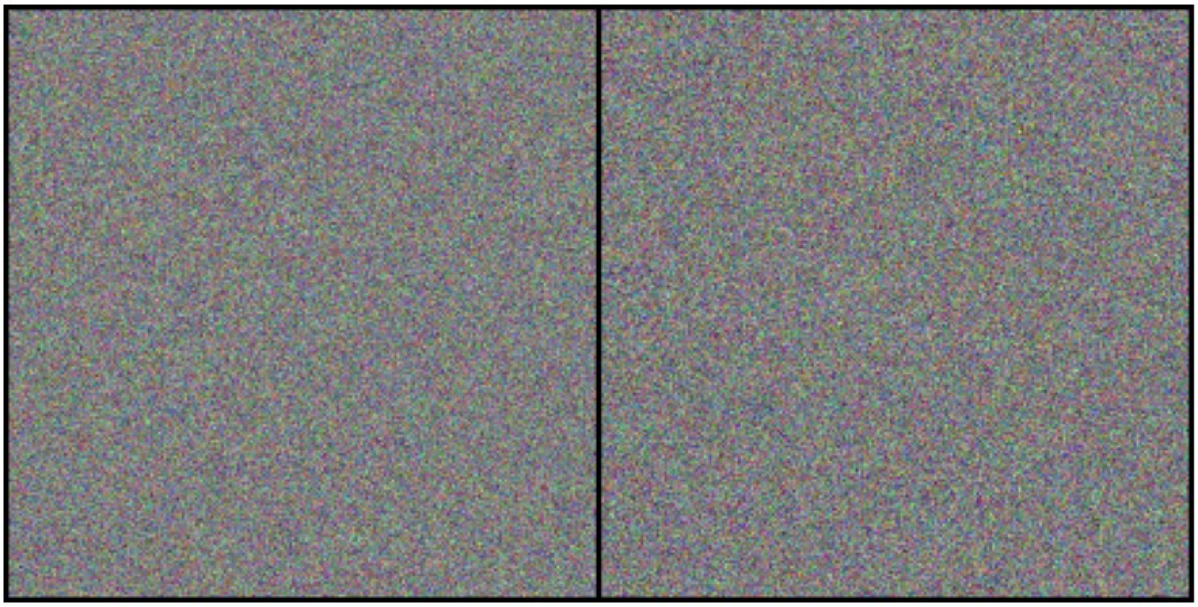



```
Iteration 116/200, SAM Loss: 0.14346767961978912
Iteration 117/200, SAM Loss: 0.671867847442627
Iteration 118/200, SAM Loss: 0.40972310304641724
Iteration 119/200, SAM Loss: 0.06658560782670975
Iteration 120/200, SAM Loss: 0.42051035165786743
```



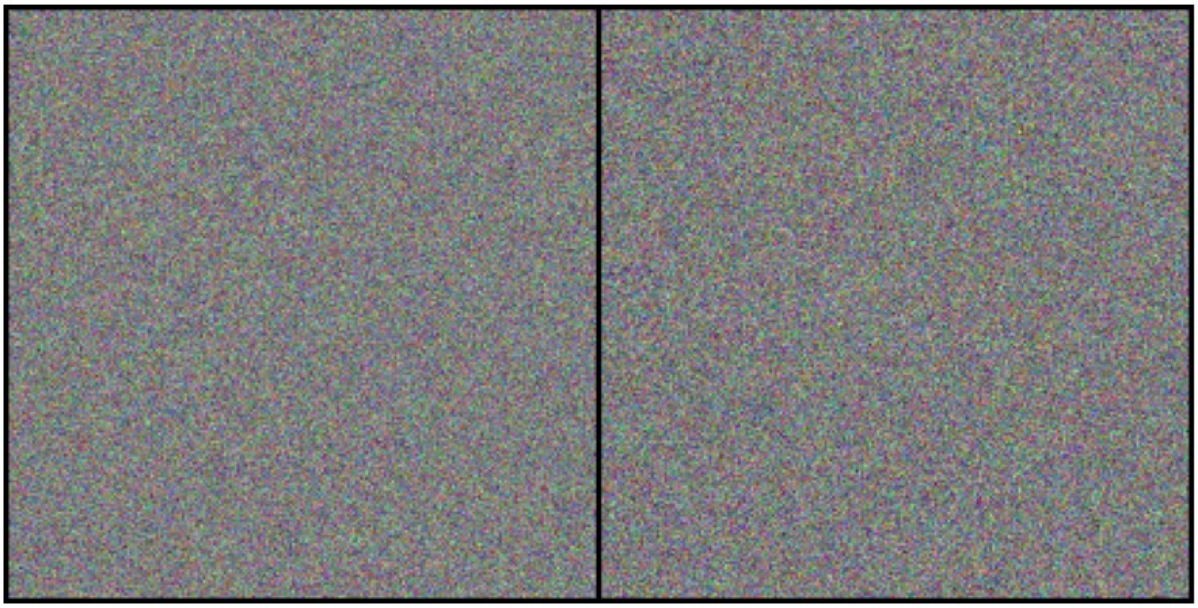

```
Iteration 121/200, SAM Loss: 0.7531847953796387
Iteration 122/200, SAM Loss: 0.40031421184539795
Iteration 123/200, SAM Loss: 0.3954012989997864
Iteration 124/200, SAM Loss: 0.11514837294816971
Iteration 125/200, SAM Loss: 0.9670853614807129
```

Visualized Synthetic (Condensed) Images



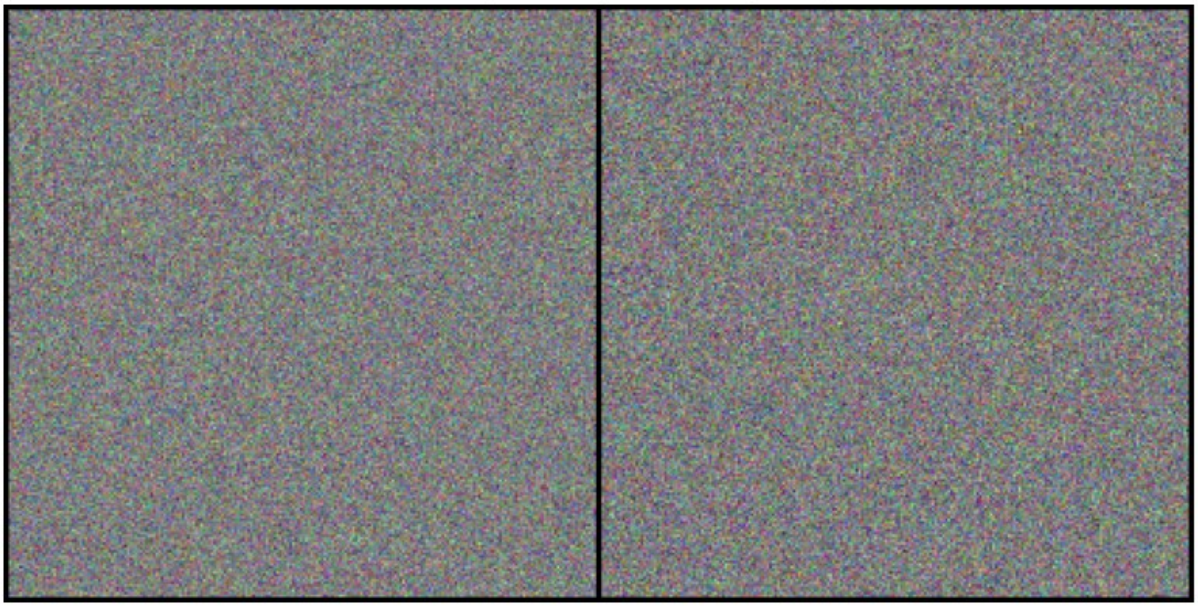
```
Iteration 126/200, SAM Loss: 0.48582708835601807
Iteration 127/200, SAM Loss: 0.5928347110748291
Iteration 128/200, SAM Loss: 0.11420298367738724
Iteration 129/200, SAM Loss: 0.638157308101654
Iteration 130/200, SAM Loss: 0.5357222557067871
```

