UMBC CMSC 475/675 Neural Networks

Homework 2. 100 points

Due: April 11, 2025. 23:59 Eastern

Answer all problems according to the instructions provided.
Submit one `.zip` file named `LastName_FirstName_hw2.zip` containing a single `pdf` named `hw2.pdf` with the answers to both the written and coding sections. The pdf **must** contain typeset or handwritten answers for the written problems and relevant code snippets, outputs and results, and other requested information for the programming tasks. Put all other files (code, data, outputs, etc.) in a single directory named "`code`"

1 Image Synthesis using GANs (40)

Generative Adversarial Networks (GANs) are powerful models designed to generate high-quality images. In this section, we will implement a specific type of GAN, known as a Deep Convolutional GAN (DCGAN), to generate images of grumpy cats from random noise samples. Follow the steps below to build the network and its training loop.

Part 1: Create Grumpy Cat Dataset We will use the provided dataset (attached) as training data. To load the images from the dataset folder, Define the `CustomDataset` as follows:

```
class CustomDataset(Dataset):
    def __init__(self, main_dir, transform=None):
        self.total_imgs = <your code>
        # use the main_dir argument to locate and load your images
        self.transform = transform

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

    def __getitem__(self, idx):
        <your code>
```

In this implementation:

- The `main_dir` specifies the directory that contains the images.
- The `transform` argument allows for any image transformations to be applied (such as normalization or augmentation).

Part 2: Data Augmentation GANs trained on small datasets are prone to overfitting. To address this, data augmentation techniques will be applied during training. You can leverage `torchvision.transforms` to compose these transformations into a transform object.

Here we'll implement a `get_transform` function that supports two transformation modes, allowing us to select the desired strategy. **Design your own 'deluxe' transformation by incorporating techniques such as random cropping, flipping, and normalization.**

```
from torchvision import transforms
from PIL import Image
```

```
def get_transform(mode:str):
    if mode == "simple":
        transform = transforms.Compose([
        transforms.Resize(opts.image_size, Image.BICUBIC),
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
    elif mode == "deluxe":
        transform = transforms.Compose([
            <your-code>
        ])
    else:
        raise NotImplementedError

    return transform
```

The transform object can then be passed to the CustomDataset class when initializing the dataset, ensuring that the images are augmented as they are loaded.

Part 3: Implement the Discriminator of the DCGAN The discriminator in this DCGAN is a convolutional neural network with the architecture shown in Fig. 1 :

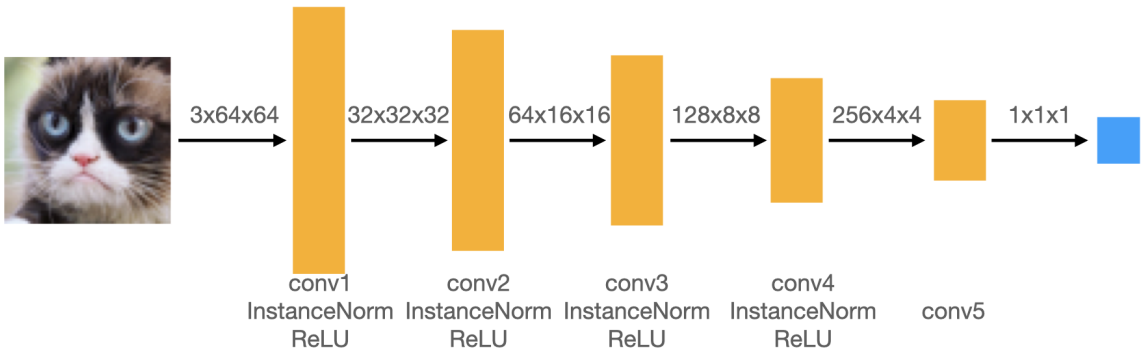

Figure 1: Architecture of the Discriminator in the DCGAN

1. The spatial dimensions of the input volume are downsampled by a factor of 2 in each of the convolutional layers. Given that the kernel size $K = 4$ and the stride $S = 2$, **what should the padding be? Write your answer in your report, and show your work (e.g., the formula you used to derive the padding).**

