Coursework 3: Generative Models

Instructions

Please submit on CATe a zip file named *CW3.zip* containing the following:

1. A version of this notebook containing your answers. Write your answers in the cells below each question.
2. Your trained models as *CAE_model.pth, DCGAN_model_D.pth, DCGAN_model_G.pth*
3. You training losses as *train_losses_CAE.npy*, *train_losses_D.npy*, *train_losses_G.npy*

Working environment:

Similarly to the previous coursework, we recommend that you use Google Colaboratory in order to train the required networks.

The deadline for submission is 19:00, Thursday 28th February, 2019

Introduction

For this coursework you are asked to implement two commonly used generative models:

1. A **Convolutional Autoencoder (CAE)**
2. A **Deep Convolutional Generative Adversarial Network (DCGAN)**

The dataset you will be using is the CIFAR-10 (https://www.cs.toronto.edu/~kriz/cifar.html (https://www.cs.toronto.edu/~kriz/cifar.html)).

Part 1 (50 points)

1. For the CAE, the success of your models will be tested as follows:
   - **By the autoencoders' reconstruction error**. You will need to achieve a low enough error in order to reconstruct the images of the dataset with relatively high fidelity. You will have to provide us with your best model's training loss curve, reconstruction error on the test set and some reconstructed images in the respective cells.
   - **By the representation learning capabilities of your model**. In particular, autoencoders are known to be able to learn quite informative features in their latent space (embeddings) that can later be used for downstream tasks. In this coursework you are asked to use the representations that your pre-trained encoder yields in order to do image classification. You can use your favourite classification module on top of the features in order to solve the problem *(Hint: a simple Support Vector Machine - SVM - is acceptable. Alternatively, you can devise more complex models such as a Multilayer Fully Connected Network)*.

Part 2 (50 points)

1. For the DCGAN, The success of your models will be tested as follows:
   - **By the model's training error**. You will need to achieve relatively balanced errors for the generator and the discriminator of your model in order to sample realistic images from the generator. You will have to provide us with your best model's training losses curves, a discussion on how you concluded to the chosen architecture, and visualizations of generated samples in the respective cells. Your results do not have to be perfect, however a good discussion on the choice of architecture will be valued.
   - **By avoiding mode collapse**. A common problem of training GANs is that they end up generating only a few different samples (if not only one), rather than learning the whole distribution of the training data. This problem is referred to as mode collapse. You will need to make a discussion on whether you noticed mode collapse or not during your experimentation and if yes, how you addressed it.

In [1]:
```
import os
import numpy as np
import torch
import torch.nn as nn
from torch.utils.data import DataLoader
from torch.utils.data import sampler
from torchvision import datasets, transforms
from torchvision.utils import save_image, make_grid
import torch.nn.functional as F
import matplotlib.pyplot as plt


def denorm(x, channels=None, w=None ,h=None, resize = False):
    x = 0.5 * (x + 1)
    x = x.clamp(0, 1)
    if resize:
        if channels is None or w is None or h is None:
            print('Number of channels, width and height must be pro
vided for resize.')
        x = x.view(x.size(0), channels, w, h)
    return x

def show(img):
    if torch.cuda.is_available():
        img = img.cpu()
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1,2,0)))
```

Device Selection

In [2]:
```
GPU = True
device_idx = 1
if GPU:
    device = torch.device("cuda:" + str(device_idx) if torch.cuda.i
s_available() else "cpu")
else:
    device = torch.device("cpu")
print(device)
```

```
cpu
```

Reproducibility

In [3]:
```
# We set a random seed to ensure that your results are reproducible
.
if torch.cuda.is_available():
    torch.backends.cudnn.deterministic = True
torch.manual_seed(0)
```

