

TP6_HABASQUE_COERCHON

September 6, 2026

0.0.1 Name : Colin Coërchon et Chloé Habasque

1 Generative modeling in deep learning

*This notebook consists of common introductory questions, followed by 4 different generative models (GAN, VAE, Flow matching, Diffusion flow), and a common conclusive question. **Choose & implement 2 among the 4 models**, not all of them! If you train more models, write down below the 2 ones you want to be graded on. Do not forget the last common question.*

Generative modeling in machine learning can aim at achieving different goals.

The first, obvious one is that a generative model can be used to generate more data, to be used afterwards by another algorithm. While a generative model cannot create more information to solve the issue of having too small datasets, it could be used to solve anonymity questions. Typically, sharing a generative model trained on private data could allow the exploitation of the statistical property of this data without sharing the data itself (which can be protected by privacy matters for example).

Another goal is to use generative modeling to better understand the data at hand. This is based on the hypothesis that a model that successfully learned to generate (and generalize) a dataset should have internally learned some efficient and compressed representation of the information contained in the data. In this case, analysing a posteriori the learned representation may give us insights on the data itself.

The notion of a generative model however needs to be more formally specified, in order to work with. What does it mean for the model to generate data that “looks like” the original dataset? A mathematical formulation of that is necessary, in order to define a training objective that can be used efficiently. Having some expert rate the quality of all generated datapoints one by one is definitely not an option.

Thus, modeling our data and models as probability distributions comes to the rescue. If we consider our data as coming from some underlying probability distribution, that we will name p_D , our goal is thus to train our model to represent another probability distribution, which we will name p_θ , that should be some good approximation of p_D . Given that we only know p_D through some set of realisations from it (the dataset), we can never hope to learn it exactly.

Q1: Can you name some metrics that can be used to compare two given distributions p_D and p_θ ?

To compare two given distributions p_D and p_θ , several metrics are commonly used:

- **Kullback–Leibler (KL) divergence** : This measures how much information is

lost when p_θ is used to approximate p_D . Note that KL divergence is **not symmetric**.

$$D_{\text{KL}}(p_D \parallel p_\theta) = \sum_x p_D(x) \log \frac{p_D(x)}{p_\theta(x)}$$

- **Jensen–Shannon (JS) divergence** : The JS divergence is a **symmetric** and smoothed version of the KL divergence.

$$D_{\text{JS}}(p_D \parallel p_\theta) = \frac{1}{2}D_{\text{KL}}(p_D \parallel m) + \frac{1}{2}D_{\text{KL}}(p_\theta \parallel m), \quad \text{where } m = \frac{1}{2}(p_D + p_\theta)$$

- **Wasserstein Distance** : It measures the minimal “cost” of transporting mass to transform one distribution into another.

$$W(p_D, p_\theta) = \inf_{\gamma \in \Pi(p_D, p_\theta)} \mathbb{E}_{(x,y) \sim \gamma} [\|x - y\|]$$

- **Total Variation (TV) Distance**: The TV distance reflects the **maximum difference** in assigned probabilities between the two distributions.

$$D_{\text{TV}}(p_D, p_\theta) = \frac{1}{2} \sum_x |p_D(x) - p_\theta(x)|$$

Most comparison methods can be separated into two kinds: those that compare the density of the distributions ($p_\theta(x)$ vs $p_D(x)$), and those that compare the values sampled from them. These two kinds of approaches have different behavior and trade-offs.

Q2: Given we want to use them as an optimisation objective, what are the caveats to keep in mind about these two kinds?

When using distribution comparison metrics as optimization objectives, it is crucial to understand the limitations of the two kind of metrics **density-based** (KL divergence, JS divergence, Cross-entropy) and **sample-based** (Wasserstein distance, Maximum Mean Discrepancy (MMD)):

- **Density of the distribution metrics** :

Density-based metrics compare the **probability values** of the model and data distributions, such as $p_\theta(x)$ / $p_D(x)$, using divergences like KL or JS.

The main caveat of this approach is that it **requires access to the density function of the data distribution**, which is rarely available in practice, we usually only have samples from the data. Additionally, these metrics rely on the model having a **tractable likelihood function** to compute $p_\theta(x)$, which excludes many powerful generative models.

- **Values samples metrics**

Sample-based metrics operate by comparing **samples** drawn from the model and the data, making them applicable to **implicit generative models** where density evaluation is intractable.

Since the metric relies on **finite samples**, the gradient estimates tend to be **noisy**, which can make optimization difficult or slow. Moreover, many sample-based objectives are known for **training instability**, often requiring regularization tricks or architectural constraints to prevent problems like mode collapse or oscillations. Finally, some of these metrics, such as the Wasserstein distance, involve computationally **expensive approximations**, which can become a bottleneck during training, especially in high-dimensional settings.

In this work, we will focus on the most widely used generative models based on deep neural networks: Generative Adversarial Networks (GAN), Variational AutoEncoders (VAE), Flow Matching and Diffusion Models, in order to compare them and understand their strengths and weaknesses.

1.1 Dataset

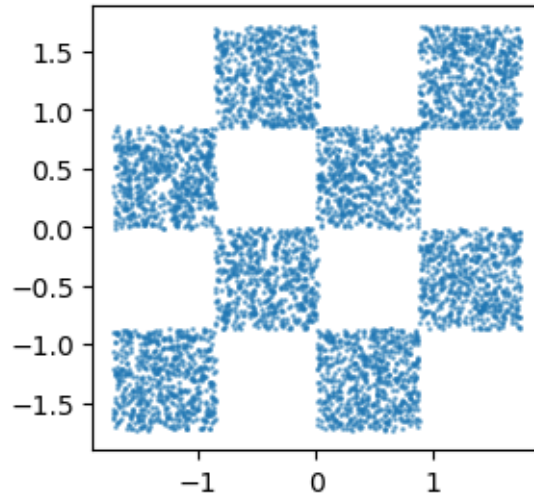
Checkerboard (prioritize this dataset)

```
[1]: import torch
import math
from sklearn import datasets
import numpy as np
import matplotlib.pyplot as plt

def generate_batch(num_datapoints: int) -> np.ndarray:
    """Checkerboard dataset."""
    x1 = np.random.rand(num_datapoints) * 4 - 2
    x2 = np.random.rand(num_datapoints) - np.random.randint(0, 2,
↳[num_datapoints]) * 2. + np.floor(x1) % 2
    data = np.stack([x1, x2]).T * 2
    data = (data - data.mean(axis=0)) / data.std(axis=0) # normalize
    return torch.from_numpy(data.astype(np.float32))

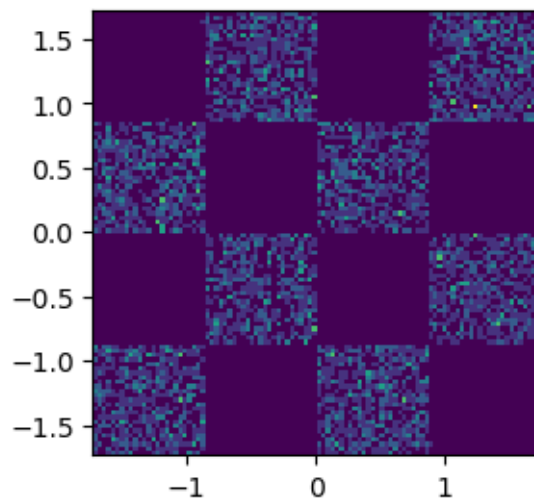
train_data = generate_batch(5000)

[2]: plt.figure(figsize=(3, 3))
plt.scatter(train_data.numpy()[:,0], train_data.numpy()[:,1], s=1.0, alpha=0.5)
plt.show()
```



It's also good to visualize the density as sometimes generative models overfit particular region of space. Once models are trained, when you sample from them, plot the two densities side-by-side to compare real & synthetic dataset.

```
[3]: plt.figure(figsize=(3, 3))
plt.hist2d(train_data.numpy()[:,0], train_data.numpy()[:,1], bins=100)
plt.show()
```



Moon (use this one only if your models struggle on checkerboard)

```
[4]: import torch
import math
```

```

from sklearn import datasets
import numpy as np

# Our dataset is mathematically defined, we can generate batches on the fly and
→ enjoy
# an infinite-size dataset
def generate_batch(batchlen):
    """This function generates a batch of length 'batchlen' from the dataset
    """
    data = datasets.make_moons(n_samples=batchlen, noise=0.05)[0].astype(np.
→float32)
    return torch.from_numpy(data)

```

Let's plot a large batch, to see what the dataset looks like.

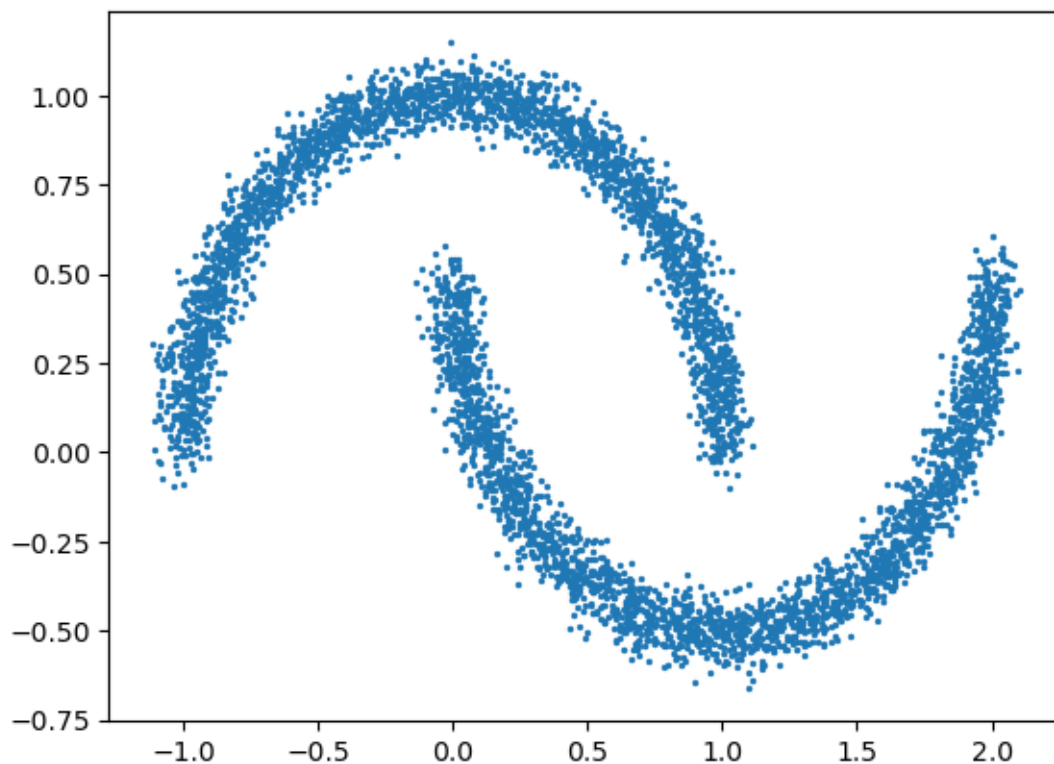
```

[5]: %matplotlib inline
import matplotlib.pyplot as plt

batch = generate_batch(5000)

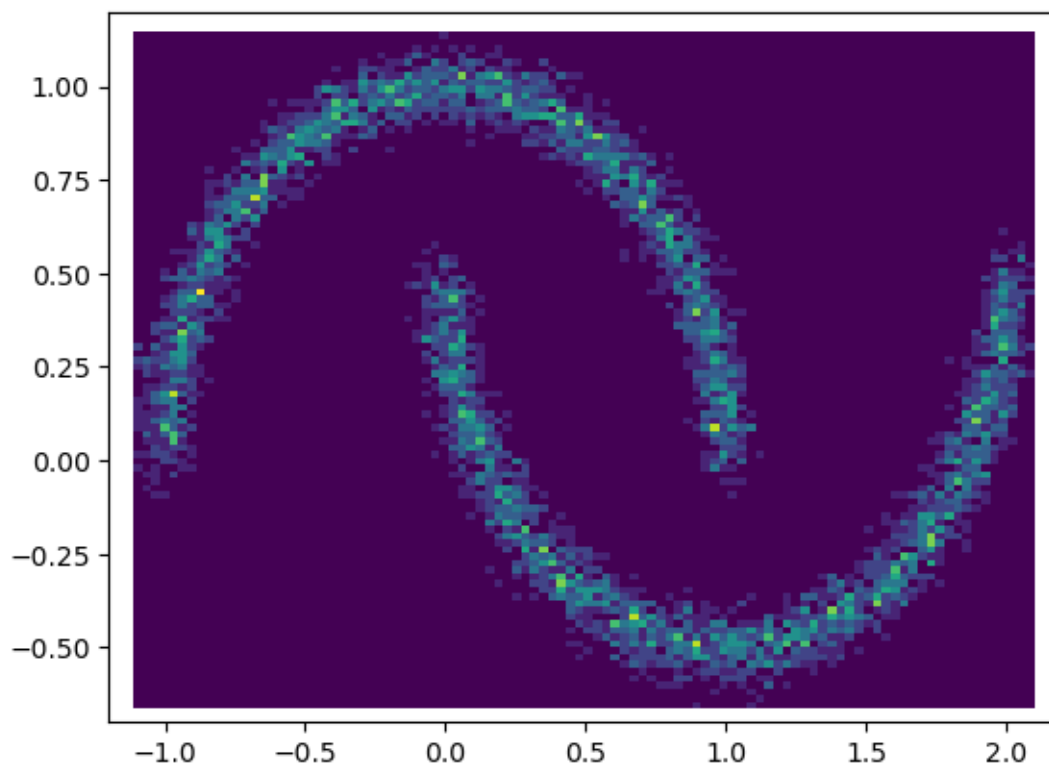
plt.scatter(batch[:,0], batch[:,1], s=2.0)
plt.show()