2. **Implement the Discriminator** class as shown below by filling in the `__init__` and `forward` methods.

```
import torch.nn as nn
Class Discriminator(nn.Module):
    def __init__(self, conv_dim=64):
            super(Discriminator, self).__init__()
            ##########################################
            ##   FILL THIS IN: CREATE ARCHITECTURE   ##
            ##########################################
            # self.conv1 = nn.Conv2d(...)
```

```
            # self.conv2 = nn.Conv2d(...)
            # self.conv3 = nn.Conv2d(...)
            # self.conv4 = nn.Conv2d(...)
            # self.conv5 = nn.Conv2d(...)

            # Remember to add normalization layers where necessary!
            # You can use either BatchNorm2d or InstanceNorm2d
            # Choose the one that helps achieve better performance
            # self.bn1 =
            # self.bn2 =
            # self.bn3 =
            # self.bn4 =

    def forward(self, z):
        """Generates an image given a sample of random noise.
            Input
            -----
                z: BS x noise_size x 1 x 1   -->  16x100x1x1
            Output
            ------
                out: BS x channels x image_width x image_height
        """
        out = F.relu(self.bn1(self.conv1(x)))
        ##########################################
        ##       FILL THIS IN: FORWARD PASS      ##
        ##########################################

        out = self.conv5(out).squeeze()
        return out
```

Part 4: Implement the Generator of the DCGAN Now, we will implement the generator of the DCGAN, which consists of a sequence of transpose convolutional layers that progressively upsample the input noise sample to generate a fake image. The generator in this DCGAN has the following architecture shown in Fig. 2:

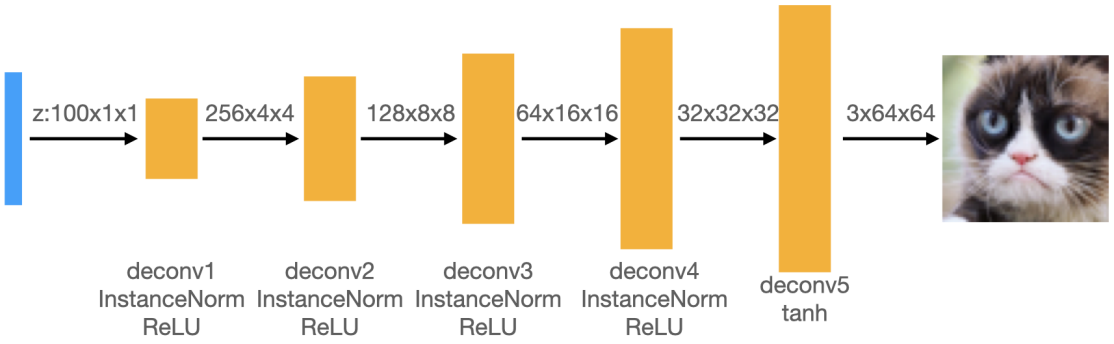

Figure 2: Architecture of the Generator in the DCGAN

Fill in the `__init__` and `forward` methods in the Generator class given below:

```
class Generator(nn.Module):
    def __init__(self, noise_size, conv_dim):
```

```
        super(Generator, self).__init__()

        ##########################################
        ##    FILL THIS IN: CREATE ARCHITECTURE   ##
        ##########################################

        self.deconv1 = nn.ConvTranspose2d(...)
        self.deconv2 = nn.ConvTranspose2d(...)
        self.deconv3 = nn.ConvTranspose2d(...)
        self.deconv4 = nn.ConvTranspose2d(...)
        self.deconv5 = nn.ConvTranspose2d(...)

        #  Don't forget to add normalization layers!
        < your-code>

    def forward(self, z):
        """Generates an image given a sample of random noise.

            Input
            -----
                z: BS x noise_size x 1 x 1   -->  16x100x1x1

            Output
            ------
                out: BS x channels x image_width x image_height  -->  16
    x3x32x32
        """
        ##########################################
        ##    FILL THIS IN: FORWARD PASS    ##
        ##########################################

        out = F.tanh(self.deconv5(out))
        return out
```

