ENS Lyon
2024/2025
Mathurin Massias, Titouan Vayer, Quentin Bertrand

Lab 3 : Generative modeling with optimal transport

In this practical session[1] we will practice the Wasserstein GAN in order to generate samples according to a distribution that we want to approximate. If p_{data} is the data distribution and p_θ is the parametrized distribution the Wasserstein GAN aims at minimizing the quantity

$$\min_\theta \; W_1(p_{\text{data}}, p_\theta) = \min_\theta \max_{f \in \text{Lip}_1} \; \mathbb{E}_{x\sim p_{\text{data}}}[f(x)] - \mathbb{E}_{x\sim p_\theta}[f(x)]\,. \tag{1}$$

In practice $x \sim p_\theta \iff x = \text{G}_\theta(z), z \sim \mathcal{N}(0, I_d)$ where G_θ is a neural network with parameters θ. We usually call G_θ the generator. In practice we also tackle (1) by parametrizing f with another neural network D_β that should be approximately 1-Lipschitz and overall we tackle the optimization problem

$$\min_\theta \max_\beta \; \mathbb{E}_{x\sim p_{\text{data}}}[\text{D}_\beta(x)] - \mathbb{E}_{z\sim\mathcal{N}(0,I_d)}[\text{D}_\beta(G_\theta(z))]\,. \tag{2}$$

For this we will need to have `pytorch` installed and then to import the following packages.

```
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
import numpy as np
from torch.utils.data import DataLoader
import torch.autograd as autograd
```

- Exercise 1: WGAN with clipping and gradient penalty -

We will work with synthetic data generated according the the following snippet.

```
nb_samples = 10000
radius = 1
nz = .1
X_train = torch.zeros((nb_samples, 2))
r = radius + nz*torch.randn(nb_samples)
theta = torch.rand(nb_samples)*2*torch.pi
X_train[:, 0] = r*torch.cos(theta)
X_train[:, 1] = r*torch.sin(theta)
```

Q1. Plot the discrete distribution corresponding the the data above. Is it p_{data} ?

We will now create our two neural networks G_θ and D_β.

Q2. What is the size of the input and output of these functions ?

Q3. The architecture for the two neural networks are the following: both are Multilayer perceptron with two hidden layers with sizes respectively 128 and 64 and ReLU activation functions. Using `nn.Linear` and `nn.ReLU` complete the following code to produce the different architectures.

[1] This practical session is based on code by Nicolas Courty and Rémi Flamary.

```
class Generator(nn.Module):
    def __init__(self, noise_dim=10):
        super(Generator, self).__init__()
        self.noise_dim = noise_dim
        self.model = nn.Sequential(
        # to complete
        )

    def forward(self, z):
        return # to complete

class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.model = nn.Sequential(
        # to complete
        )

    def forward(self, x):
        return # to complete
```

In practice to train the generator and discriminator solving (3) we sample some noise from the latent distribution and some real data points and we minimize/maximize a finite average. To plot the generated samples we can use the following snippet

```
def generate_images(generator_model, noise_dim, num_samples=1000):
    with torch.no_grad():
        z = torch.Tensor(np.random.normal(0, 1, (num_samples, noise_dim)))
        predicted_samples = generator_model(z.type(torch.float32))
    plt.figure(figsize=(6, 6))
    plt.scatter(X_train[:, 0], X_train[:, 1], s=40, alpha=0.2,
     edgecolor='k', marker='+', label='original samples')
    plt.scatter(predicted_samples[:, 0], predicted_samples[:, 1], s=10,
                alpha=0.9, c='r', edgecolor='k', marker='o', label='predicted')
    plt.grid(alpha=0.5)
    plt.legend(loc='best')
    plt.tight_layout()
    plt.show()
```

Q4. We will first impose limit the Lipschitz constant of the discriminator by clipping its weights after each training step. First initialize the discriminator/generator with a latent dimension equal to 2.