```



It's also good to visualize the density as sometimes generative models overfit particular areas. Once models are trained, when you sample from them, plot the two densities side-by-side to compare real & synthetic dataset.

```
[6]: plt.hist2d(batch.numpy()[ :,0], batch.numpy()[ :,1], bins=100)
plt.xlim([-1.2,2.2])
plt.ylim([-0.7,1.2])
plt.show()
```



1.2 Generative Adversarial Networks

GANs structure is based on modeling the distribution p_θ as a learned deterministic function applied to a standard noise. Sampling from it is thus done as follows: first, some noise is sampled from a standard N-dimensional Gaussian distribution: $\epsilon \sim \mathcal{N}(0; I)$, and then the output is computed as a deterministic function $x = f_\theta(\epsilon)$. The function f_θ is implemented as a neural network, θ representing its learned parameters.

Q3: What is, a priori, the impact of the choice of N, the dimension of the input noise ϵ ?

In a **GAN**, the generated distribution p_θ is obtained by applying a deterministic function f_θ , typically a neural network, to a random noise $\epsilon \sim \mathcal{N}(0, I)$ with dimension N.

The choice of N is crucial as it determines the model's ability to capture the complexity of the target distribution. If N is too small, the latent space becomes too restricted, making it difficult to model complex distributions. As a result, some structures of the true data distribution may become inaccessible to the generator. On the other hand, if N is too large, it increases the difficulty of learning: the generator must learn to ignore irrelevant directions in the latent space, which can slow convergence or lead to **overfitting**. Therefore, the choice of N involves a trade-off between the model's ability to represent a complex distribution and the stability of training. In practice, N is chosen empirically, and it should be large enough to allow for a compressed yet expressive representation of the data, while avoiding excessive increases in model complexity and optimization difficulty.

By construction, this generator structure only allows sampling the distribution p_θ , and does not allow the computation of the density $p_\theta(x)$ (at least not without strong assumptions on f_θ). Such a model thus needs a comparison method based on samples to be trained.

The smart idea of GANs is to instead use another neural network to model the objective. Another neural network is introduced: a classifier (that we call the discriminator) which is trained to distinguish examples from the dataset from examples generated by p_θ . The reasoning is as follows:

The discriminator D is trained using a classic classifier loss between the two classes defined as the samples generated by either p_D or p_θ . This way $D(x)$ can be interpreted as the probability that x came from the real dataset:

$$\mathcal{L}_D = \mathbb{E}_{p_D} [-\log D(x)] + \mathbb{E}_{p_\theta} [-\log (1 - D(x))]$$

From that, it can be shown that for the generator fixed, the optimal discriminator is given by $D(x) = \frac{p_D(x)}{p_\theta(x) + p_D(x)}$, and when reached its loss takes a specific value:

$$\mathcal{L}_D = 2 (\log 2 - JSD(p_\theta \| p_D))$$

, where JSD is the JS divergence used to measure the similarity of two distributions.

$$JSD(p_\theta \| p_D) = \frac{1}{2} \mathbb{E}_{p_D} \left[\log \left(\frac{2p_D}{p_D + p_\theta} \right) \right] + \frac{1}{2} \mathbb{E}_{p_\theta} \left[\log \left(\frac{2p_\theta}{p_D + p_\theta} \right) \right]$$

So, training the generator network to *maximize* the same loss would, assuming the discriminator is always trained to optimality, minimize the Jensen-Shannon Divergence between p_θ and p_D , and thus bring p_θ closer to p_D .

Q4: Can you anticipate a caveat of using the JSD as a training objective for the generator?

When the generator's distribution p_θ is very different from the data distribution p_D , the Jensen-Shannon divergence $JSD(p_\theta \| p_D)$ reaches its maximum value, which is $\log 2$. In this situation, the optimal discriminator perfectly distinguishes real from fake samples, and its output becomes saturated ($D(x) \rightarrow 1$ for real, $D(x) \rightarrow 0$ for fake). As a consequence, the generator receives almost no gradient data. This makes learning very difficult in early training stages, when p_θ is still far from p_D . Therefore, although minimizing JSD is a reasonable goal in theory, it leads to vanishing gradients in practice

when the two distributions are initially far apart, which is a major limitation of the original GAN formulation.

Having the generator trained to maximize $\mathcal{L}_D$ is equivalent to setting its training loss to $\mathcal{L}_G = \mathbb{E}_{p_\theta} \log(1 - D(x))$.

Q5: This loss only gives feedback to the generator on samples it generated, what problem may this cause?

This loss function for the generator only provides learning signal on the samples it generates. In particular, it encourages the generator to produce samples that the discriminator classifies as “real”. However, this can cause a major issue when the discriminator becomes too good early in training. In such cases, x becomes very close to 0 for all generated samples, and thus $\log(1 - D(x)) \approx 0$, leading to very small gradients for the generator. The generator gets almost no information about how to improve, and training may fail to progress.

We will now work on implementing a GAN on a simple toy problem, to get a feeling of its behavior and test our theoretical insights. For this we will use the `pytorch` library.

While a real problem would be generating images for example (each datapoint x would then be a different image), this is a kind of task that easily requires intensive CPU/GPU power, and image datasets are difficult to visualize from a geometric point of view (even small images contains hundreds of pixels, and nobody can visualize points in a 100-dimensional space). So instead we will focus on points in the plane: each datapoint x will actually be a couple of numbers (x_1, x_2) , and our target dataset will be a 2D two-moons shape with some noise.

We now need to define our two neural networks, the generator and the discriminator. The generator will take as input a value z sampled from a Gaussian prior, and output a value x (thus a couple (x_1, x_2)). The discriminator takes as input a value x , and is a binary classifier.

```
[7]: import torch.nn as nn
import torch.nn.functional as F

# Choose a value for the prior dimension
PRIOR_N = 2

# Define the generator
class Generator(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(PRIOR_N, 2)

    def __call__(self, z):
        return self.fc1(z)

    def generate(self, batchlen):
        z = torch.normal(torch.zeros(batchlen, PRIOR_N), 1.0)
        return self.__call__(z)
```

```

# Define the discriminator
class Discriminator(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(2, 1)

    def __call__(self, x):
        return torch.sigmoid(self.fc1(x))

```

```

[ ]: class Generator(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(PRIOR_N, 64),
            nn.ReLU(),
            nn.Linear(64, 64),
            nn.ReLU(),
            nn.Linear(64, 2)
        )

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

    def generate(self, batchlen):
        z = torch.randn(batchlen, PRIOR_N)
        return self.forward(z)

class Discriminator(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(2, 64),
            nn.ReLU(),
            nn.Linear(64, 64),
            nn.ReLU(),
            nn.Linear(64, 1),
            nn.Sigmoid()
        )

    def forward(self, x):
        return self.net(x)

```

With these classes in shape, now is only needed the training loop. To stick with the mathematical GAN framework, we should train the discriminator until convergence between each training step of the generator. This is not practical for two reasons: first it takes a lot of time, and second if the discriminator is too good, it will generate vanishing gradients to the generator (as seen in **Q4**).