Part 5: Training Loop Next, you will implement the training loop for the DCGAN. We train it in exactly the same way as a standard GAN. Complete the code shown below:

```
def training_loop(dataloader):
    # Create generators and discriminators
    < your-code >

    # Create optimizers for the generators and discriminators
    g_optimizer =
    d_optimizer =

    for epoch in range(num_epochs):
        for data in dataloader:
            # Load real images
            < your-code >
            ##########################################
            ##        TRAIN THE DISCRIMINATOR        ##
            ##########################################

            d_optimizer.zero_grad()
```

```
            # 1. Compute the discriminator loss on real images
            D_real_loss =

            # 2. Sample noise
            noise = sample_noise(opts.noise_size)

            # 3. Generate fake images from the noise
            fake_images =

            # 4. Compute the discriminator loss on the fake images
            D_fake_loss =

            D_total_loss =
            if iteration % 2 == 0:
                D_total_loss.backward()
                d_optimizer.step()

            ########################################
            ###          TRAIN THE GENERATOR       ###
            ########################################

            g_optimizer.zero_grad()

            # FILL THIS IN
            # 1. Sample noise
            noise =

            # 2. Generate fake images from the noise
            fake_images =

            # 3. Compute the generator loss
            G_loss =

            G_loss.backward()
            g_optimizer.step()


            #  Print each loss every 200 iterations
            < your-code >
            #save the model every 200 iterations
            < your-code >
```

Train the DCGAN for 100 epochs. **Report the following in your submission**:

1. Screenshots of discriminator and generator training loss for both transformation `–mode=simple/deluxe` – 4 curves in total.

2. Set data augmentation mode to deluxe and then show one of the samples from early in training (e.g., iteration 200) and one of the samples from later in training, and give the iteration number for those samples. **Briefly comment on the quality of the samples, and in what way they improve through training.**

2 Low-Dimensional Representations using Autoencoders (50)

In this section, we will develop and train a convolutional autoencoder with two hidden layers in both the encoder and decoder. We will use the following notations: input x, latent vector z, encoder E s.t. $z = E(x)$, output $\hat{x}$, decoder D, s.t. $\hat{x} = D(z)$. Follow the instructions below to complete this section of the assignment.

Part 1. Loading the Dataset We will use the CIFAR-10 dataset (you can find the dataset in the inbuilt PyTorch datasets). Link: https://pytorch.org/vision/main/datasets.html. Load the dataset and apply transformations to resize and normalize it.

```
T = transforms.Compose([
    transforms.Resize((32, 32)),  # Resize the images to 64x64
    transforms.ToTensor(),        # Convert images to PyTorch tensors
    transforms.Normalize((0.5,), (0.5,))  # Normalize the tensor values to
    [-1, 1]
    ])
trainset = <your-code>
trainLoader = <your-code>
```

Part 2. Defining the Neural Network Architecture We will use a two-layer encoder and two-layer decoder. The output of the encoder should be a $B \times 1 \times 8 \times 8$ latent vector where B is the batchsize. The output of the decoder should be the same as the input image. Write code for the conv layers in both the encoder and decoder with the appropriate chose of input-output channels, kernel size, padding, stride, activations, and pooling/downsampling.

```
# Define the autoencoder architecture
class Autoencoder(nn.Module):
  def __init__(self):
    super(Autoencoder, self).__init__()
    self.encoder = <your code>
    self.decoder = <your-code>

  def forward(self, x):
    x = self.encoder(x)
    x = self.decoder(x)
    return x
model = Autoencoder()
print(model)
summary(model, (3, 32, 32))
# Move the model to GPU if available
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model.to(device)
```

Part 3. Train the Model. Train the model for 50 epochs. Note: you may have to play around with the learning rate and batch size to reach convergence. I recommend using Google Colab. Depending on whether you use CPU or GPU devices, it may take between a few minutes and a few hours to train your model. Plan ahead.