Visualized Synthetic (Condensed) Images



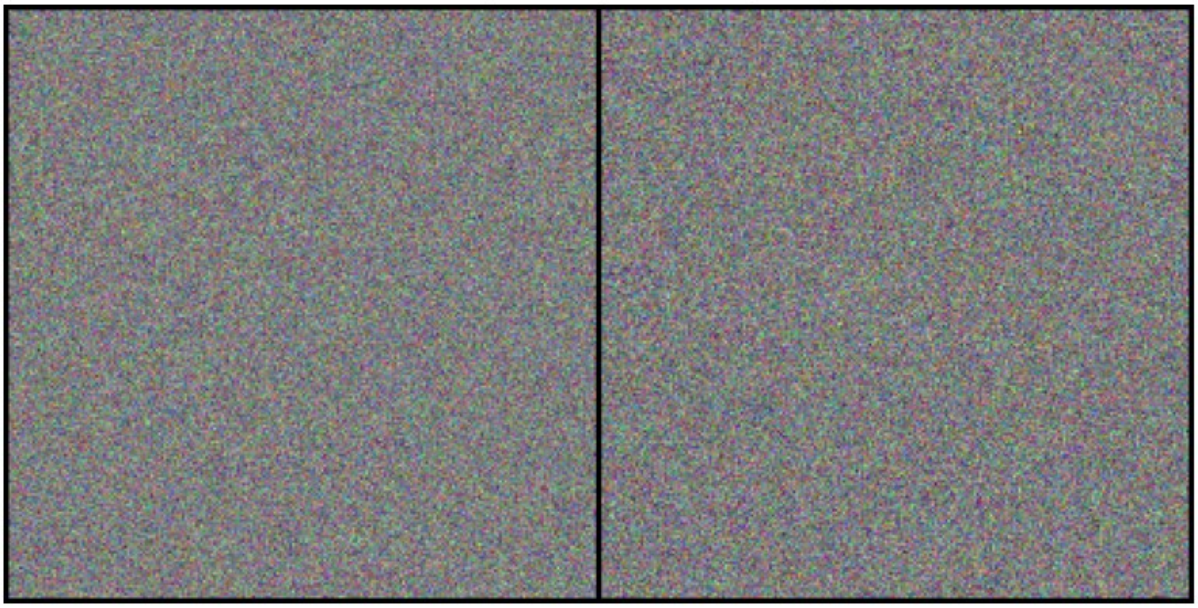
```
Iteration 131/200, SAM Loss: 1.1997615098953247
Iteration 132/200, SAM Loss: 0.2857159376144409
Iteration 133/200, SAM Loss: 0.12626107037067413
Iteration 134/200, SAM Loss: 0.10883898288011551
Iteration 135/200, SAM Loss: 0.07292408496141434
```

Visualized Synthetic (Condensed) Images



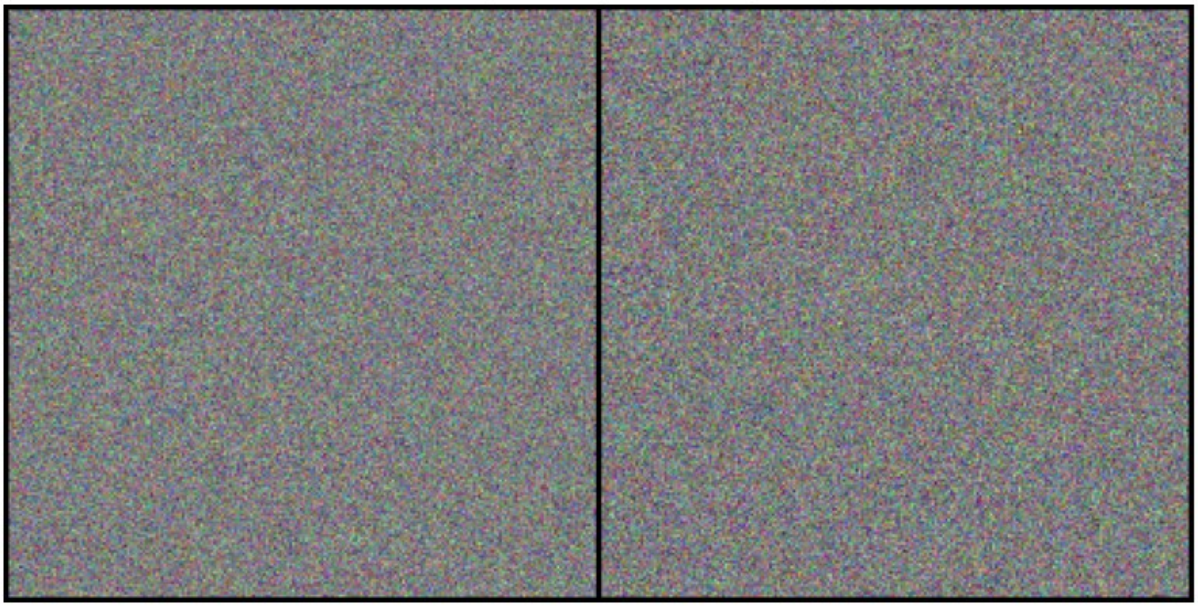
```
Iteration 136/200, SAM Loss: 0.047876425087451935
Iteration 137/200, SAM Loss: 0.08579916507005692
Iteration 138/200, SAM Loss: 0.06914681941270828
Iteration 139/200, SAM Loss: 0.29878270626068115
Iteration 140/200, SAM Loss: 0.8300502300262451
```

