# Advanced Machine Learning – Optimal Transport: Exam

Master 2 IASD – 2024/2025

The exam consists of a mix of theoretical questions and coding exercises, on the module Optimal Transport for the course Advanced ML. **Please read these instructions carefully before beginning the exam.**

## Instructions

### Exam content

- The exam contains two exercises. Each exercise is designed to be solved independently.
- Points may be awarded for incomplete but well-reasoned responses.

### Submission guidelines

- The completed exam must be submitted within the allocated time: **today (March 11, 2025) before 12h15**.
- **Send your completed exam as a single archive named `firstname_lastname`**, at kimia.nadjahi@ens.fr
- The completed exam should consist of **two Python files (one per exercise)** containing your answers to the questions. Answers to the theoretical questions can be written directly in the Python file as comments, or in a separate text (or LaTeX) file.
- This exam is to be completed individually: collaboration or communication with other students is not allowed.

### Materials allowed

- The lecture notes are permitted (slides for lectures 1 and 2).
- No external resources, including textbooks or online materials, are allowed.
- The use of Large Language Models (LLMs) or any AI-powered tools is strictly prohibited.

## Exercise 1. The Sliced-Wasserstein distance (40 points)

The Sliced-Wasserstein distance (SW) is a metric inspired by optimal transport, which is defined as: for any two probability measures $\mu$ and $\nu$ on $\mathbb{R}^d$,

$$\mathrm{SW}_2(\mu,\nu)^2 = \mathbb{E}_{\theta\sim\mathcal{U}(\mathbb{S}^{d-1})}\left[\mathrm{W}_2\left((P_\theta)_\sharp,(P_\theta)_\sharp\nu\right)^2\right] \tag{1}$$

where $\mathbb{S}^{d-1} = \{\theta \in \mathbb{R}^d : \|\theta\| = 1\}$ is the $d$-dimensional sphere, $\mathcal{U}(\mathbb{S}^{d-1})$ is the uniform distribution on $\mathbb{S}^{d-1}$, and $P_\theta(x) = \theta^\top x$ for any $(\theta, x) \in \mathbb{S}^{d-1} \times \mathbb{R}^d$.

The goal of this exercise is to implement the Sliced-Wasserstein distance and analyze its behavior in practice. In particular, we are going to compare SW to the Wasserstein distance. Consider two discrete probability measures, $\mu = \sum_{i=1}^n a_i\delta_{x_i}$ and $\nu = \sum_{j=1}^m b_j\delta_{y_j}$.

**Question 1.1.** Let $\theta \in \mathbb{S}^{d-1}$. Give the mathematical expressions for $(P_\theta)_\sharp\mu$ and $(P_\theta)_\sharp\nu$, and explain their meanings. **(4 points)**

**Question 1.2.** Give the expression of $\mathrm{W}_2\left((P_\theta)_\sharp\mu,(P_\theta)_\sharp\nu\right)$. How to compute it in practice? **(3 pts)**

Based on the elements above, one can now implement SW. The first step is to import the following packages.

```python
import numpy as np
import ot
import matplotlib.pyplot as plt
```

The following code generates synthetic data in $\mathbb{R}^d$ by sampling $n$ realizations from a mixture of two Gaussians.

```python
def generate_data(n, d):
    means = [-np.ones(d), np.ones(d)]
    # random covariance matrix
    A = np.random.randn(d, d)
    cov = np.dot(A, A.T)  # ensure positive semi-definiteness
    # sample points
    idx = np.random.choice(2, size=n)
    samples = np.array([np.random.multivariate_normal(means[comp], cov) for
        comp in idx])
    return samples
```

One can visualize the generated samples in dimension 2 using the script below.

```python
n = 1000  # number of samples
d = 2     # data dimension
Xs = generate_data(n,d)
Ys = generate_data(n,d)
plt.plot()
plt.scatter(Xs[:, 0], Xs[:, 1], label="mu",alpha=0.7)
plt.scatter(Ys[:, 0], Ys[:, 1], label="nu", alpha=0.7)
plt.legend()
plt.show()
```