We will thus train the discriminator a fixed number of times between each training iteration of the generator.

```
[ ]: # Choose a value for the prior dimension
PRIOR_N = 10

losses_gen = []
losses_disc = []
steps = []

# Number of times to train the discriminator between two generator steps
TRAIN_RATIO = 1
# Total number of training iterations for the generator
N_ITER = 20001
# Batch size to use
BATCHLEN = 128
# Learning rate
LR = 0.001

generator = Generator()
discriminator = Discriminator()

# Optimizers
optim_gen = torch.optim.Adam(generator.parameters(), lr=LR, betas=(0.5, 0.9))
optim_disc = torch.optim.Adam(discriminator.parameters(), lr=LR, betas=(0.5, 0.
→9))

# Training loop
for i in range(N_ITER):
    # Train the discriminator
    for _ in range(TRAIN_RATIO):
        discriminator.zero_grad()

        real_batch = generate_batch(BATCHLEN)
        real_labels = torch.ones(BATCHLEN, 1) # Label real data as 1

        fake_batch = generator.generate(BATCHLEN)
        fake_labels = torch.zeros(BATCHLEN, 1) # Label fake data as 0

        # == COMPUTE THE DISCRIMINATOR LOSS HERE ==
        real_loss = F.binary_cross_entropy(discriminator(real_batch),
→real_labels)
        fake_loss = F.binary_cross_entropy(discriminator(fake_batch),
→fake_labels)
        disc_loss = real_loss + fake_loss
```

```

        disc_loss.backward()
        optim_disc.step()

# Train the generator
        generator.zero_grad()

        fake_batch = generator.generate(BATCHLEN)

        # == COMPUTE THE GENERATOR LOSS HERE
        gen_loss = F.binary_cross_entropy(discriminator(fake_batch), torch.
→ones(BATCHLEN, 1))

        gen_loss.backward()
        optim_gen.step()

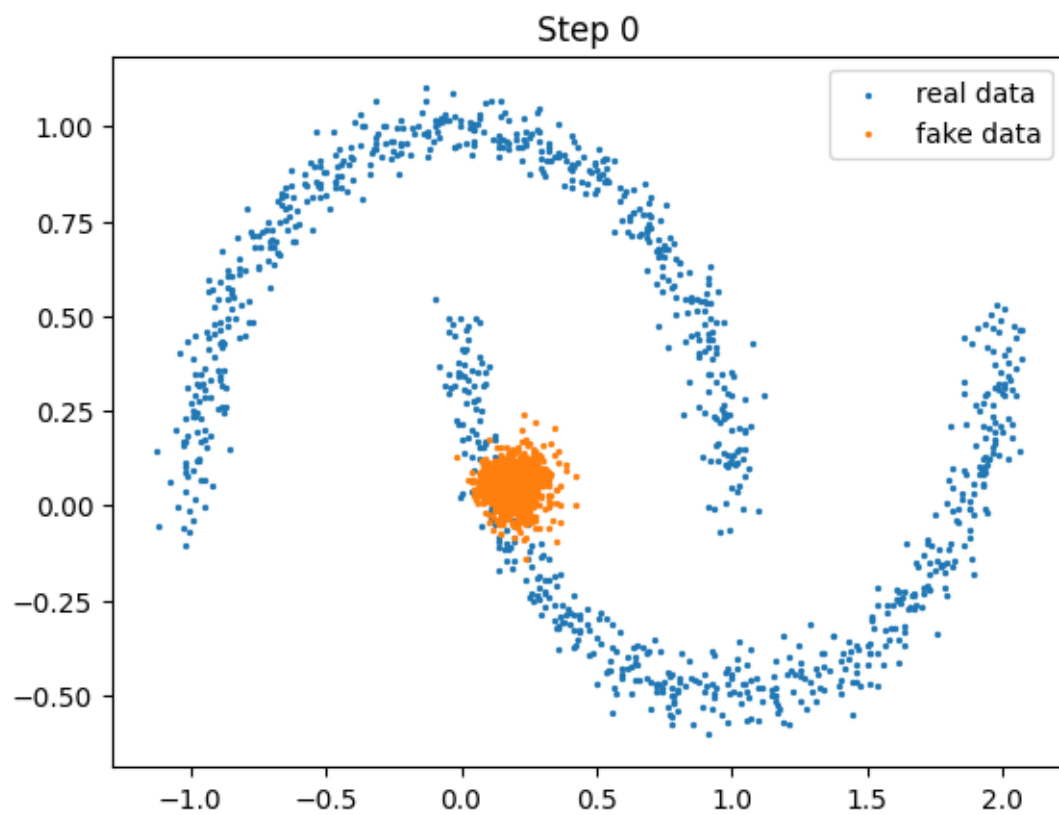
        if i % 100 == 0:
            losses_gen.append(gen_loss.item())
            losses_disc.append(disc_loss.item())
            steps.append(i)

        if i % 1000 == 0:
            print('Step {}: discriminator loss: {:.3e}, generator loss: {:.3e}'.
→format(i, float(disc_loss), float(gen_loss)))
            # plot the result
            real_batch = generate_batch(1024)
            fake_batch = generator.generate(1024).detach()
            plt.scatter(real_batch[:, 0], real_batch[:, 1], s=2.0, label='real data')
            plt.scatter(fake_batch[:, 0], fake_batch[:, 1], s=2.0, label='fake data')
            plt.legend()
            plt.title(f"Step {i}")
            plt.show()

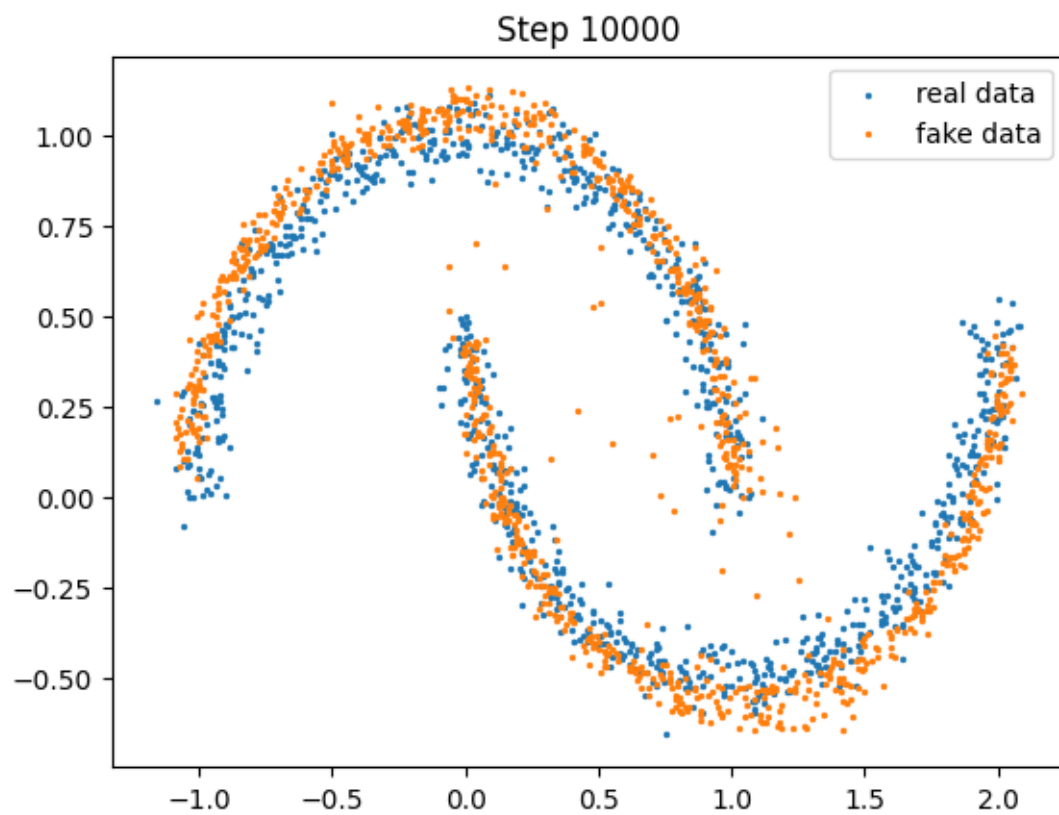
# Plot losses
plt.figure(figsize=(8, 5))
plt.plot(steps, losses_gen, label="Generator Loss")
plt.plot(steps, losses_disc, label="Discriminator Loss")
plt.xlabel("Training Steps")
plt.ylabel("Loss")
plt.title("Losses over time")
plt.legend()
plt.grid(True)
plt.show()

```

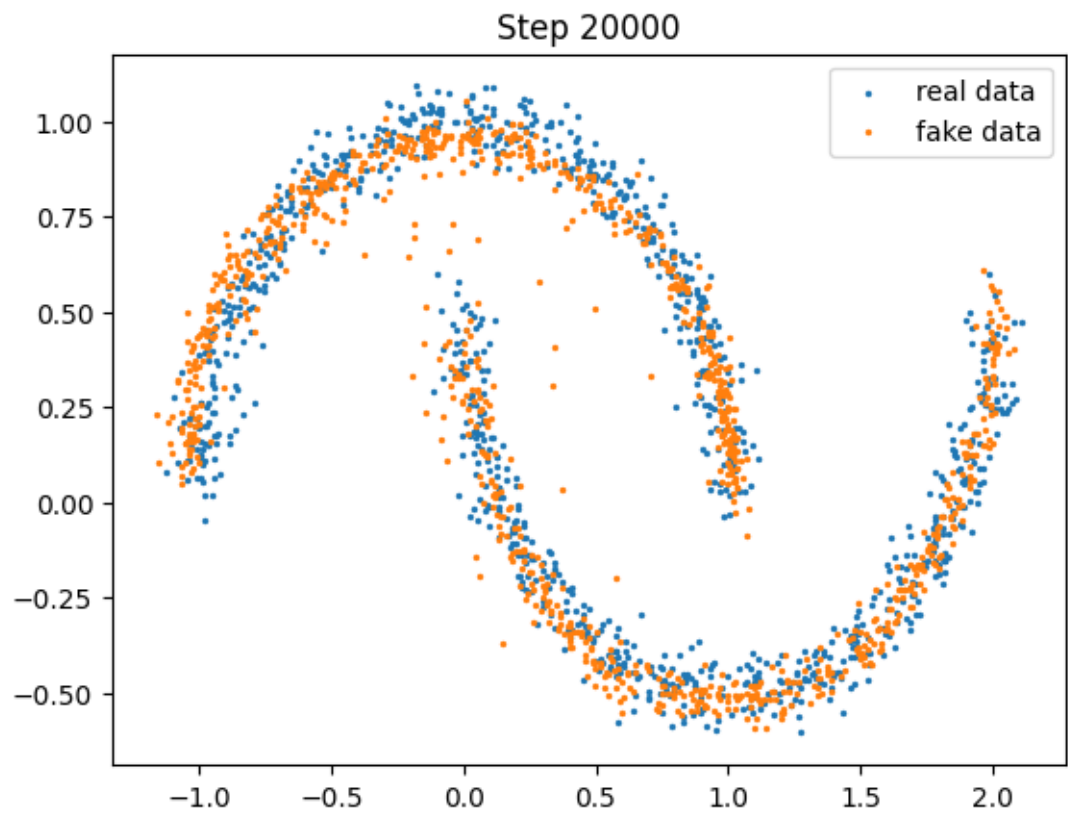
Step 0: discriminator loss: 1.365e+00, generator loss: 6.405e-01



Step 10000: discriminator loss: 1.361e+00, generator loss: 7.133e-01



Step 20000: discriminator loss: 1.336e+00, generator loss: 7.357e-01



Complete the previous code and train your model.

Depending on your choice of parameters, the training may not go well at all, with the generator completely collapsing quickly at the beginning of the training. It has been observed by the literature that the generator's loss $\mathcal{L}_G = \mathbb{E}_{p_\theta} \log(1 - D(x))$ is often to blame.

Q6: Why could we anticipate that this loss could cause this?

(Write your answer here, consider about gradient vanishing when discriminator is trained very well)

This loss encourages the generator to produce outputs that **fool** the discriminator, but it can lead to **instability**, especially at the beginning of training. In the early stages, the discriminator is often too powerful, and the generator may fail to provide meaningful gradients, leading to a situation where the generator's loss stagnates or diverges. This is often referred to as the **vanishing gradient problem**, where the generator receives very small gradients, making it difficult for the network to learn effectively. Moreover, the $\log(1 - D(x))$ term is not ideal in these situations because, when the discriminator is confident ($D(x)$ is close to 1), the gradient becomes very small, which stops learning. This causes the generator to quickly collapse, as it fails to learn how to generate high-quality samples that fool the discriminator.

This issue is solved by replacing the generator loss by an alternative loss: $\mathcal{L}_G = \mathbb{E}_{p_\theta} [-\log D(x)]$ to avoid gradient vanishing.