Out[3]: <torch._C.Generator at 0x1062eed70>

Data loading

In [4]:
```
batch_size = 128

if not os.path.exists('./CW/CAE'):
    os.makedirs('./CW/CAE')
if not os.path.exists('./CW/DCGAN'):
    os.makedirs('./CW/DCGAN')


NUM_TRAIN = 49000

transform = transforms.Compose([
     transforms.ToTensor(),
     transforms.Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5)
)
])


data_dir = './datasets'
cifar10_train = datasets.CIFAR10(data_dir, train=True, download=Tru
e,
                             transform=transform)
cifar10_val = datasets.CIFAR10(data_dir, train=True, download=True,
                             transform=transform)
cifar10_test = datasets.CIFAR10(data_dir, train=False, download=Tru
e,
                            transform=transform)

loader_train = DataLoader(cifar10_train, batch_size=batch_size,
                          sampler=sampler.SubsetRandomSampler(range
(NUM_TRAIN)))
loader_val = DataLoader(cifar10_val, batch_size=batch_size,
                        sampler=sampler.SubsetRandomSampler(range(N
UM_TRAIN, 50000)))
loader_test = DataLoader(cifar10_test, batch_size=batch_size)

it = iter(loader_test)
sample_inputs, _ = next(it)
fixed_input = sample_inputs[0:32, :, :, :]
save_image(denorm(fixed_input), './CW/CAE/input_sample.png')
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.g
z to ./datasets/cifar-10-python.tar.gz
Files already downloaded and verified
Files already downloaded and verified
```

Part 1 - Convolutional Autoencoder

Part 1.1 (30 points)

Your Task:

a. Implement the CAE architecture. Fill in the missing parts in the cells below in order to complete the CAE class. You will need to define:

- The hyperparameters
- The constructor
- `encode`
- `decode`

b. Plot your training loss curve (x-axis: epochs, y-axis: loss)

c. Calculate the reconstruction error on your test set

d. Visualize a subset of the images of the test set and their reconstructions

For b., c. and d. the code is already given. Make sure that the version of the notebook you deliver includes these results.

Some reccomendations:

- add several convolutional layers (3-4).
- accelerate training with batch normalization after every convolutional layer or fully connected layer.
- use the appropriate activation functions.
- Encoder module: hierarchially downsample your images with pooling layers, or strided convolutions.
- Decoder module: the upsampling can be done with various methods, such as nearest neighbor upsampling (`torch.nn.Upsample`) or transposed convolutions(`torch.nn.ConvTranspose2d`).

Try to follow the common practices for CNNs (e.g small receptive fields, max pooling, RELU activations), in order to narrow down your possible choices. **You will need to choose sufficiently large size for your latent vectors (hidden_size variable), in order to allow enough capacity for your network to represent the data.**

The number of epochs that will be needed in order to train the network will vary depending on your choices. In most of the cases, it will be a long procedure (a few hours), so you can leave your notebook running until the training converges. You don't need to train the network to an extreme if you don't have the time. As an advice, we recommend that while experimenting you should allow around 20 epochs and if the loss doesn't sufficiently drop, restart the training with a more powerful architecture.

Hyper-parameter selection

In [5]:
```
# *CODE FOR PART 1.1 IN THIS CELL*

### Choose the number of epochs and the learning rate.
num_epochs = 10
learning_rate  = 0.1
###

# Define here other hyperparameters that you used.
```

Define model

In [ ]:
```
# *CODE FOR PART 1.1 IN THIS CELL*

### Choose a value for the latent space dimension and use it in you
r model
hidden_size = None
###

class CAE(nn.Module):
    def __init__(self):
        super(CAE, self).__init__()

        ###########################################################
############
        #                       ** START OF YOUR CODE **
        ###########################################################
############

        ###########################################################
############
        #                       ** END OF YOUR CODE **
        ###########################################################
############

    def encode(self, x):

        ###########################################################
############
        #                       ** START OF YOUR CODE **
        ###########################################################
############

        ###########################################################
############
        #                       ** END OF YOUR CODE **
        ###########################################################
############

    def decode(self, z):
```

```
        ########################################################################
        #                        ** START OF YOUR CODE **
        ########################################################################

        ########################################################################
        #                        ** END OF YOUR CODE **
        ########################################################################

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

Define Loss function

In [ ]:
```
criterion = nn.MSELoss(reduction='mean')  # can we use any other loss here? You are free to choose.
def loss_function_CAE(recon_x, x):
    recon_loss = criterion(recon_x, x)
    return recon_loss
```

Initialize Model and print number of parameters

In [ ]:
```
model = CAE().to(device)
params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print("Total number of parameters is: {}".format(params))
print(model)
```

Choose and initialize optimizer

In [ ]:
```
# You are free to add a scheduler or change the optimizer if you want. We chose one for you for simplicity.
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