Q5. To train the networks fill the following code and comment it

```
# Hyperparameters
lr_G = #to choose
lr_D = #to choose
n_epochs = #to choose
clip_value = #to choose
n_critic = 5
batch_size = #to choose
optimizer_G = torch.optim.Adam(# to fill,
lr=lr_G, betas=(0.5, 0.9))
optimizer_D = torch.optim.Adam(# to fill,
lr=lr_D, betas=(0.5, 0.9))
dataloader = DataLoader(X_train, batch_size, shuffle=True) #data loader
for epoch in range(n_epochs):
    for i, x in enumerate(dataloader):
        x = x.type(torch.float32)
        # ----------------------
        #  Train Discriminator
        # ----------------------
        optimizer_D.zero_grad()

        # Sample noise for generator input
        z = #to fill

        # Generate a batch of fake data
        fake_x = # to fill
        # Compute loss for the discriminator
        loss_D = #to fill
        loss_D.backward() # backpropagation
        optimizer_D.step()

        # Clip weights of discriminator
        for p in discriminator.parameters():
            p.data.clamp_(-clip_value, clip_value)

        # Train the generator every n_critic iterations
        if i % n_critic == 0:
            # ------------------
            #  Train Generator
            # ------------------
            optimizer_G.zero_grad()
            fake_x = # to fill
            loss_G = # to fill

            loss_G.backward()
            optimizer_G.step()

    # Visualization of intermediate results
    if epoch % 10 == 0:
        print("Epoch: ", epoch)
        generate_images(generator, noise_dim)
```

Q6. Test the code with specific choices of hyperparameters. Is it sensitive to hyperparameters ?

The weight clipping is a very crude solution for enforcing the Lipschitz constant. The idea now is to rely

on a different loss promoting this regularization. One possibility writes

$$\min_{\theta}\max_{\beta}\ \mathbb{E}_{x\sim p_{\text{data}}}[\mathrm{D}_\beta(x)] - \mathbb{E}_{z\sim\mathcal{N}(0,I_d)}[\mathrm{D}_\beta(G_\theta(z))] + \lambda\cdot\mathbb{E}_{\hat{x}}[(\|\nabla\,\mathrm{D}_\beta(\hat{x})\|_2 - 1)^2]\,. \tag{3}$$

In practice $\hat{x}$ is sampled as a linear combination of real and fake data points.

Q7. The following code computes $\nabla\,\mathrm{D}_\beta(\hat{x}_1), \cdots, \nabla\,\mathrm{D}_\beta(\hat{x}_n)$.

```
def compute_gradient_penalty(D, real_samples, fake_samples):
    # Random weight term for interpolation between real and fake samples
    alpha = torch.Tensor(np.random.random((real_samples.size(0), 1)))
    # Get random interpolation between real and fake samples
    interpolates = (alpha * real_samples + ((1 - alpha) * fake_samples))
    d_interpolates = D(interpolates.requires_grad_(True))
    # Get gradient w.r.t. interpolates
    gradients = autograd.grad(
        outputs=d_interpolates,
        inputs=interpolates,
        grad_outputs=torch.ones_like(d_interpolates),
        create_graph=True,
        retain_graph=True,
        only_inputs=True,
    )[0]
    return gradients
```

Use it to retrain the previous networks without weight clipping but with gradient penalty. You must choose a regularization parameter λ.

- Exercise 2 (if time): WGAN on MNIST data -

Use the previous work to generate MNIST images (you can use Google Collab if you want). To load these data you can use the snippet (good luck!).

```
import torchvision.transforms as transforms
from torchvision import datasets
dataloader = torch.utils.data.DataLoader(
    datasets.MNIST(
        "../../data/mnist",
        train=True,
        download=True,
        transform=transforms.Compose(
            [transforms.Resize(img_size), transforms.ToTensor(),
             transforms.Normalize([0.5], [0.5])]
        ),
    ),
    batch_size=batch_size,
    shuffle=True,
)
```