Q7: Inspect the impact of these different factors:

- depth / width of the generator network
- depth / width of the discriminator network
- impact of TRAIN_RATIO

```
[ ]: class Generator(nn.Module):
    def __init__(self, prior_n, depth, width):
        super().__init__()
        layers = []
        layers.append(nn.Linear(prior_n, width))
        layers.append(nn.ReLU())

        for _ in range(depth - 1):
            layers.append(nn.Linear(width, width))
            layers.append(nn.ReLU())

        layers.append(nn.Linear(width, 2))
        self.net = nn.Sequential(*layers)

    def forward(self, z):
        return self.net(z)
```

```

def generate(self, batchlen):
    z = torch.randn(batchlen, PRIOR_N)
    return self.forward(z)

class Discriminator(nn.Module):
    def __init__(self, depth, width):
        super().__init__()
        layers = []
        layers.append(nn.Linear(2, width))
        layers.append(nn.ReLU())

        for _ in range(depth - 1):
            layers.append(nn.Linear(width, width))
            layers.append(nn.ReLU())

        layers.append(nn.Linear(width, 1))
        layers.append(nn.Sigmoid())
        self.net = nn.Sequential(*layers)

    def forward(self, x):
        return self.net(x)

# Train
def train_gan(prior_n=10, gen_depth=3, gen_width=64, disc_depth=3,
    ↪disc_width=64, train_ratio=5, n_iter=20001, batchlen=128, lr=0.001):
    losses_gen = []
    losses_disc = []
    steps = []

    generator = Generator(prior_n, gen_depth, gen_width)
    discriminator = Discriminator(disc_depth, disc_width)

    optim_gen = torch.optim.Adam(generator.parameters(), lr=lr, betas=(0.5, 0.9))
    optim_disc = torch.optim.Adam(discriminator.parameters(), lr=lr, betas=(0.5,
    ↪0.9))

    for i in range(n_iter):
        # Train the discriminator
        for _ in range(train_ratio):
            discriminator.zero_grad()

            real_batch = generate_batch(batchlen)
            real_labels = torch.ones(batchlen, 1) # Label real data as 1

            fake_batch = generator.generate(batchlen)
            fake_labels = torch.zeros(batchlen, 1) # Label fake data as 0

```

```

        # Discriminator loss (real + fake)
        real_loss = F.binary_cross_entropy(discriminator(real_batch),
↪real_labels)
        fake_loss = F.binary_cross_entropy(discriminator(fake_batch),
↪fake_labels)
        disc_loss = real_loss + fake_loss
        disc_loss.backward()
        optim_disc.step()

    # Train the generator
    generator.zero_grad()
    fake_batch = generator.generate(batchlen)
    gen_loss = F.binary_cross_entropy(discriminator(fake_batch), torch.
↪ones(batchlen, 1))
    gen_loss.backward()
    optim_gen.step()

    if i % 100 == 0:
        losses_gen.append(gen_loss.item())
        losses_disc.append(disc_loss.item())
        steps.append(i)

    # if i % 1000 == 0:
    #     print(f'Step {i}: discriminator loss: {disc_loss:.3e}, generator
↪loss: {gen_loss:.3e}')

    #     real_batch = generate_batch(1024)
    #     fake_batch = generator.generate(1024).detach()

    # plt.scatter(real_batch[:, 0], real_batch[:, 1], s=2.0, label='real
↪data')
    # plt.scatter(fake_batch[:, 0], fake_batch[:, 1], s=2.0, label='fake
↪data')

    # plt.legend()
    # plt.title(f"Step {i}")
    # plt.show()

    print(f'Step {i}: discriminator loss: {disc_loss:.3e}, generator loss:
↪{gen_loss:.3e}')

    real_batch = generate_batch(1024)
    fake_batch = generator.generate(1024).detach()

    fig, axs = plt.subplots(1, 2, figsize=(10, 4))
    axs[0].scatter(real_batch[:, 0], real_batch[:, 1], s=2.0, label='real data')
    axs[0].scatter(fake_batch[:, 0], fake_batch[:, 1], s=2.0, label='fake data')

```

```

axs[0].legend()
axs[0].set_title(f"Generated vs Real (Step {i})")

axs[1].plot(steps, losses_gen, label="Generator Loss")
axs[1].plot(steps, losses_disc, label="Discriminator Loss")
axs[1].set_xlabel("Training Steps")
axs[1].set_ylabel("Loss")
axs[1].set_title("Losses over Time")
axs[1].legend()
axs[1].grid(True)

plt.tight_layout()
plt.show()

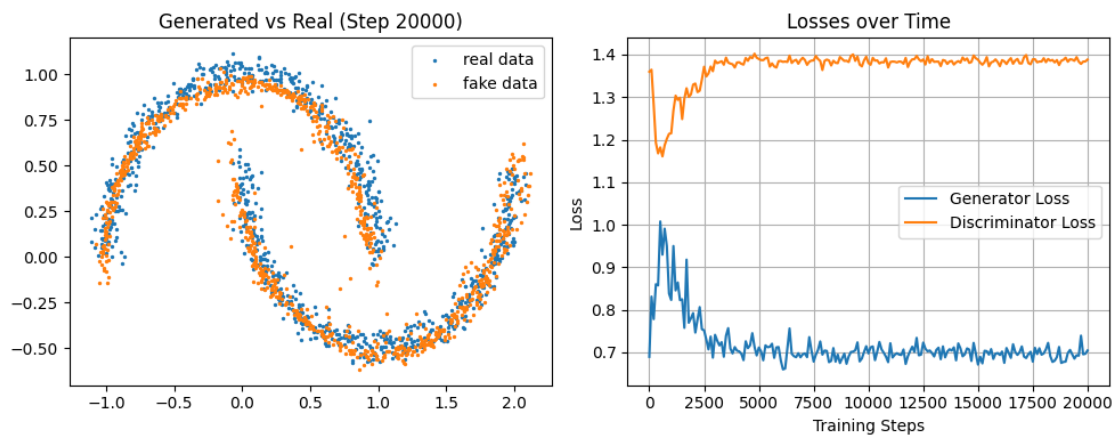
```

```

[64]: train_gan(prior_n=10, gen_depth=2, gen_width=32, disc_depth=2, disc_width=32,
→train_ratio=5)

```

Step 20000: discriminator loss: 1.388e+00, generator loss: 7.041e-01

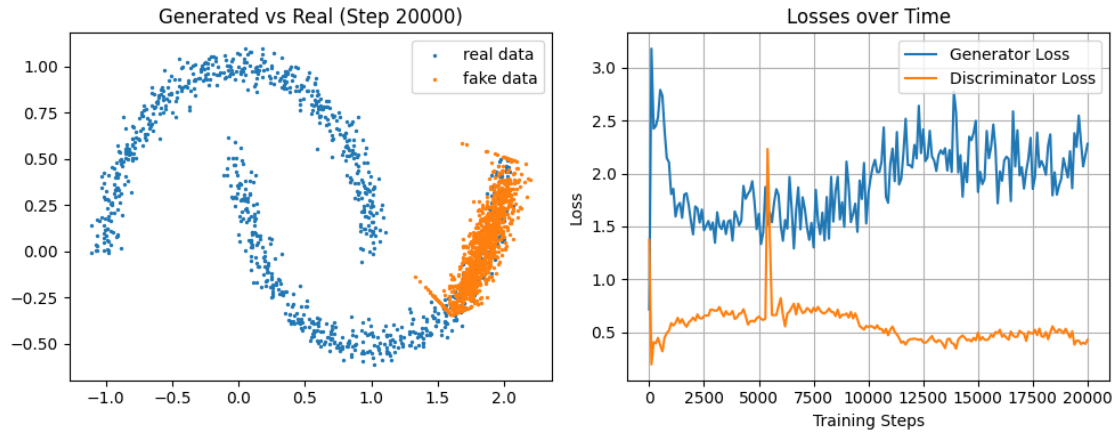


```

[63]: train_gan(prior_n=10, gen_depth=2, gen_width=2, disc_depth=2, disc_width=32,
→train_ratio=5)

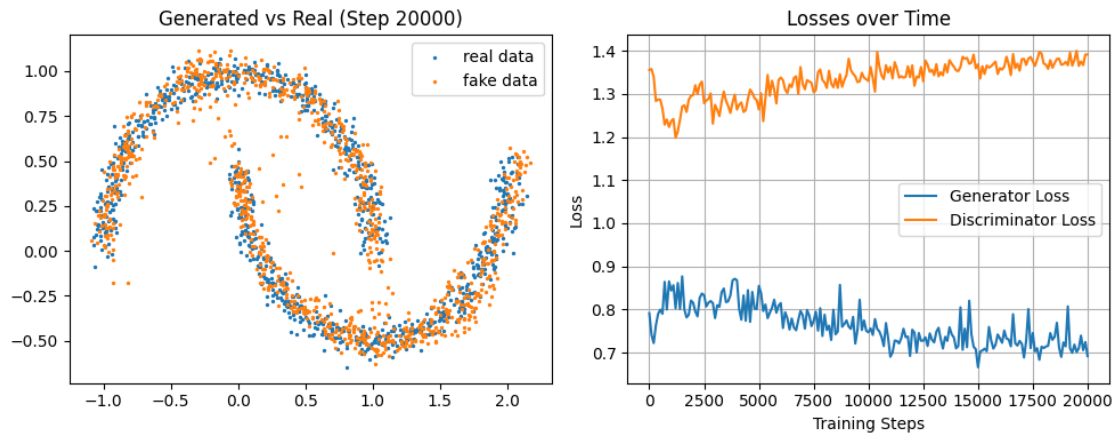
```

Step 20000: discriminator loss: 4.278e-01, generator loss: 2.280e+00



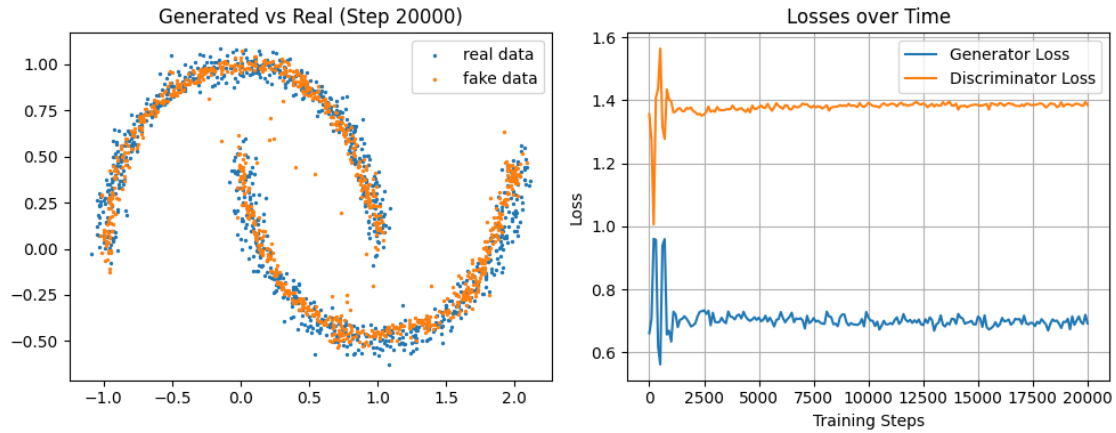
```
[62]: train_gan(prior_n=10, gen_depth=1, gen_width=32, disc_depth=1, disc_width=32,
→train_ratio=5)
```

Step 20000: discriminator loss: 1.392e+00, generator loss: 6.919e-01



```
[61]: train_gan(prior_n=10, gen_depth=2, gen_width=32, disc_depth=2, disc_width=32,
→train_ratio=1)
```

Step 20000: discriminator loss: 1.385e+00, generator loss: 6.922e-01

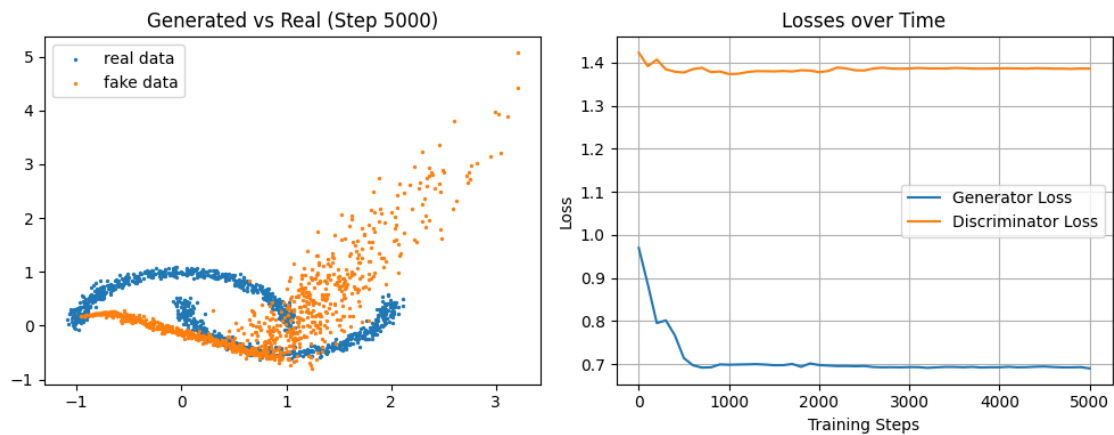


```
[60]: for tr_ratio in [1,5,8]:
      for depth_g in [1,2,4]:
        for depth_d in [1,2,4]:
          for width in [4,32,128]:
            print('Parameters: TRAIN_RATIO : {}, depth_generator : {},
↳depth_discriminator : {}, width : {}'.format(tr_ratio, depth_g, depth_d,
↳width))

            train_gan(prior_n=10, gen_depth=depth_g, gen_width=width,
↳disc_depth=depth_d, disc_width=width, train_ratio=tr_ratio, n_iter = 5001)
```