Train

In [ ]:
```
train_losses = []
model.train()
for epoch in range(num_epochs):
    train_loss = 0
    for batch_idx, data in enumerate(loader_train):
        img, _ = data
        img = img.to(device)
        optimizer.zero_grad()
        # forward
        recon_batch = model(img)
        loss = loss_function_CAE(recon_batch, img)
        # backward
        loss.backward()
        train_loss += loss.item()
        optimizer.step()
    # print out losses and save reconstructions for every epoch
    print('epoch [{}/{}], loss:{:.4f}'
          .format(epoch + 1, num_epochs, train_loss / len(loader_tr
ain)))
    recon = model(fixed_input.to(device))
    recon = denorm(recon.cpu())
    save_image(recon, './CW/CAE/reconstructed_epoch_{}.png'.format(
epoch))
    train_losses.append(train_loss/ len(loader_train))

# save the model and the loss values
np.save('./CW/CAE/train_losses.npy', np.array(train_losses))
torch.save(model.state_dict(), './CW/CAE/CAE_model.pth')
```

Train loss curve

In [ ]:
```
import matplotlib.pyplot as plt
train_losses = np.load('./CW/CAE/train_losses.npy')
plt.plot(list(range(0,train_losses.shape[0])), train_losses)
plt.title('Train Loss')
plt.show()
```

Test set reconstruction error

In [ ]:
```
# load the model
model.load_state_dict(torch.load('./CW/CAE/model.pth'))
model.eval()
test_loss = 0
with torch.no_grad():
    for i, data in enumerate(loader_test):
        img,_ = data
        img = img.to(device)
        recon_batch = model(img)
        test_loss += loss_function_CAE(recon_batch, img)
    # loss calculated over the whole test set
    test_loss /= len(loader_test.dataset)
    print('Test set loss: {:.4f}'.format(test_loss))
```

Test set images and reconstructions

In [ ]:
```
# load the model
model.load_state_dict(torch.load('./CW/CAE/CAE_model.pth'))
it = iter(loader_test)
sample_inputs, _ = next(it)
fixed_input = sample_inputs[0:32, :, :, :]

# visualize the original images of the last batch of the test set
img = make_grid(denorm(fixed_input), nrow=8, padding=2, normalize=F
alse,
                range=None, scale_each=False, pad_value=0)
show(img)
```

In [ ]:
```
with torch.no_grad():
    # visualize the reconstructed images of the last batch of test
set
    recon_batch = model(fixed_input.to(device)).cpu()
    recon_batch = make_grid(denorm(recon_batch), nrow=8, padding=2,
normalize=False,
                            range=None, scale_each=False, pad_value
=0)
    show(recon_batch)
```

Part 1. 2 (20 points)

Your Task:

In this part of the exercise you will use your pretrained encoder as a feature extractor in order to solve a downstream task:

- For every sample of your training set you will need to extract its latent representation by passing it through the encoder.
- Create a classifier of your choice and train it with the extracted features in order to predict the class that each image belongs to. You can access the sample's classes as follows:

```
 it = iter(loader_test)
samples, classes = next(it)
```

- Use the encoder to encode all your test images into latent representations and then use your trained classifier to predict their classes
- Print the accuracy of your model.