**Question 1.3.** Complete the code below to implement SW. Apply the completed function to compare the data generated above. **(6 pts)**

```python
def sliced_wasserstein(X, Y, m=100):
    d = X.shape[1]
    sw2 = 0.0
    for _ in range(m):
        # Sample a random vector uniformly on the unit sphere
        theta = np.random.randn(d)
        theta /= np.linalg.norm(theta)

        # Implement answer to question 1
        X_theta = # to complete
        Y_theta = # to complete

        # Compute W2 (answer to question 2)
        w2 = # to complete
        sw2 += w2
    sw2 /= m
    return np.sqrt(sw2)
```

**Question 1.4.** What is the computational complexity of computing $\mathrm{SW}_2(\mu,\nu)$ (for discrete measures)? How does this compare to computing $\mathrm{W}_2(\mu,\nu)$ directly? *(No code)* **(5 pts)**

**Question 1.5.** Explain the impact of the hyperparameter $m$ on your implementation of SW, in terms of running time, precision (or bias) and variance. You can add empirical results to illustrate your answer. **(5 pts)**

Next, we fix $m = 100$ and study the influence of the $(n, d)$ on SW with the code below. We also evaluate the Wasserstein distance and its entropic regularization using the `POT` library.

```python
d = 2
ns = np.array([10, 100, 1000, 2000])
len_n = len(ns)
sw2s = np.zeros(len_n)
w2s = np.zeros(len_n)
entropy_w2s = np.zeros(len_n)

for i in range(len_n):
    # generate data from the same distribution
    n = ns[i]
    data = generate_data(n,d)
    Xs = data[:int(n/2)]
    Ys = data[int(n/2):]
    M =  # to complete
    a =  # to complete
    b =  # to complete
    sw2s[i] = sliced_wasserstein(Xs, Ys, m=100)
    w2s[i] = np.sqrt(ot.emd2(a, b, M))
    entropy_w2s[i] = np.sqrt(ot.sinkhorn2(a, b, M, reg=2))

# Plot the results
plt.figure(figsize=(8, 5))
plt.plot(ns, sw2s, label="SW")
plt.plot(ns, w2s, label="Wass.")
plt.plot(ns, entropy_w2s, label="Sinkhorn")
plt.xscale("log")  # log scale for better visualization
plt.xlabel("n")
plt.title(f"Comparison␣for␣d={d}")
plt.legend()
plt.grid(True)
plt.show()
```

**Question 1.6.** Complete the code above to implement the Wasserstein distance and its entropic regularization. **(4 pts)**

**Question 1.7.** Discuss the impact of $n$ on the implemented SW, Wasserstein distance, and entropic regularization. In particular, is the entropic regularized version of $\mathrm{W}_2$ a proper distance? **(5 pts)**

**Question 1.8.** Increase the value of $d$ in the above code and analyze its impact on SW as compared to the Wasserstein distance and entropic regularizated OT. **(5 pts)**

**Question 1.9.** Suppose that $\mu$ and $\nu$ are Gaussian distributions. In this case, is it relevant to use the Sliced-Wasserstein distance instead of the Wasserstein distance? Why? *(No code)* **(3 pts)**

## Exercise 2. Generative modeling with optimal transport (60 points)

In this exercise, we will implement a generative model called Wasserstein GAN to generate samples that follow a target distribution we aim to approximate.

Denote by $p_{\text{data}}$ the data distribution and by $p_\theta$ a distribution parameterized by $\theta$. The objective of the Wasserstein GAN is

$$\min_{\theta} \max_{f \in \text{Lip}_1} \mathbb{E}_{x\sim p_{\text{data}}}[f(x)] - \mathbb{E}_{x\sim p_\theta}[f(x)] \,, \tag{2}$$

where $\text{Lip}_1$ is the class of 1-Lipschitz functions: $\text{Lip}_1 = \{f \ : \ \forall(x,y),\ |f(x) - f(y)| \leq \|x - y\|\}$.