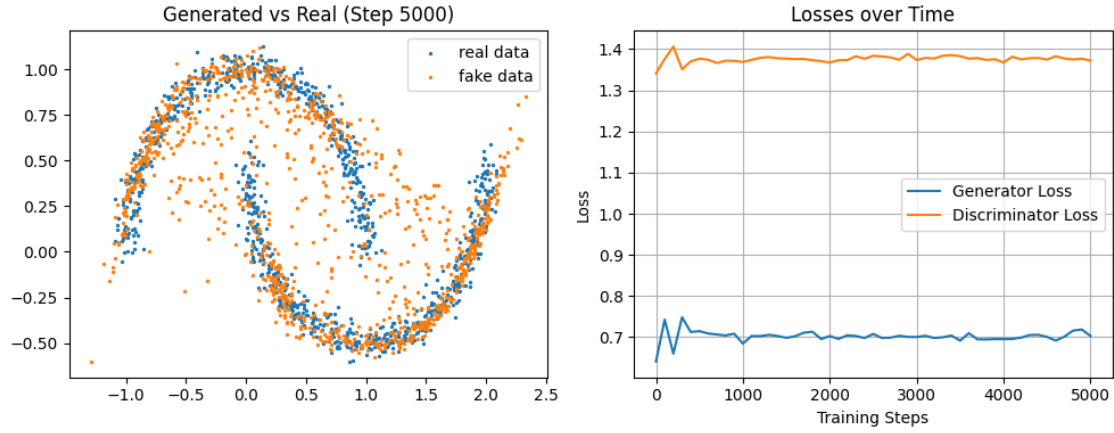
Parameters: TRAIN_RATIO : 1, depth_generator : 1, depth_discriminator : 1, width : 4

Step 5000: discriminator loss: 1.386e+00, generator loss: 6.904e-01



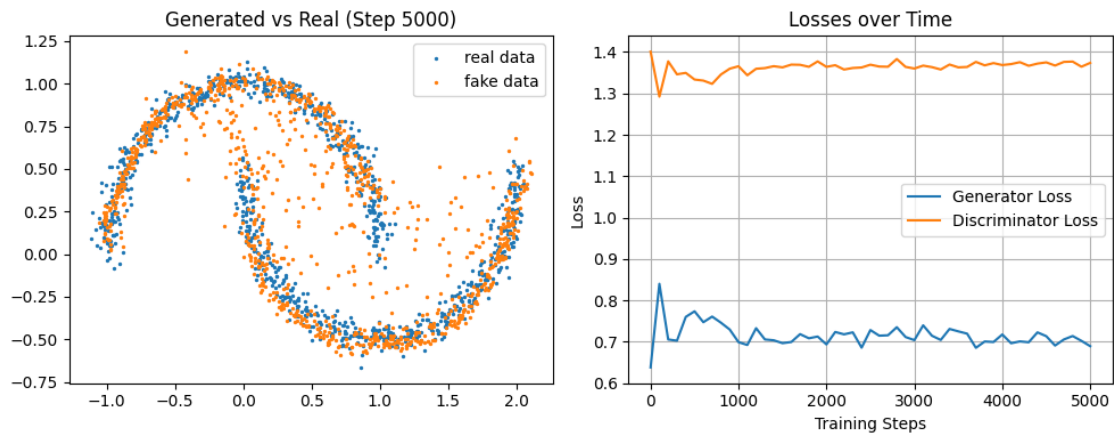
Parameters: TRAIN_RATIO : 1, depth_generator : 1, depth_discriminator : 1, width : 32

Step 5000: discriminator loss: 1.372e+00, generator loss: 7.030e-01



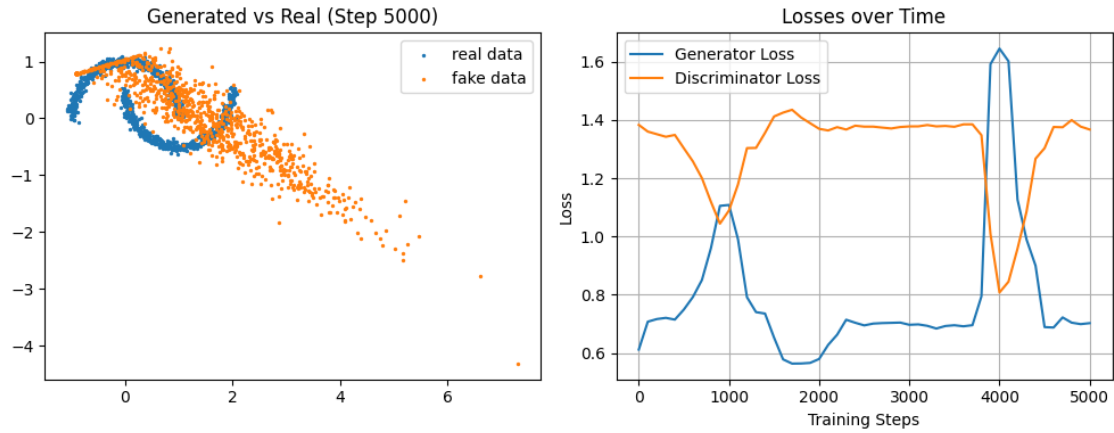
Parameters: TRAIN_RATIO : 1, depth_generator : 1, depth_discriminator : 1, width : 128

Step 5000: discriminator loss: 1.374e+00, generator loss: 6.892e-01



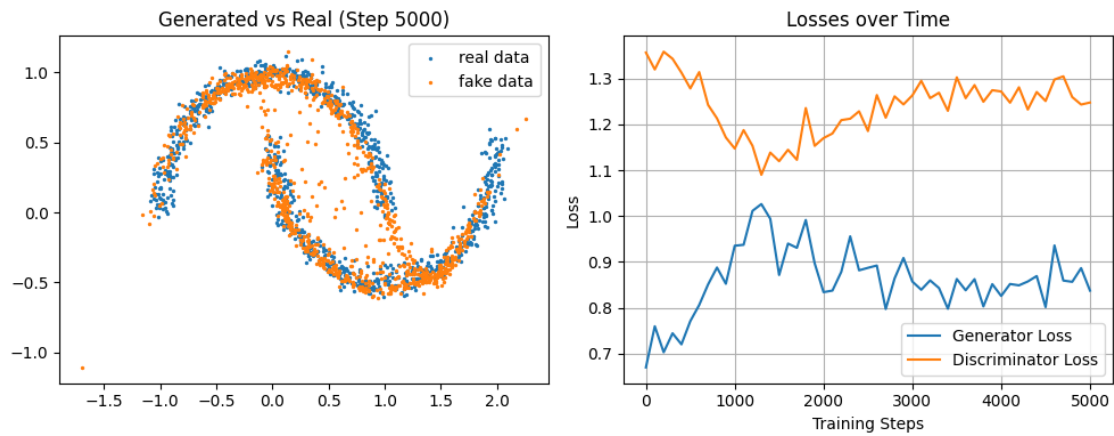
Parameters: TRAIN_RATIO : 1, depth_generator : 1, depth_discriminator : 2, width : 4

Step 5000: discriminator loss: 1.368e+00, generator loss: 7.026e-01



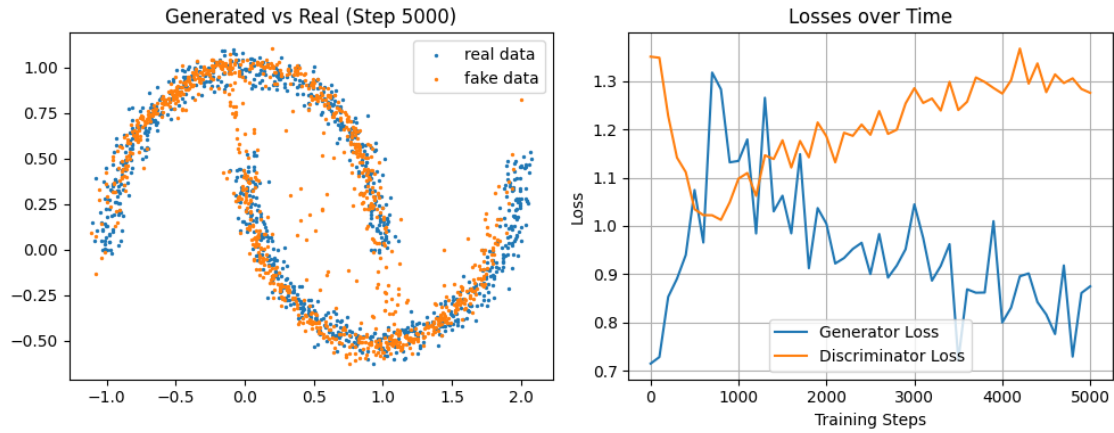
Parameters: TRAIN_RATIO : 1, depth_generator : 1, depth_discriminator : 2, width : 32

Step 5000: discriminator loss: 1.248e+00, generator loss: 8.376e-01



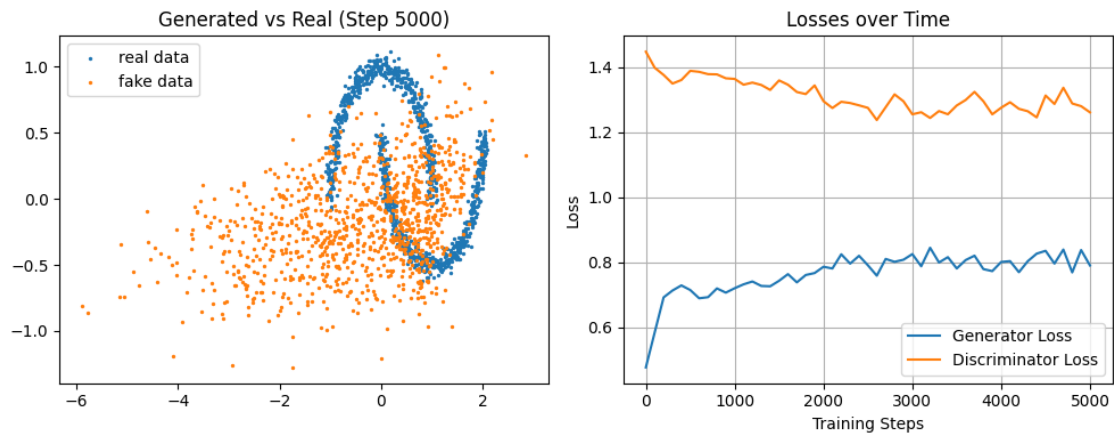
Parameters: TRAIN_RATIO : 1, depth_generator : 1, depth_discriminator : 2, width : 128