The classifier can be trained with representations that do not yield very accurate reconstructions, so you can stop your training even if the reconstructed images are blurry. Also, note that you do not have to acheive high classification accuracy to get full marks for this question. Instead, focus on describing how you experimented in order to build your best classifier.

In [ ]:
```
# *CODE FOR PART 1.2 IN THIS CELL*

####################################################################
####
#                       ** START OF YOUR CODE **
####################################################################
####

####################################################################
####
#                       ** END OF YOUR CODE **
####################################################################
####
# name your accuracy variable as accuracy
print('Classification accuracy: {:.4f}'.format(accuracy))
```

In [ ]:
```

```

Part 2 - Deep Convolutional GAN

In this task, your main objective is to train a DCGAN (https://arxiv.org/abs/1511.06434 (https://arxiv.org/abs/1511.06434)) on the CIFAR-10 dataset. You should experiment with different architectures, tricks for stability in training (such as using different activation functions, batch normalization, different values for the hyper-parameters, etc.). In the end, you should provide us with:

- your best trained model (which we will be able to run),
- some generations for the fixed latent vectors $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ we have provided you with (train for a number of epochs and make sure there is no mode collapse),
- plos with the losses for the discriminator D and the generator G as the training progresses and explain whether your produced plots are theoretically sensible and why this is (or not) the case.
- a discussion on whether you noticed any mode collapse, where this behaviour may be attributed to, and explain what you did in order to cope with mode collapse.

Clarification: You should not be worrying too much about getting an "optimal" performance on your trained GAN. We want you to demonstrate to us that you experimented with different types of DCGAN variations, report what difficulties transpired throughout the training process, etc. In other words, if we see that you provided us with a running implementation, that you detail different experimentations that you did before providing us with your best one, and that you have grasped the concepts, you can still get full marks. The attached model does not have to be perfect.

Part 2.1 (30 points)

Your Task:

a. Implement the DCGAN architecture. Fill in the missing parts in the cells below in order to complete the Generator and Discriminator classes. You will need to define:

- The hyperparameters
- The constructors
- `decode`
- `discriminator`

b. visualize images sampled from your best model's generator.

c. Discuss the experimentations which led to your final architecture. You can plot losses or generated results by other architectures that you tested to back your arguments (but this is not necessary to get full marks).

For b. the code is already given. Make sure that the version of the notebook you deliver includes these results.

Recomendations for experimentation:

- use the architecture that you implemented for the Autoencoder of Part 1 (encoder as discriminator, decoder as generator).
- use the architecture desribed in the DCGAN paper (https://arxiv.org/abs/1511.06434 (https://arxiv.org/abs/1511.06434)).

Some general reccomendations:

- add several convolutional layers (3-4).
- accelerate training with batch normalization after every convolutional layer.
- use the appropriate activation functions.
- Generator module: the upsampling can be done with various methods, such as nearest neighbor upsampling (`torch.nn.Upsample`) or transposed convolutions(`torch.nn.ConvTranspose2d`).
- Discriminator module: Experiment with batch normalization (`torch.nn.BatchNorm2d`) and leaky relu (`torch.nn.LeakyReLu`) units after each convolutional layer.

Try to follow the common practices for CNNs (e.g small receptive fields, max pooling, RELU activations), in order to narrow down your possible choices.

The number of epochs that will be needed in order to train the network will vary depending on your choices. As an advice, we recommend that while experimenting you should allow around 20 epochs and if the loss doesn't sufficiently drop, restart the training with a more powerful architecture. You don't need to train the network to an extreme if you don't have the time.