Plot the training loss curve. Show reconstruction on the first batch of the test set after every 10 epochs. You can use the code below to show this visualization:

```
with torch.no_grad():
  for data, _ in test_loader:
    data = data.to(device)
    recon = model(data)
    break
  plt.figure(dpi=250)
  fig, ax = plt.subplots(2, 7, figsize=(15, 4))
  for i in range(7):
    ax[0, i].imshow(data[i].cpu().numpy().transpose((1, 2, 0)))
    ax[1, i].imshow(recon[i].cpu().numpy().transpose((1, 2, 0)))
    ax[0, i].axis('OFF')
    ax[1, i].axis('OFF')
  plt.show()
```

Part 4. Reconstruction Error. Calculate and report the reconstruction error on the test set, for each set of hyperparameters that you tried. Report this in a table.

Part 5. Mean and Variance of Latent Vectors per Class. After training you have an encoder that gives you latent codes for each image. Apply the encoder on the test set to get a set of z vectors. Let's test if images belonging to the same class are clustered together. Let $\mathcal{Z}_c$ be the set of z vectors for class c. Find the mean z_c^{mean} for each class. Find the distance between all z and z_c^{mean} for each class, i.e.

$$dist_c = \frac{1}{n}\sum_{i=1}^{n} ||z_c^i - z_c^{mean}||_2.$$

Report these distances. Which class has the least distance and which class has the highest distance?

Part 6. Unsupervised Classification Next, perform clustering in the latent space using pre-built libraries such as Scikit-learn. Here is an example of K-means clustering using 2D toy data with 3 clusters.

```
from sklearn.cluster import KMeans
import numpy as np
import matplotlib.pyplot as plt

# Sample data
X = np.array([
      [1, 2], [1, 2.1], [1, 1.9], [1.1, 2],
      [1, 4], [1.2, 4], [0.8, 4.3], [1, 3.8],
      [10, 2], [10, 2.4], [10, 1.6], [11, 2.1]
      ])

# Initialize KMeans with the number of clusters
kmeans = KMeans(n_clusters=3, random_state=0)
# Fit the model
kmeans.fit(X)
# Get the cluster labels and centroids
labels = kmeans.labels_
centroids = kmeans.cluster_centers_
print(f"Cluster Labels: {labels}")
print(f"Centroids: {centroids}")

```

```
# Visualize the clusters
for i in range(3):  # Number of clusters
    cluster_points = X[labels == i]
    plt.scatter(cluster_points[:, 0], cluster_points[:, 1], label=f"
    Cluster {i}")

# Plot centroids
plt.scatter(centroids[:, 0], centroids[:, 1], c='red', marker='x', label='
    Centroids')
plt.legend()
plt.show()
```

You can start with this code and modify it for our autoencoder latent space with 10 clusters (because we have 10 classes). Once clustering is complete, show the class distribution of each cluster (i.e. what percentage of each CIFAR-10 class is present in each cluster).

3 Graduate Section: VQ-VAE (10 points)

This part is required for the grad section and optional (extra credit) for the undergrad section.

Read the paper on Vector-Quantized Variational Autoencoder (VQ-VAE) by van der Oord et al [1]. Link: https://arxiv.org/pdf/1711.00937. In class we discussed the idea of Variational Autoencoders (VAE). Building on that knowledge, and using the same notation we used in class, explain how VQ-VAE works. The VQ-VAE idea found a lot of utility in a particular generative task in computer vision. Do a literature survey and describe how VQ-VAE helped improve the state of the art in this task. Cite relevant papers.

References

[1] Aaron Van Den Oord, Oriol Vinyals, et al. Neural discrete representation learning. *Advances in neural information processing systems*, 30, 2017. 8