Step 5000: discriminator loss: 1.276e+00, generator loss: 8.750e-01



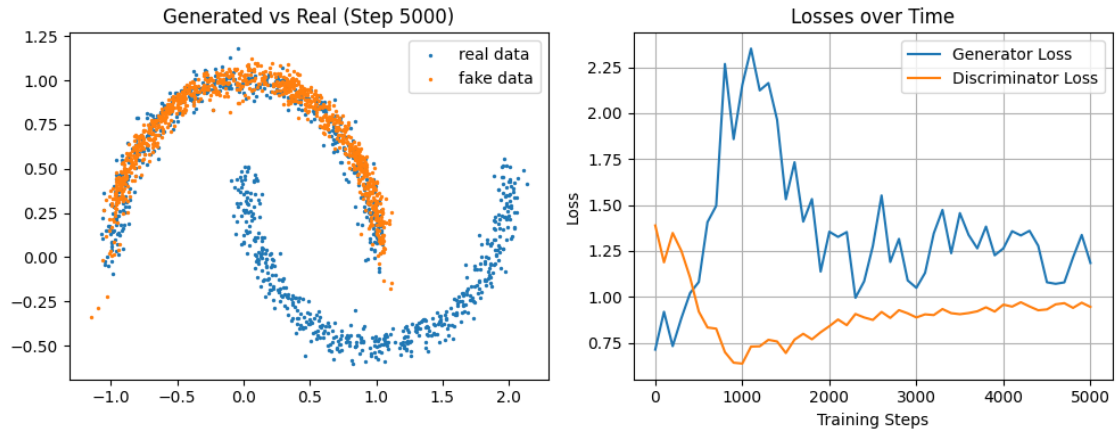
Parameters: TRAIN_RATIO : 1, depth_generator : 1, depth_discriminator : 4, width : 4

Step 5000: discriminator loss: 1.261e+00, generator loss: 7.901e-01



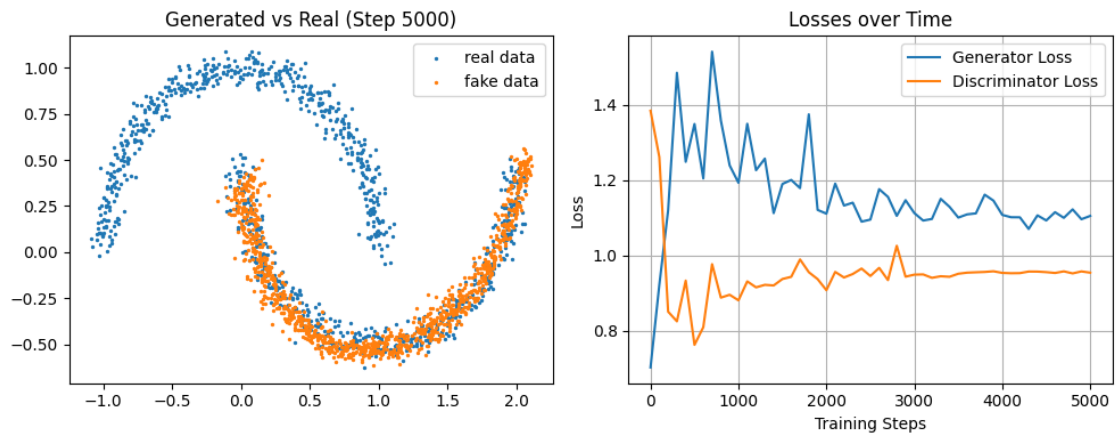
Parameters: TRAIN_RATIO : 1, depth_generator : 1, depth_discriminator : 4, width : 32

Step 5000: discriminator loss: 9.448e-01, generator loss: 1.184e+00



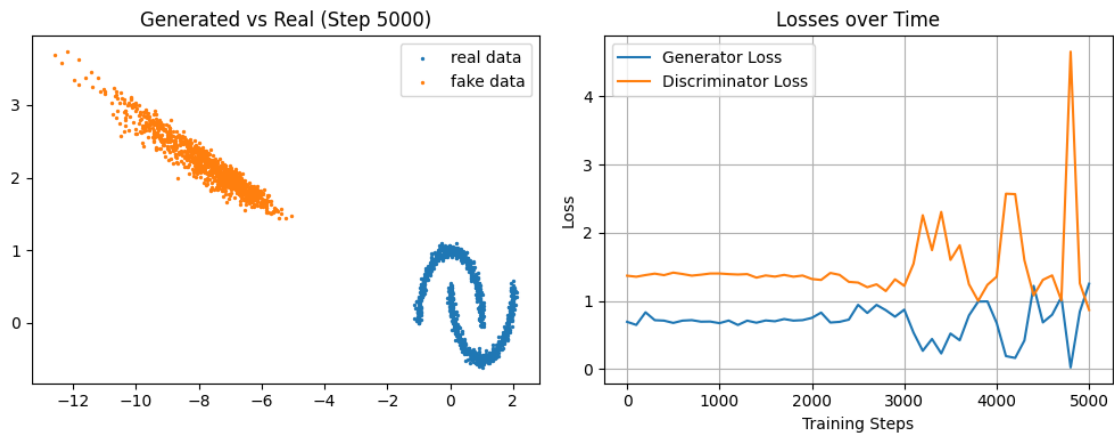
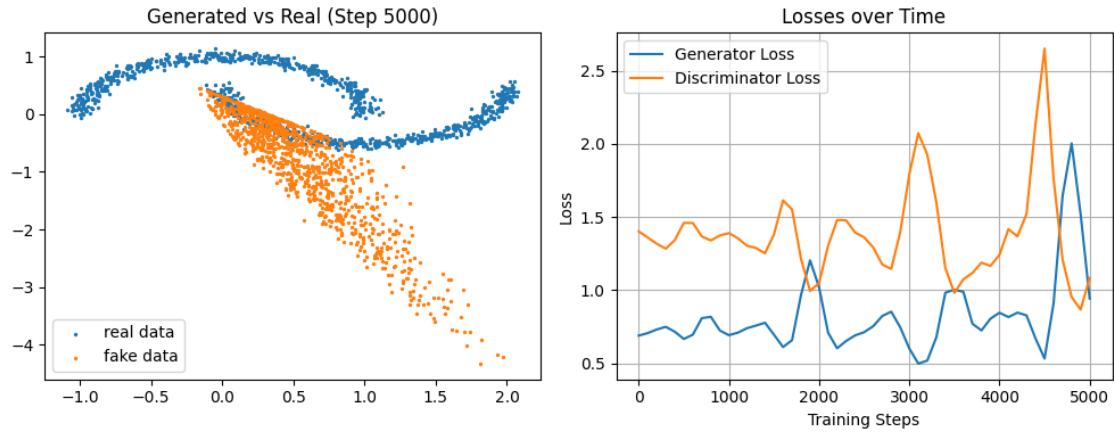
Parameters: TRAIN_RATIO : 1, depth_generator : 1, depth_discriminator : 4, width : 128

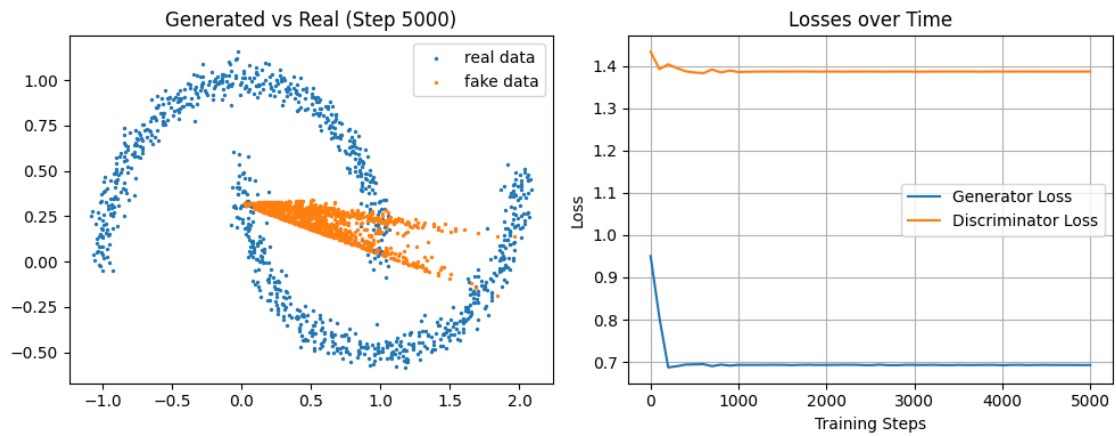
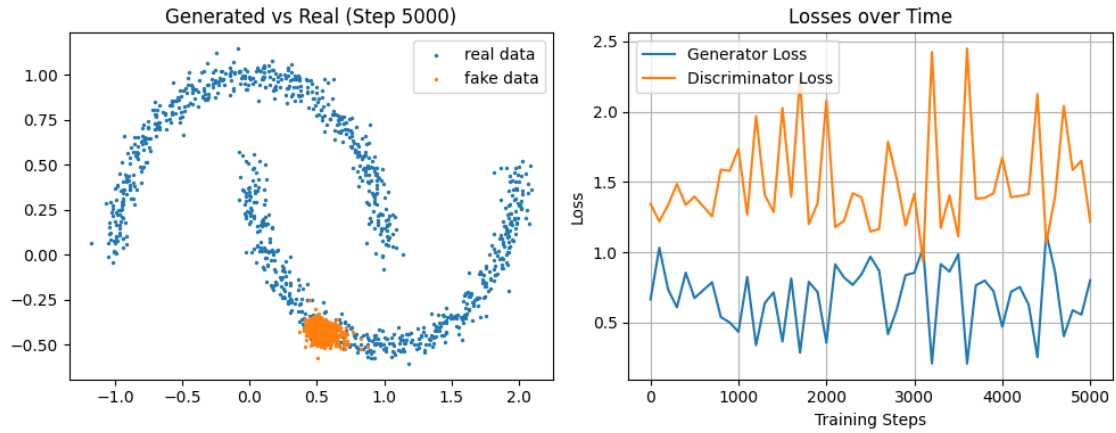
Step 5000: discriminator loss: 9.539e-01, generator loss: 1.105e+00

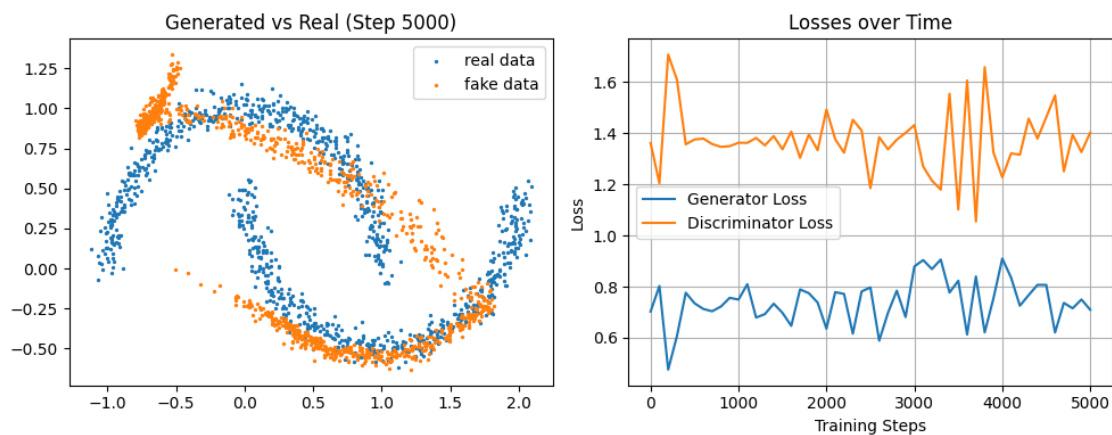


Parameters: TRAIN_RATIO : 1, depth_generator : 2, depth_discriminator : 1, width : 4