Hyper-parameter selection

In [ ]:
```
# *CODE FOR PART 2.1 IN THIS CELL*

### Choose the number of epoch, the learning rate
#   and the size of the Generator's input noise vetor.
num_epochs = None
learning_rate  = None
latent_vector_size = None
###

# Define here other hyperparameters that you used.
```

In [ ]:
```
# *CODE FOR PART 2.1 IN THIS CELL*


class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        ############################################################
############
        #                       ** START OF YOUR CODE **
        ############################################################
############

        ############################################################
############
        #                       ** END OF YOUR CODE **
        ############################################################
############

    def decode(self, z):
        ############################################################
############
        #                       ** START OF YOUR CODE **
        ############################################################
############

        ############################################################
############
        #                       ** END OF YOUR CODE **
        ############################################################
############
        return x

    def forward(self, z):
        return self.decode(z)


class Discriminator(nn.Module):
```

```
    def __init__(self):
        super(Discriminator, self).__init__()
        ###########################################################
############
        #                          ** START OF YOUR CODE **
        ###########################################################
############

        ###########################################################
############
        #                          ** END OF YOUR CODE **
        ###########################################################
############

    def discriminator(self, x):
        ###########################################################
############
        #                          ** START OF YOUR CODE **
        ###########################################################
############

        ###########################################################
############
        #                          ** END OF YOUR CODE **
        ###########################################################
############

        return out

    def forward(self, x):
        out = self.discriminator(x)
        return outs.view(-1, 1).squeeze(1)
```

Initialize Model and print number of parameters

You can use method `weights_init` to initialize the weights of the Generator and Discriminator networks. Otherwise, implement your own initialization, or do not use at all. You will not be penalized for not using initialization.

In [ ]:

```
# custom weights initialization called on netG and netD
def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        m.weight.data.normal_(0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        m.weight.data.normal_(1.0, 0.02)
        m.bias.data.fill_(0)
```

In [ ]:
```
use_weights_init = True

model_G = Generator().to(device)
if use_weights_init:
    model_G.apply(weights_init)
params_G = sum(p.numel() for p in model_G.parameters() if p.require
s_grad)
print("Total number of parameters in Generator is: {}".format(param
s_G))
print(model_G)
print('\n')

model_D = Discriminator().to(device)
if use_weights_init:
    model_D.apply(weights_init)
params_D = sum(p.numel() for p in model_D.parameters() if p.require
s_grad)
print("Total number of parameters in Discriminator is: {}".format(p
arams_D))
print(model_D)
print('\n')

print("Total number of parameters is: {}".format(params_G + params_
D))
```

Define loss function

In [ ]:
```
criterion = nn.BCELoss(reduction='mean')
def loss_function(out, label):
    loss = criterion(out, label)
    return loss
```

Choose and initialize optimizers

In [ ]:
```
# setup optimizer
# You are free to add a scheduler or change the optimizer if you wa
nt. We chose one for you for simplicity.
beta1 = 0.5
optimizerD = torch,optim.Adam(model_D.parameters(), lr=learning_rat
e, betas=(beta1, 0.999))
optimizerG = torch.optim.Adam(model_G.parameters(), lr=learning_rat
e, betas=(beta1, 0.999))
```

Define fixed input vectors to monitor training and mode collapse.

In [ ]:
```
fixed_noise = torch.randn(batch_size, latent_vector_size, 1, 1, dev
ice=device)
real_label = 1
fake_label = 0
```