Visualized Synthetic (Condensed) Images



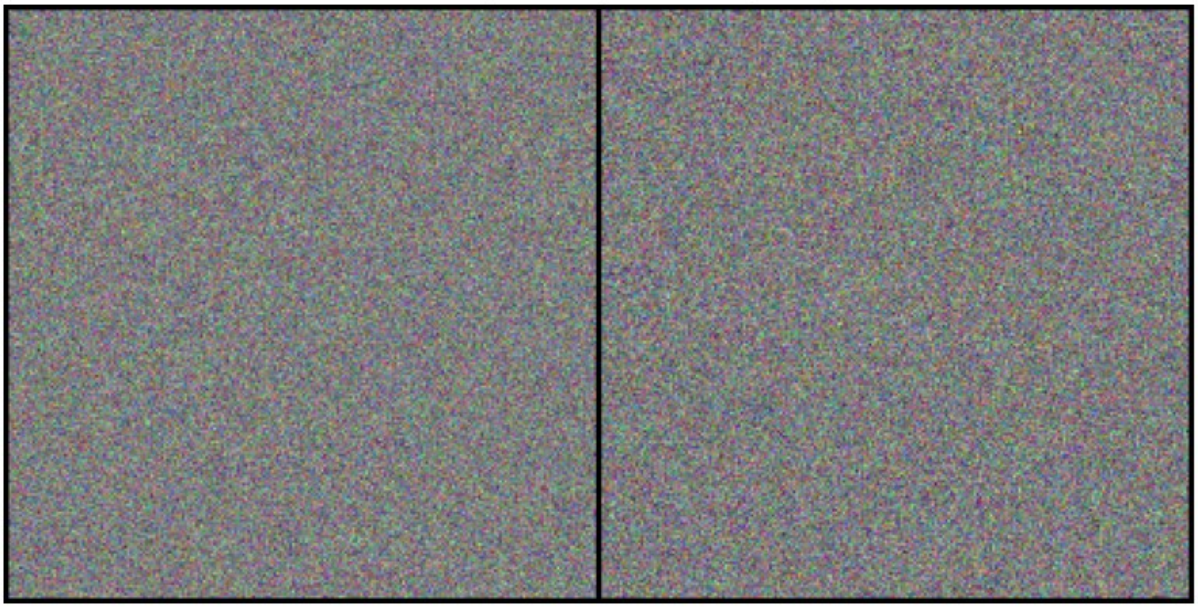
```
Iteration 141/200, SAM Loss: 0.675663948059082
Iteration 142/200, SAM Loss: 0.6147029399871826
Iteration 143/200, SAM Loss: 1.3402807712554932
Iteration 144/200, SAM Loss: 1.086564302444458
Iteration 145/200, SAM Loss: 0.273964524269104
```

Visualized Synthetic (Condensed) Images



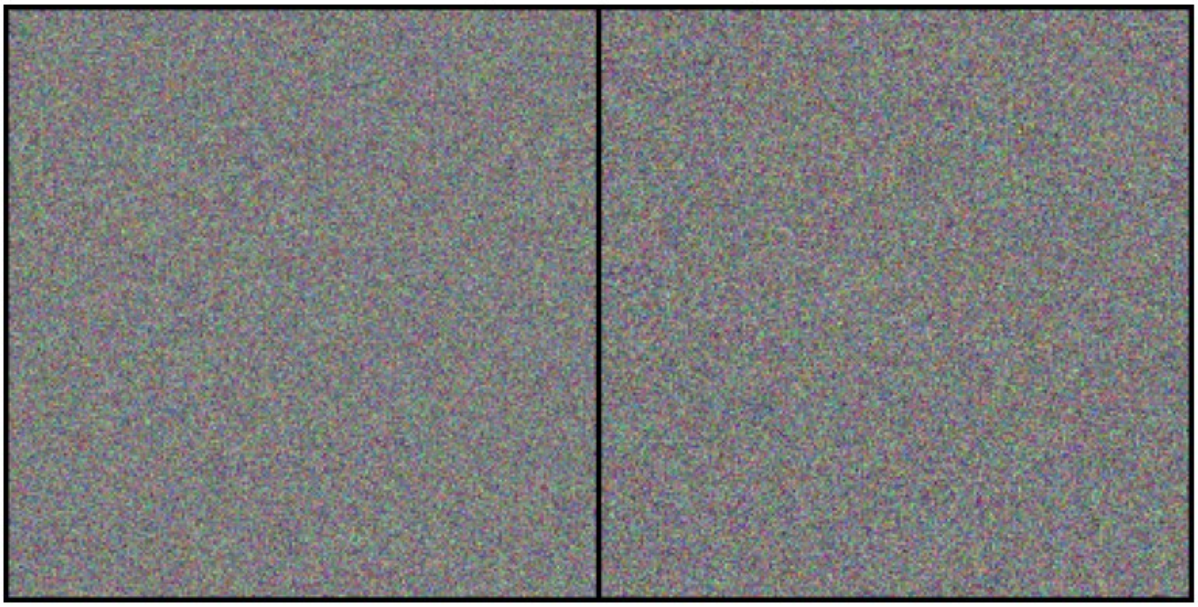
```
Iteration 146/200, SAM Loss: 0.11010397225618362
Iteration 147/200, SAM Loss: 0.13847823441028595
Iteration 148/200, SAM Loss: 1.2674968242645264
Iteration 149/200, SAM Loss: 1.735368251800537
Iteration 150/200, SAM Loss: 0.43955087661743164
```

Visualized Synthetic (Condensed) Images



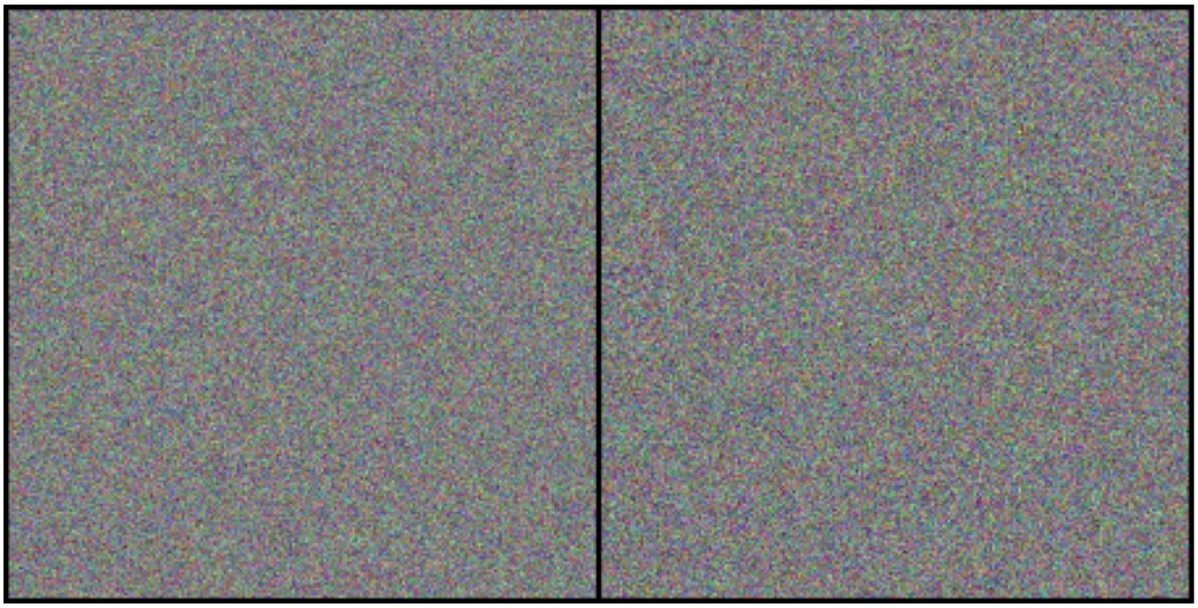
```
Iteration 151/200, SAM Loss: 0.5145469903945923
Iteration 152/200, SAM Loss: 0.587695300579071
Iteration 153/200, SAM Loss: 0.819360613822937
Iteration 154/200, SAM Loss: 0.15936626493930817
Iteration 155/200, SAM Loss: 0.2758117914199829
```