Step 5000: discriminator loss: 1.084e+00, generator loss: 9.427e-01

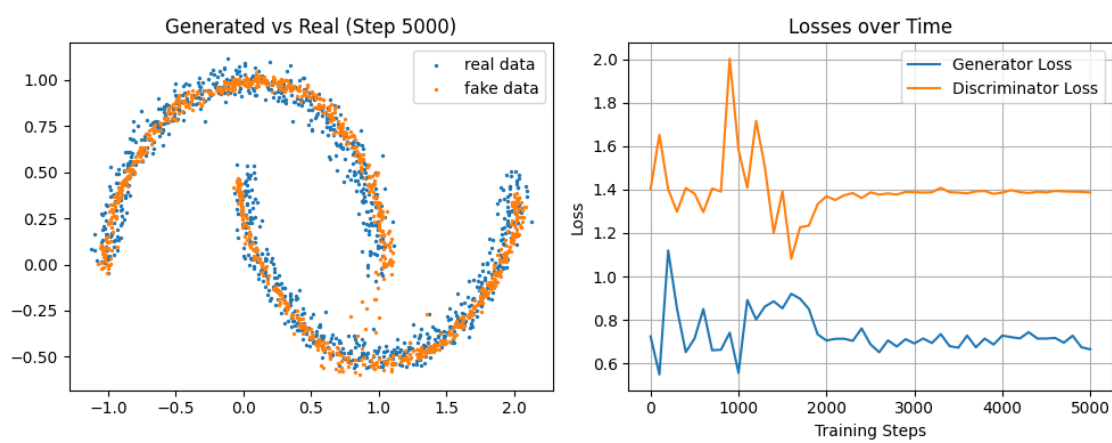






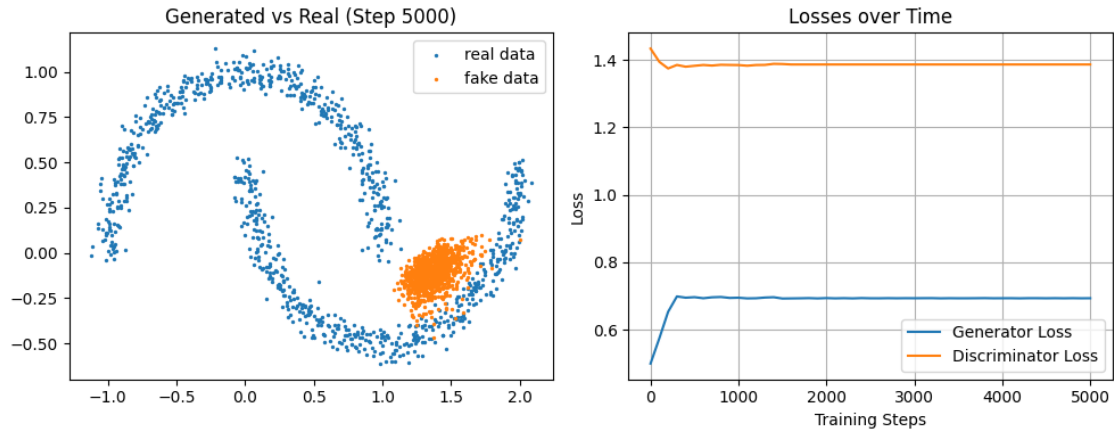
Parameters: TRAIN_RATIO : 1, depth_generator : 2, depth_discriminator : 2, width : 128

Step 5000: discriminator loss: 1.386e+00, generator loss: 6.648e-01



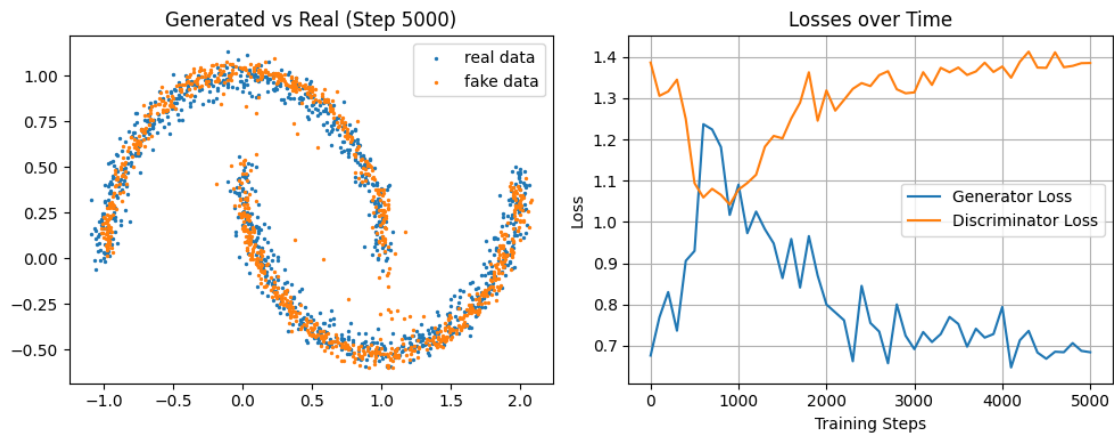
Parameters: TRAIN_RATIO : 1, depth_generator : 2, depth_discriminator : 4, width : 4

Step 5000: discriminator loss: 1.386e+00, generator loss: 6.932e-01



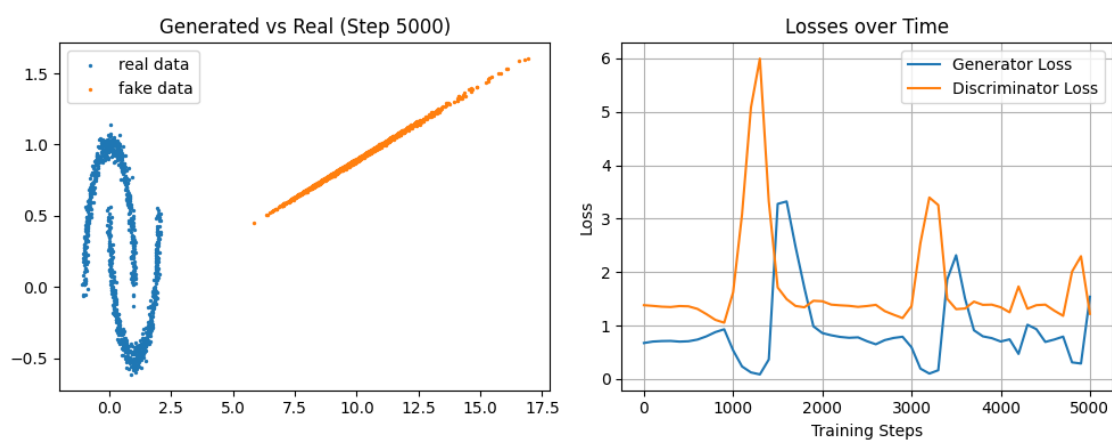
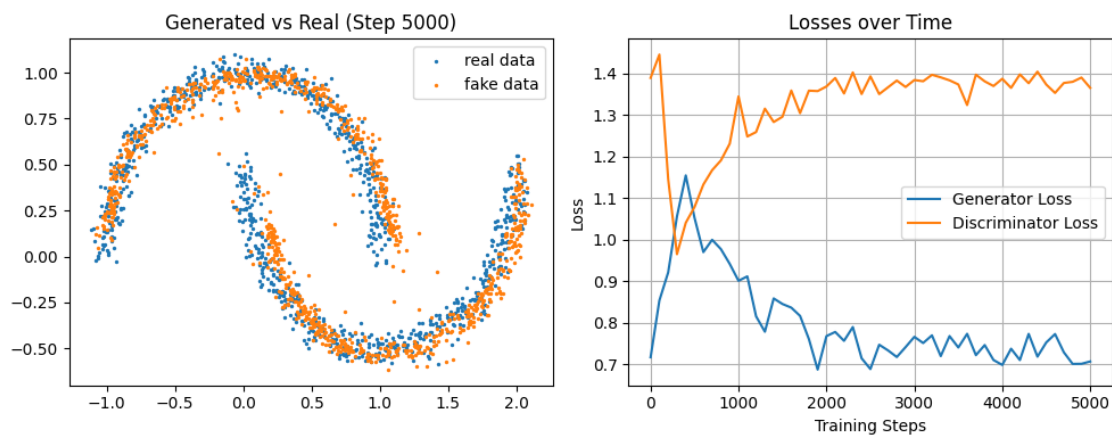
Parameters: TRAIN_RATIO : 1, depth_generator : 2, depth_discriminator : 4, width : 32

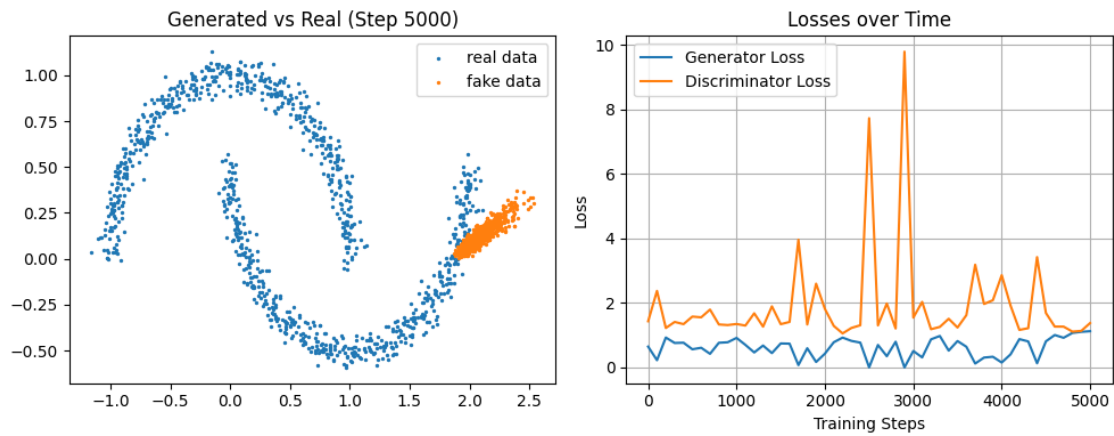
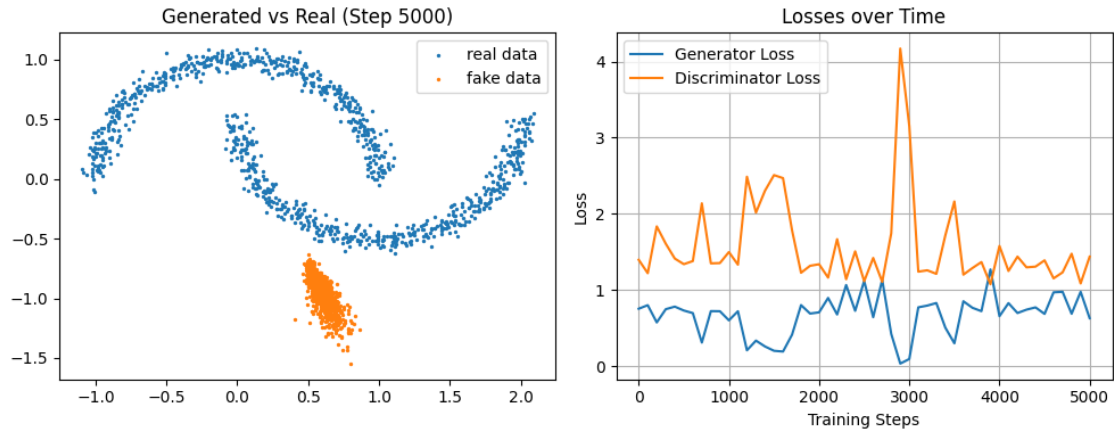
Step 5000: discriminator loss: 1.385e+00, generator loss: 6.841e-01