Train

In [ ]:
```
export_folder = './CW/DCGAN'
train_losses_G = []
train_losses_D = []

for epoch in range(num_epochs):
    for i, data in enumerate(loader_train, 0):
        train_loss_D = 0
        train_loss_G = 0

        ############################
        # (1) Update D network: maximize log(D(x)) + log(1 - D(G(z)
))
        ###########################
        # train with real
        model_D.zero_grad()
        real_cpu = data[0].to(device)
        batch_size = real_cpu.size(0)
        label = torch.full((batch_size,), real_label, device=device
)

        output = model_D(real_cpu)
        errD_real = loss_function(output, label)
        errD_real.backward()
        D_x = output.mean().item()

        # train with fake
        noise = torch.randn(batch_size, latent_vector_size, 1, 1, d
evice=device)
        fake = model_G(noise)
        label.fill_(fake_label)
        output = model_D(fake.detach())
        errD_fake = loss_function(output, label)
        errD_fake.backward()
        D_G_z1 = output.mean().item()
        errD = errD_real + errD_fake
        train_loss_D += errD.item()
        optimizerD.step()

        ############################
        # (2) Update G network: maximize log(D(G(z)))
        ###########################
        model_G.zero_grad()
        label.fill_(real_label)  # fake labels are real for generat
```

```
or cost
        output = model_D(fake)
        errG = loss_function(output, label)
        errG.backward()
        D_G_z2 = output.mean().item()
        train_loss_G += errG.item()
        optimizerG.step()

        print('[%d/%d][%d/%d] Loss_D: %.4f Loss_G: %.4f D(x): %.4f
D(G(z)): %.4f / %.4f'
              % (epoch, num_epochs, i, len(loader_train),
                 errD.item(), errG.item(), D_x, D_G_z1, D_G_z2))

    if epoch == 0:
        save_image(denorm(real_cpu.cpu()), './CW/DCGAN/real_samples
.png')

    fake = model_G(fixed_noise)
    save_image(denorm(fake.cpu()), './CW/DCGAN/fake_samples_epoch_%
03d.png' % epoch)
    train_losses_D.append(train_loss_D / len(loader_train))
    train_losses_G.append(train_loss_G / len(loader_train))

# save losses and models
np.save(np.array(train_losses_D),'./CW/DCGAN/train_losses_D.npy')
np.save(np.array(train_losses_G),'./CW/DCGAN/train_losses_G.npy')
torch.save(model_G.state_dict(), './CW/DCGAN/DCGAN_model_G.pth')
torch.save(model_D.state_dict(), './CW/DCGAN/DCGAN_model_D.pth')
```

In [ ]:

```
# DISCUSS THE SELECTION OF THE ARCHITECTURE IN THIS CELL*
```

Generator samples

In [ ]:

```
it = iter(loader_test)
sample_inputs, _ = next(it)
fixed_input = sample_inputs[0:32, :, :, :]

# visualize the original images of the last batch of the test set
img = make_grid(denorm(fixed_input), nrow=4, padding=2, normalize=F
alse,
                range=None, scale_each=False, pad_value=0)
show(img)
```

In [ ]:
```
# load the model
model_G.load_state_dict(torch.load('./CW/DCGAN/DCGAN_model_G.pth'))
input_noise = torch.randn(batch_size, latent_vector_size, 1, 1, dev
ice=device)

with torch.no_grad():
    # visualize the generated images
    generated = model_G(input_noise).cpu()
    generated = make_grid(denorm(generated)[:32], nrow=8, padding=2
, normalize=False,
                        range=None, scale_each=False, pad_value=0)
    show(generated)
```

Part 2.2 (10 points)

Train losses curves

Your task:

Plot the losses curves for the discriminator D and the generator G as the training progresses and explain whether the produced curves are theoretically sensible and why this is (or not) the case (x-axis: epochs, y-axis: loss).

The code for generating the plot is already given. Make sure that the version of the notebook you deliver includes these results.

In [ ]:
```
import matplotlib.pyplot as plt
train_losses_D = np.load('./CW/DCGAN/train_losses_D.npy')
train_losses_G = np.load('./CW/DCGAN/train_losses_G.npy')
plt.plot(list(range(0,train_losses_D.shape[0])), train_losses_D, la
bel='loss_D')
plt.plot(list(range(0,train_losses_G.shape[0])), train_losses_G, la
bel='loss_G')
plt.legend()
plt.title('Train Losses')
plt.show()
```

In [ ]:
```
# ANSWER FOR PART 2.2 IN THIS CELL*
```

Part 2.3 (10 points)

Your task:

Based on the images created by your generator using the `fixed_noise` vector during training, provide a discussion on whether you noticed any mode collapse, where this behaviour may be attributed to, and explain what you did in order to cope with mode collapse.

In [ ]:
```
# ANSWER FOR PART 2.3 IN THIS CELL*
```