Visualized Synthetic (Condensed) Images



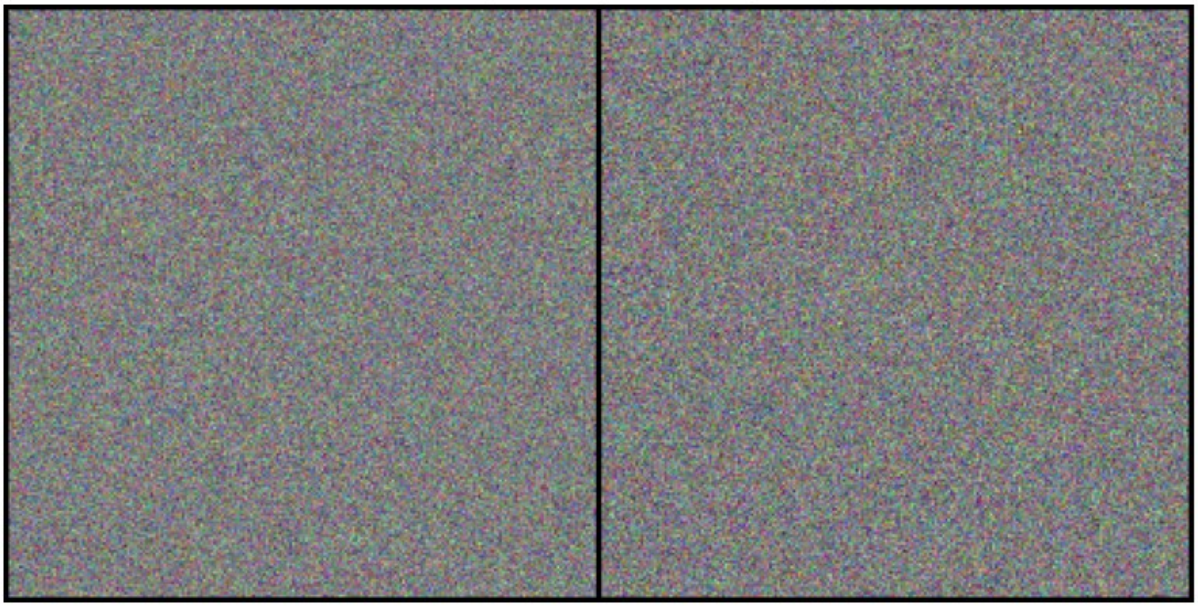
```
Iteration 156/200, SAM Loss: 0.8227286338806152
Iteration 157/200, SAM Loss: 0.4915117025375366
Iteration 158/200, SAM Loss: 0.09718889743089676
Iteration 159/200, SAM Loss: 0.8133030533790588
Iteration 160/200, SAM Loss: 1.123751163482666
```

Visualized Synthetic (Condensed) Images



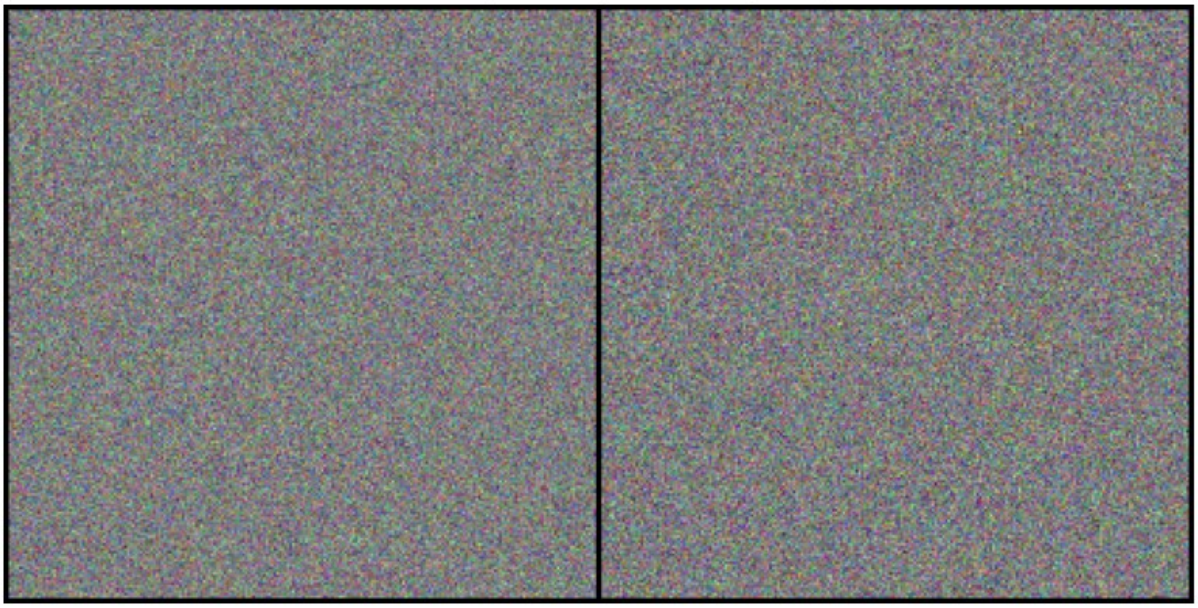
```
Iteration 161/200, SAM Loss: 0.9222137928009033
Iteration 162/200, SAM Loss: 0.11858690530061722
Iteration 163/200, SAM Loss: 0.26395732164382935
Iteration 164/200, SAM Loss: 0.6197378635406494
Iteration 165/200, SAM Loss: 1.2812974452972412
```

Visualized Synthetic (Condensed) Images



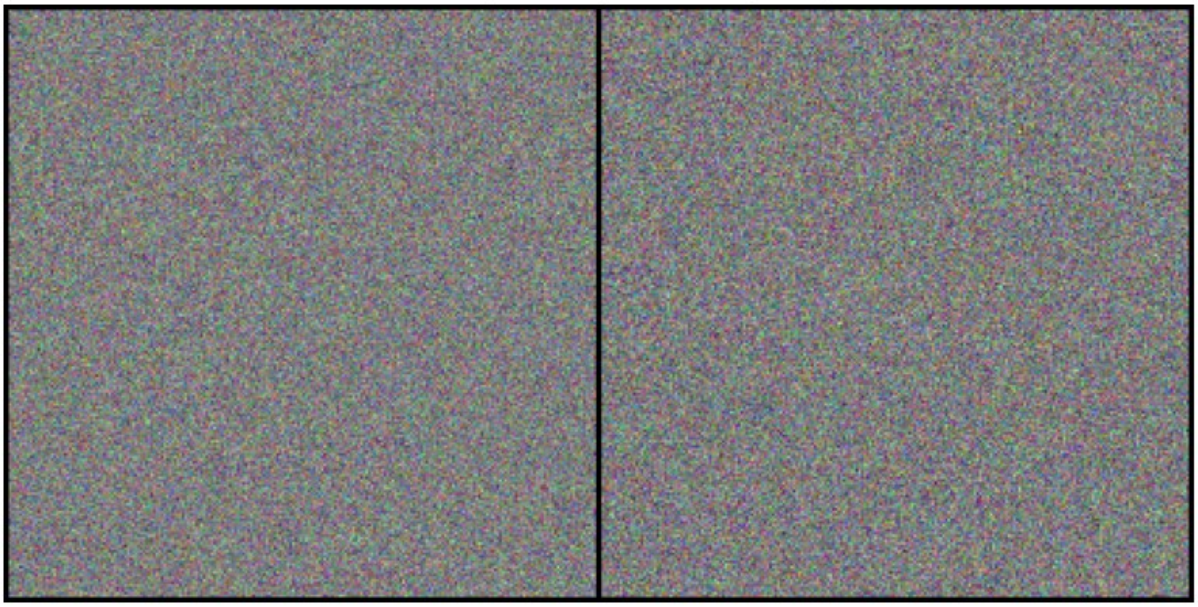
```
Iteration 166/200, SAM Loss: 0.06185764819383621
Iteration 167/200, SAM Loss: 1.125948429107666
Iteration 168/200, SAM Loss: 0.09742606431245804
Iteration 169/200, SAM Loss: 0.24399800598621368
Iteration 170/200, SAM Loss: 0.413662850856781
```