**Question 2.1.** How is the objective function (2) related to optimal transport? Is there a unique solution for the maximization problem? **(6 points)**

We choose $p_\theta$ such that $x \sim p_\theta \iff x = \text{G}_\theta(z), z \sim \mathcal{N}(0, I_d)$, where $\text{G}_\theta$ is a neural network with parameters $\theta$. In this context, $\text{G}_\theta$ is usually called the generator. Additionally, we parameterize $f$ with another neural network $\text{D}_\beta$ with parameters $\beta$, which should be approximately 1-Lipschitz. The objective (2) can thus be reformulated as follows.

$$\min_{\theta} \max_{\beta} \mathbb{E}_{x\sim p_{\text{data}}}[\text{D}_\beta(x)] - \mathbb{E}_{z\sim\mathcal{N}(0,I_d)}[\text{D}_\beta(G_\theta(z))] \,. \tag{3}$$

To complete this exercise, one needs to import the following packages.

```python
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
```

We consider synthetic data, which are sampled using the code below.

```python
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)
```

**Question 2.2.** Plot the discrete distribution of the synthetic data sampled above. Is it $p_{\text{data}}$? **(4 pts)**

Next, we implement the two neural networks $G_\theta$ and $D_\beta$.

**Question 2.3.** What are the dimensions of the input and output of $G_\theta$? Same question for $D_\beta$. **(4 pts)**

**Question 2.4.** Both neural networks have the same architecture: a multilayer perceptron based on two hidden layers of respective sizes 128 and 64, and ReLU activation functions. Using `nn.Linear` and `nn.ReLU`, complete the code below to implement $G_\theta$ and $D_\beta$. **(10 pts)**

```
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 solve (3), we replace the expectations by computing finite averages over a limited number of samples from the latent distribution $z \sim \mathcal{N}(0, I_d)$, and the data distribution $x \sim p_{\text{data}}$. We can use the following code to plot the samples generated by our Wasserstein GAN.

```
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()
```

**Question 2.5.** After each training step, we will clamp (or clip) $\beta$, meaning that all its elements are restricted to the range $[-a, a]$, where $a \in \mathbb{R}$ is a value chosen by the user. Explain the purpose of weight clipping in our context. **(5 pts)**

**Question 2.6.** Instantiate the two neural networks with a latent dimension equal to 2. Then, complete the following code to implement the training procedure for $G_\theta$ and $D_\beta$. **(10 pts)**

```
# 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 complete

        # Generate a batch of fake data
        fake_x = # to complete
        # Compute loss for the discriminator
        loss_D = # to complete
        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 complete
            loss_G = # to complete

            loss_G.backward()
            optimizer_G.step()

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

**Question 2.7.** Test your code with different hyperparameter choices. How sensitive is the model to these hyperparameters? **(5 pts)**

We will now replace the weight clipping with a different strategy. The idea is to solve the following optimization problem,

$$\min_{\theta} \max_{\beta} \ \mathbb{E}_{x\sim p_{\text{data}}}[\mathrm{D}_\beta(x)] - \mathbb{E}_{z\sim\mathcal{N}(0,I_d)}[\mathrm{D}_\beta(G_\theta(z))] - \lambda \cdot \mathbb{E}_{\hat{x}}[(\|\nabla \mathrm{D}_\beta(\hat{x})\|_2 - 1)^2]\,, \tag{4}$$

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

**Question 2.8.** What is the difference between (4) and the previous objective in (2)? Explain the motivation behind this change. **(5 pts)**

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

```python
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
```

**Question 2.9.** Use the above code to train the networks *without* weight clipping but with gradient penalty. This involves choosing a regularization parameter $\lambda$. **(6 pts)**

**Question 2.10.** Test different hyperparameters values and discuss their impact on the model's performance. **(5 pts)**