Visualized Synthetic (Condensed) Images



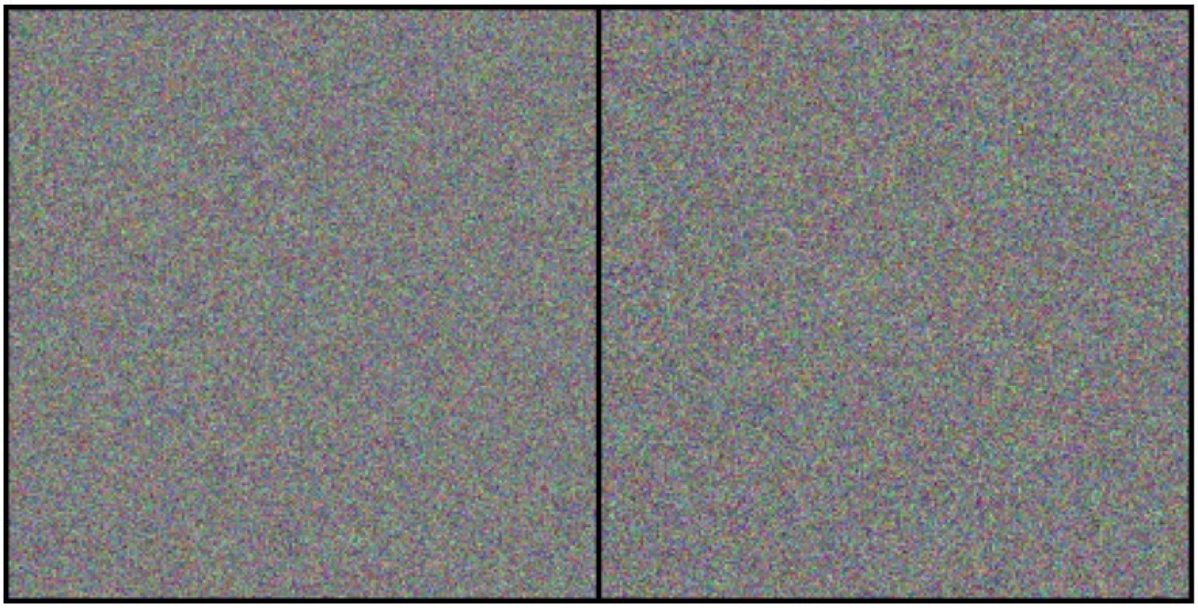
```
Iteration 171/200, SAM Loss: 0.31732791662216187
Iteration 172/200, SAM Loss: 0.30036675930023193
Iteration 173/200, SAM Loss: 0.047031112015247345
Iteration 174/200, SAM Loss: 0.17348714172840118
Iteration 175/200, SAM Loss: 0.3493116497993469
```

Visualized Synthetic (Condensed) Images



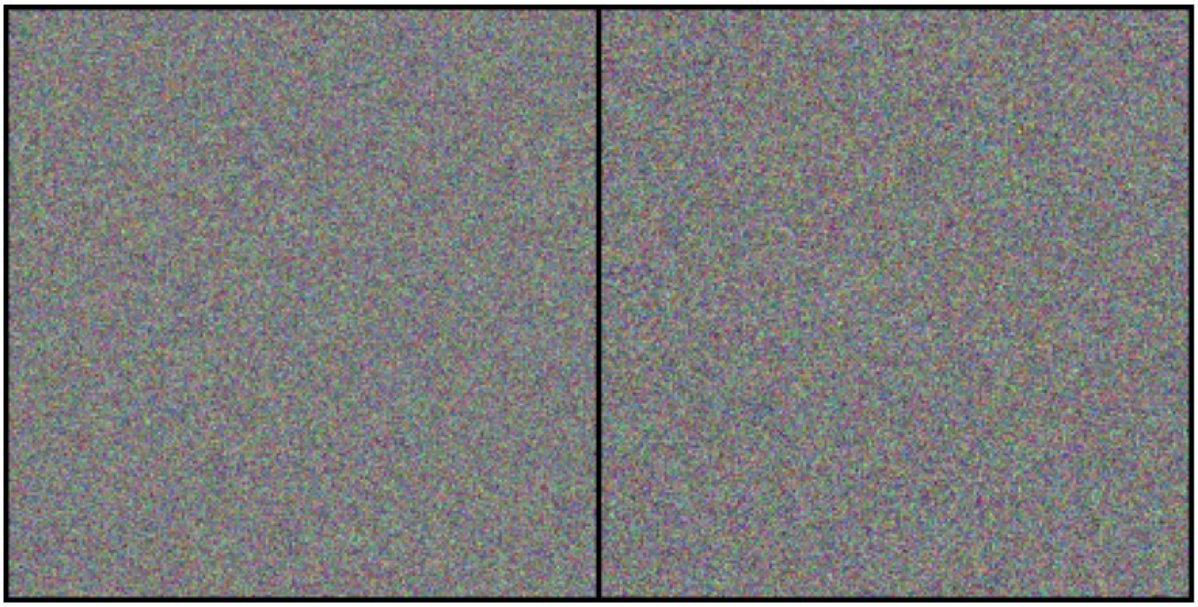
```
Iteration 176/200, SAM Loss: 0.3099749684333801
Iteration 177/200, SAM Loss: 0.7088462114334106
Iteration 178/200, SAM Loss: 1.9418553113937378
Iteration 179/200, SAM Loss: 0.19808097183704376
Iteration 180/200, SAM Loss: 1.5228946208953857
```

Visualized Synthetic (Condensed) Images



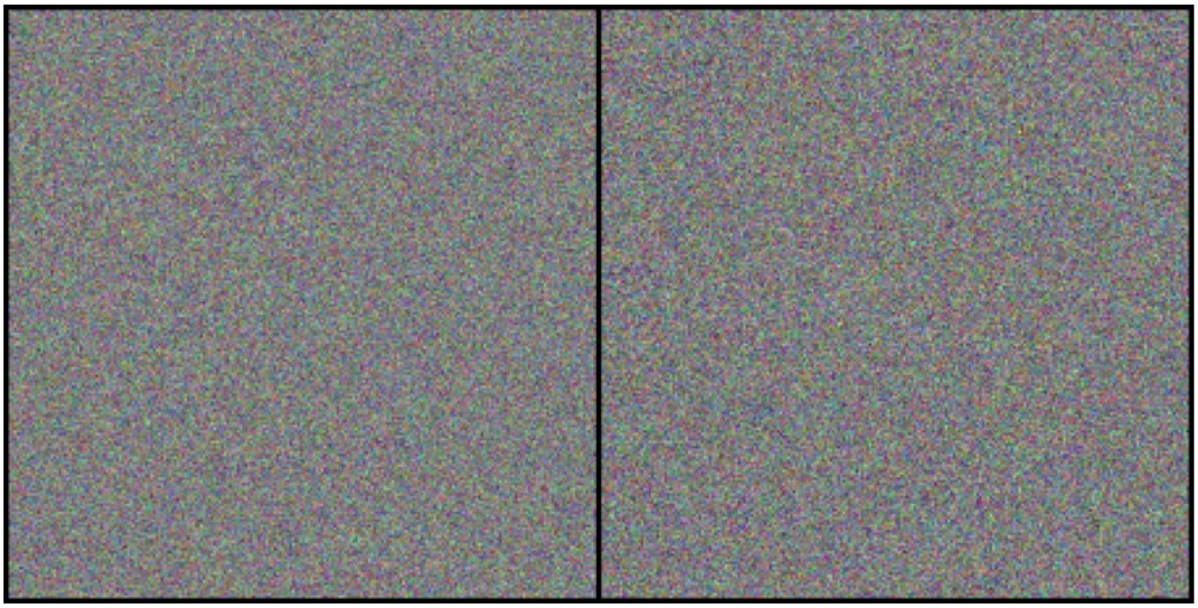
```
Iteration 181/200, SAM Loss: 0.12810786068439484
Iteration 182/200, SAM Loss: 1.5827234983444214
Iteration 183/200, SAM Loss: 0.6552492380142212
Iteration 184/200, SAM Loss: 0.82565838098526
Iteration 185/200, SAM Loss: 0.298545241355896
```

Visualized Synthetic (Condensed) Images

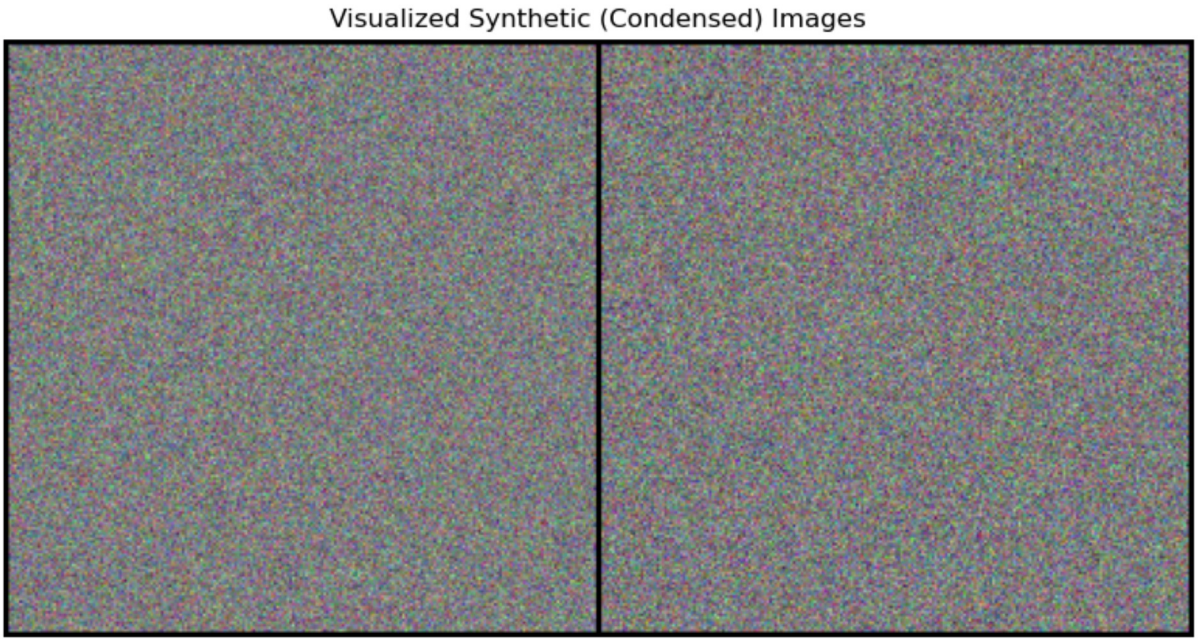


```
Iteration 186/200, SAM Loss: 0.17319132387638092
Iteration 187/200, SAM Loss: 0.07210607081651688
Iteration 188/200, SAM Loss: 0.11654055863618851
Iteration 189/200, SAM Loss: 0.6630160808563232
Iteration 190/200, SAM Loss: 0.92356276512146
```

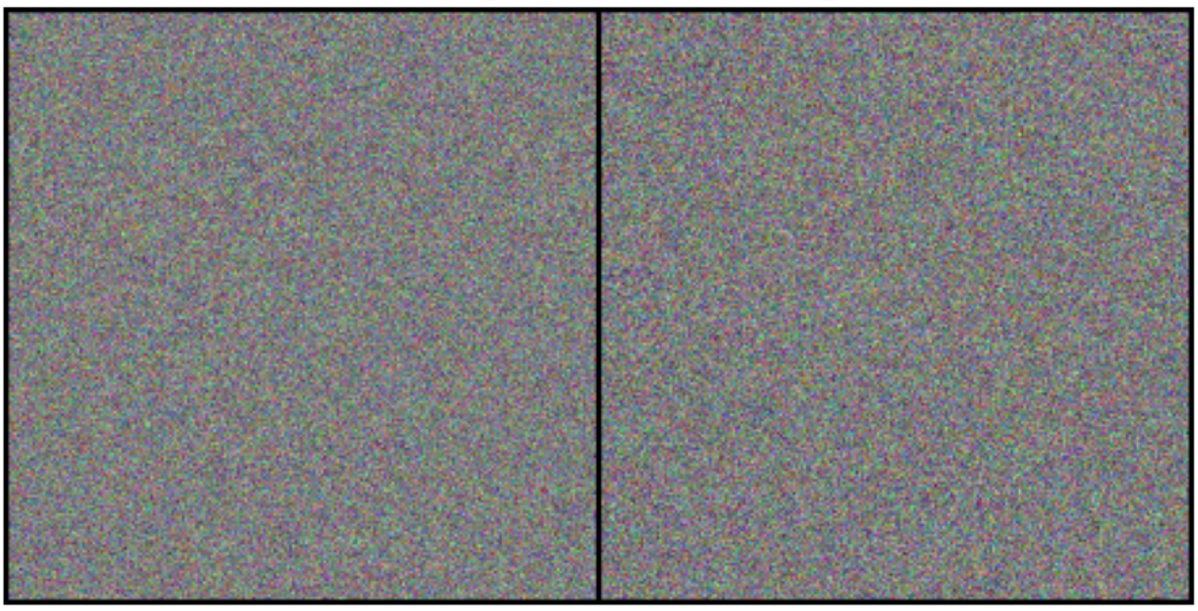
Visualized Synthetic (Condensed) Images



```
Iteration 191/200, SAM Loss: 0.25965720415115356
Iteration 192/200, SAM Loss: 0.29672902822494507
Iteration 193/200, SAM Loss: 1.6893587112426758
Iteration 194/200, SAM Loss: 0.12827540934085846
Iteration 195/200, SAM Loss: 0.6120777726173401
```




```
Iteration 196/200, SAM Loss: 0.08390320092439651
Iteration 197/200, SAM Loss: 0.09214503318071365
Iteration 198/200, SAM Loss: 0.5700350999832153
Iteration 199/200, SAM Loss: 0.02359355427324772
Iteration 200/200, SAM Loss: 0.3157395124435425
```



In [50]:
```python
# Task 1 - Question 2 - Part e

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

num_classes = 2
num_samples_per_class = 50
synthetic_data_expanded = synthetic_data.repeat_interleave(num_samples_per_class, d

synthetic_labels = torch.cat([torch.zeros(num_samples_per_class, dtype=torch.long),
```

```
synthetic_dataset = TensorDataset(synthetic_data_expanded, synthetic_labels)
synthetic_loader = DataLoader(synthetic_dataset, batch_size=16, shuffle=True)

model_name = 'ConvNetD3'
model_synthetic = get_network(model_name, channel=3, num_classes=num_classes, im_si

optimizer_synthetic = optim.SGD(model_synthetic.parameters(), lr=0.01)
criterion = torch.nn.CrossEntropyLoss()

epochs = 10
start_time = time.time()

print("Training on Synthetic Dataset...")
for epoch in range(epochs):
    model_synthetic.train()
    epoch_loss, correct, total = 0, 0, 0
    for images, labels in synthetic_loader:
        images, labels = images.to(device), labels.to(device)

        optimizer_synthetic.zero_grad()
        outputs = model_synthetic(images)
        loss = criterion(outputs, labels)

        loss.backward()
        optimizer_synthetic.step()

        epoch_loss += loss.item() * labels.size(0)
        _, predicted = outputs.max(1)
        total += labels.size(0)
        correct += predicted.eq(labels).sum().item()

    accuracy = 100 * correct / total
    print(f"Epoch [{epoch+1}/{epochs}], Loss: {epoch_loss/total:.4f}, Accuracy: {ac

training_time_synthetic = time.time() - start_time
print(f"Training Time on Synthetic Dataset: {training_time_synthetic:.2f} seconds")

test_loader = DataLoader(test_dataset, batch_size=16, shuffle=False)  # Assuming `t

def evaluate_model(model, test_loader):
    model.eval()
    correct, total = 0, 0
    with torch.no_grad():
        for images, labels in test_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, predicted = outputs.max(1)
            total += labels.size(0)
            correct += predicted.eq(labels).sum().item()
    accuracy = 100 * correct / total
    return accuracy

test_accuracy_synthetic = evaluate_model(model_synthetic, test_loader)
print(f"Test Accuracy on Real Data (trained on synthetic data): {test_accuracy_synt
```

```
Using device: cpu
Training on Synthetic Dataset...
Epoch [1/10], Loss: 0.1136, Accuracy: 93.00%
Epoch [2/10], Loss: 0.0000, Accuracy: 100.00%
Epoch [3/10], Loss: 0.0000, Accuracy: 100.00%
Epoch [4/10], Loss: 0.0000, Accuracy: 100.00%
Epoch [5/10], Loss: 0.0000, Accuracy: 100.00%
Epoch [6/10], Loss: 0.0000, Accuracy: 100.00%
Epoch [7/10], Loss: 0.0000, Accuracy: 100.00%
Epoch [8/10], Loss: 0.0000, Accuracy: 100.00%
Epoch [9/10], Loss: 0.0000, Accuracy: 100.00%
Epoch [10/10], Loss: 0.0000, Accuracy: 100.00%
Training Time on Synthetic Dataset: 3764.23 seconds
Test Accuracy on Real Data (trained on synthetic data): 29.89%
```

In [41]:

```python
# Task 1 - Question 3

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

num_samples_per_class = 50
synthetic_data_expanded = synthetic_data.repeat_interleave(num_samples_per_class, d

synthetic_labels = torch.cat([
    torch.zeros(num_samples_per_class, dtype=torch.long),
    torch.ones(num_samples_per_class, dtype=torch.long)
]).to(device)

synthetic_dataset = TensorDataset(synthetic_data_expanded, synthetic_labels)
synthetic_loader = DataLoader(synthetic_dataset, batch_size=16, shuffle=True)

num_classes = 2
channel = 3
im_size = (224, 224)

convnet_model = ConvNet(channel=channel, num_classes=num_classes, net_width=64, net

optimizer = optim.SGD(convnet_model.parameters(), lr=0.01, momentum=0.9)
criterion = torch.nn.CrossEntropyLoss()

epochs = 10
print("Training ConvNetW64 on Condensed Synthetic Dataset...")

for epoch in range(epochs):
    convnet_model.train()
    epoch_loss, correct, total = 0, 0, 0

    for images, labels in synthetic_loader:
        images, labels = images.to(device), labels.to(device)

        optimizer.zero_grad()
        outputs = convnet_model(images)
        loss = criterion(outputs, labels)

        loss.backward()
        optimizer.step()
```

```
        epoch_loss += loss.item() * labels.size(0)
        _, predicted = outputs.max(1)
        total += labels.size(0)
        correct += predicted.eq(labels).sum().item()

    accuracy = 100 * correct / total
    print(f"Epoch [{epoch + 1}/{epochs}], Loss: {epoch_loss/total:.4f}, Accuracy: {

def evaluate_model(model, test_loader):
    model.eval()
    correct, total = 0, 0
    with torch.no_grad():
        for images, labels in test_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, predicted = outputs.max(1)
            total += labels.size(0)
            correct += predicted.eq(labels).sum().item()
    accuracy = 100 * correct / total
    return accuracy

test_accuracy = evaluate_model(convnet_model, test_loader)
print(f"Test Accuracy on Real Data (trained on condensed data with ConvNetW64): {te
```

```
Using device: cpu
Training ConvNetW64 on Condensed Synthetic Dataset...
Epoch [1/10], Loss: 2.4998, Accuracy: 71.00%
Epoch [2/10], Loss: 0.0000, Accuracy: 100.00%
Epoch [3/10], Loss: 0.0000, Accuracy: 100.00%
Epoch [4/10], Loss: 0.0000, Accuracy: 100.00%
Epoch [5/10], Loss: 0.0000, Accuracy: 100.00%
Epoch [6/10], Loss: 0.0000, Accuracy: 100.00%
Epoch [7/10], Loss: 0.0000, Accuracy: 100.00%
Epoch [8/10], Loss: 0.0000, Accuracy: 100.00%
Epoch [9/10], Loss: 0.0000, Accuracy: 100.00%
Epoch [10/10], Loss: 0.0000, Accuracy: 100.00%
Test Accuracy on Real Data (trained on condensed data with ConvNetW64): 70.11%
```

In [42]:

```
# Task 1 - Question 4
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

num_phases = 3
phase_datasets = random_split(mhist_dataset, [len(mhist_dataset) // num_phases] * n

num_classes = 2
print(f"synthetic_data shape: {synthetic_data.shape}")

if synthetic_data.shape[0] != 100:
    synthetic_data = torch.randn(100, 3, 224, 224, requires_grad=True, device=devic

print(f"synthetic_labels shape: {synthetic_labels.shape}")

synthetic_labels = torch.cat([torch.zeros(50, dtype=torch.long), torch.ones(50, dty
```

```
print(f"New synthetic_data shape: {synthetic_data.shape}")
print(f"New synthetic_labels shape: {synthetic_labels.shape}")

synthetic_dataset = TensorDataset(synthetic_data, synthetic_labels)
synthetic_loader = DataLoader(synthetic_dataset, batch_size=16, shuffle=True)

model = ConvNet(channel=3, num_classes=num_classes, net_width=64, net_depth=3, net_
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
criterion = torch.nn.CrossEntropyLoss()

epochs = 5
for phase, phase_data in enumerate(phase_datasets, start=1):
    phase_loader = DataLoader(phase_data, batch_size=16, shuffle=True)

    phase_images = []
    phase_labels = []
    for image, label in phase_data:
        phase_images.append(image)
        phase_labels.append(label)

    phase_images = torch.stack(phase_images)
    phase_labels = torch.tensor(phase_labels, dtype=torch.long)

    phase_dataset_with_replay = ConcatDataset([TensorDataset(phase_images, phase_la
    phase_loader_with_replay = DataLoader(phase_dataset_with_replay, batch_size=16,

    print(f"--- Training Phase {phase} ---")
    for epoch in range(epochs):
        model.train()
        total_loss, correct, total = 0, 0, 0
        for images, labels in phase_loader_with_replay:
            images, labels = images.to(device), labels.to(device)

            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            total_loss += loss.item() * labels.size(0)
            _, predicted = outputs.max(1)
            total += labels.size(0)
            correct += predicted.eq(labels).sum().item()

        accuracy = 100 * correct / total
        print(f"Phase {phase}, Epoch {epoch+1}/{epochs}, Loss: {total_loss/total:.4

    test_accuracy = evaluate_model(model, test_loader)
    print(f"Test Accuracy after Phase {phase}: {test_accuracy:.2f}%")

print("Continual Learning with Synthetic Replay Buffer Complete.")
```

```
Using device: cpu
synthetic_data shape: torch.Size([2, 3, 224, 224])
synthetic_labels shape: torch.Size([100])
New synthetic_data shape: torch.Size([100, 3, 224, 224])
New synthetic_labels shape: torch.Size([100])
--- Training Phase 1 ---
Phase 1, Epoch 1/5, Loss: 5.9463, Accuracy: 62.30%
Phase 1, Epoch 2/5, Loss: 0.6159, Accuracy: 69.45%
Phase 1, Epoch 3/5, Loss: 0.6080, Accuracy: 69.45%
Phase 1, Epoch 4/5, Loss: 0.5957, Accuracy: 69.45%
Phase 1, Epoch 5/5, Loss: 0.5825, Accuracy: 69.58%
Test Accuracy after Phase 1: 70.11%
--- Training Phase 2 ---
Phase 2, Epoch 1/5, Loss: 0.6130, Accuracy: 68.24%
Phase 2, Epoch 2/5, Loss: 0.5979, Accuracy: 68.24%
Phase 2, Epoch 3/5, Loss: 0.5817, Accuracy: 67.88%
Phase 2, Epoch 4/5, Loss: 0.5685, Accuracy: 69.21%
Phase 2, Epoch 5/5, Loss: 0.5318, Accuracy: 74.18%
Test Accuracy after Phase 2: 71.49%
--- Training Phase 3 ---
Phase 3, Epoch 1/5, Loss: 0.5836, Accuracy: 70.91%
Phase 3, Epoch 2/5, Loss: 0.5130, Accuracy: 74.42%
Phase 3, Epoch 3/5, Loss: 0.4549, Accuracy: 78.67%
Phase 3, Epoch 4/5, Loss: 0.3811, Accuracy: 83.27%
Phase 3, Epoch 5/5, Loss: 0.3017, Accuracy: 87.52%
Test Accuracy after Phase 3: 73.56%
Continual Learning with Synthetic Replay Buffer Complete.
```

In [97]:

```python
# Task 2 - Question 2

import torch
import torch.optim as optim
from torch.utils.data import DataLoader, Subset, TensorDataset
from torchvision import transforms
import torch.nn as nn
import numpy as np
from torch.utils.data import Dataset
from PIL import Image
import pandas as pd
import os

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

class MHISTDataset(Dataset):
    def __init__(self, annotations_file, img_dir, transform=None):
        self.img_labels = pd.read_csv(annotations_file)
        self.img_dir = img_dir
        self.transform = transform
        self.label_mapping = {'SSA': 0, 'HP': 1}

    def __len__(self):
        return len(self.img_labels)

    def __getitem__(self, idx):
        img_name = os.path.join(self.img_dir, self.img_labels.iloc[idx, 0])
        image = Image.open(img_name).convert("RGB")
```

```
        label_str = self.img_labels.iloc[idx, 1]
        label = self.label_mapping[label_str]

        if self.transform:
            image = self.transform(image)
        return image, label

transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])

img_dir = "C:/Users/rupin/Downloads/ECE1512_2024F_ProjectA_submission_files/submiss
annotations_file = r"C:\Users\rupin\Downloads\ECE1512_2024F_ProjectA_submission_fil
mhist_dataset = MHISTDataset(annotations_file=annotations_file, img_dir=img_dir, tr

batch_size = 16
real_data_loader = DataLoader(mhist_dataset, batch_size=batch_size, shuffle=True)

def calculate_el2n_scores(model, dataloader, device):
    model.eval()
    el2n_scores = []
    with torch.no_grad():
        for images, labels in dataloader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            probs = torch.softmax(outputs, dim=1)
            correct_probs = probs.gather(1, labels.unsqueeze(1)).squeeze()
            el2n_scores.extend((1 - correct_probs).cpu().numpy())
    return el2n_scores

channel, num_classes = 3, 2
model = ConvNet(channel=3, num_classes=num_classes, net_width=64, net_depth=3, net_
el2n_scores = calculate_el2n_scores(model, real_data_loader, device)

num_samples = 100
sorted_indices = np.argsort(el2n_scores)
selected_indices = np.concatenate([sorted_indices[:num_samples // 2], sorted_indice
distilled_dataset = Subset(mhist_dataset, selected_indices)
distilled_loader = DataLoader(distilled_dataset, batch_size=16, shuffle=True)

def match_loss_with_trajectory(outputs_syn, outputs_real):
    loss = torch.norm(outputs_syn - outputs_real)
    return loss

epochs = 5
learning_rate = 0.01
synthetic_data = torch.randn(num_classes, channel, 224, 224, requires_grad=True, de
optimizer_syn = optim.SGD([synthetic_data], lr=learning_rate)
criterion = nn.CrossEntropyLoss()

for epoch in range(epochs):
    model.train()
    for images, labels in distilled_loader:
        images, labels = images.to(device), labels.to(device)
```

```
        optimizer_syn.zero_grad()

        repeated_synthetic_data = synthetic_data.repeat(images.size(0) // synthetic

        outputs_real = model(images)
        outputs_syn = model(repeated_synthetic_data)

        loss = match_loss_with_trajectory(outputs_syn, outputs_real) + criterion(ou

        loss.backward()
        optimizer_syn.step()

    print(f"Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}")

model.eval()
correct, total = 0, 0
with torch.no_grad():
    for images, labels in distilled_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs, 1)
        correct += (predicted == labels).sum().item()
        total += labels.size(0)

accuracy = 100 * correct / total
print(f"Accuracy on distilled dataset: {accuracy:.2f}%")
```

```
Epoch [1/5], Loss: 1.3374
Epoch [2/5], Loss: 1.5557
Epoch [3/5], Loss: 1.1639
Epoch [4/5], Loss: 1.3493
Epoch [5/5], Loss: 1.2279
Accuracy on distilled dataset: 70.00%
```