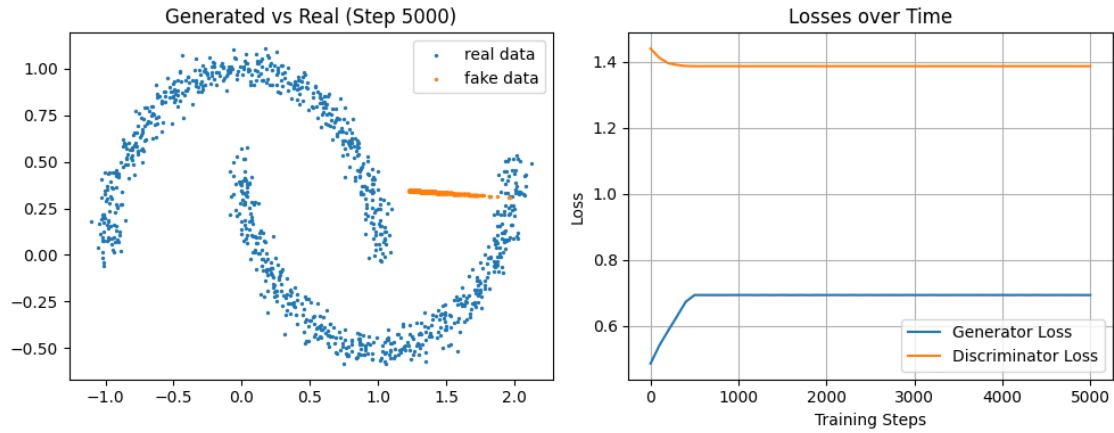


Parameters: TRAIN_RATIO : 1, depth_generator : 2, depth_discriminator : 4, width : 128

Step 5000: discriminator loss: 1.365e+00, generator loss: 7.065e-01

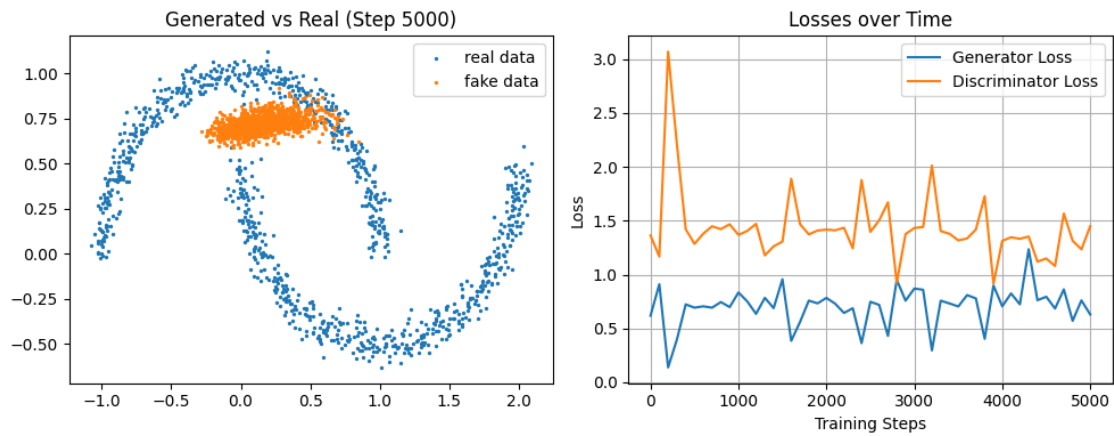






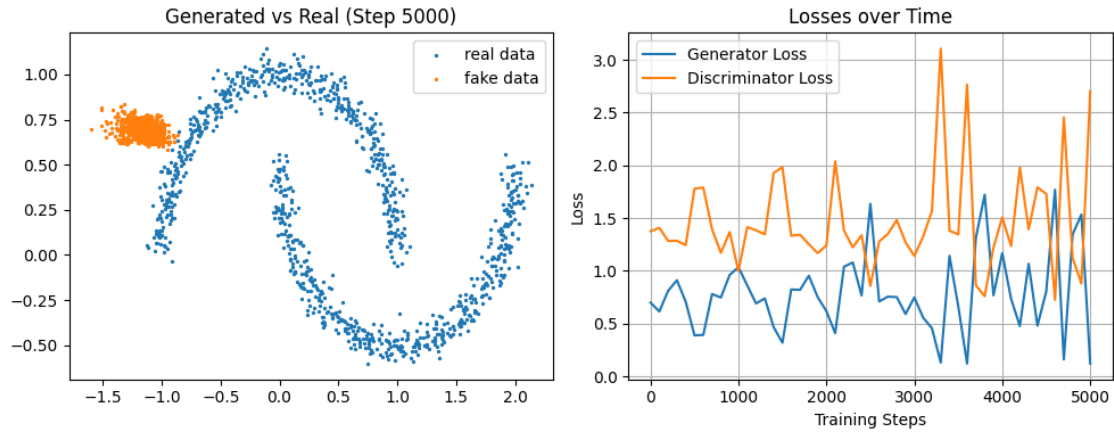
Parameters: TRAIN_RATIO : 1, depth_generator : 4, depth_discriminator : 2, width : 32

Step 5000: discriminator loss: 1.450e+00, generator loss: 6.302e-01



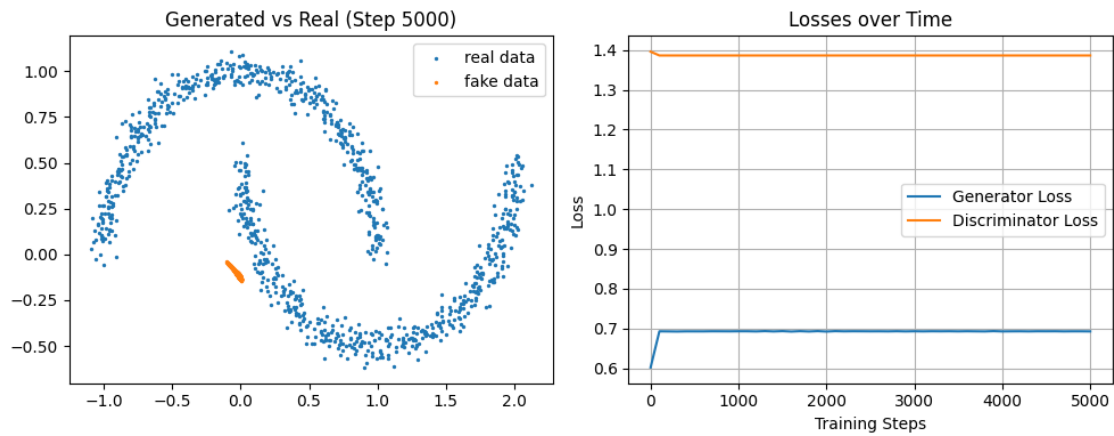
Parameters: TRAIN_RATIO : 1, depth_generator : 4, depth_discriminator : 2, width : 128

Step 5000: discriminator loss: 2.702e+00, generator loss: 1.216e-01



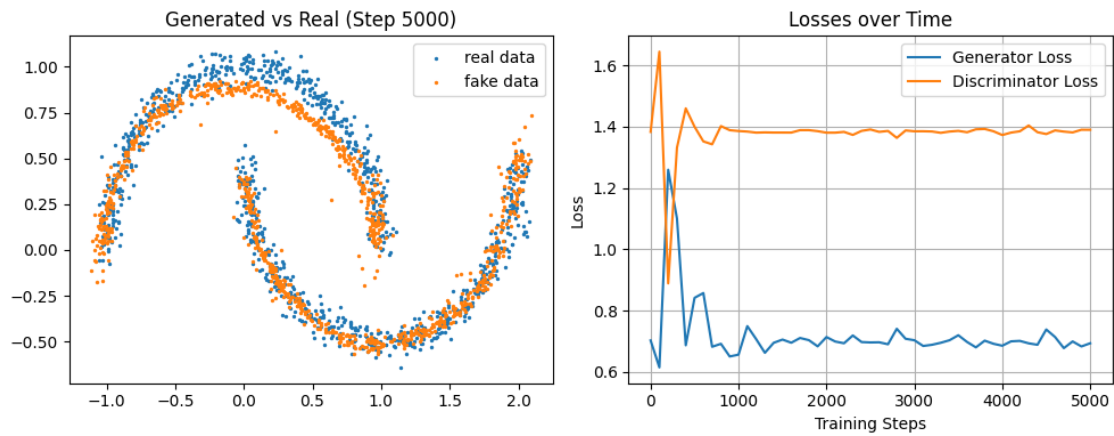
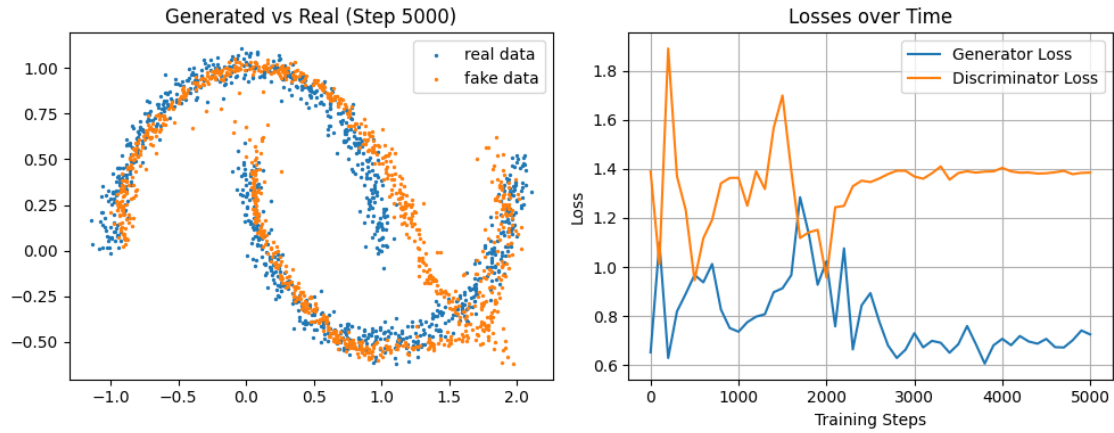
Parameters: TRAIN_RATIO : 1, depth_generator : 4, depth_discriminator : 4, width : 4

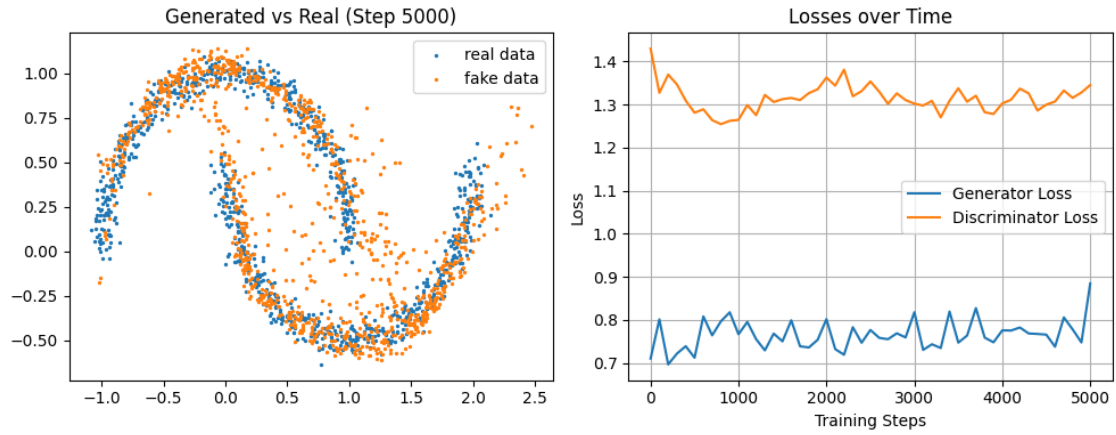
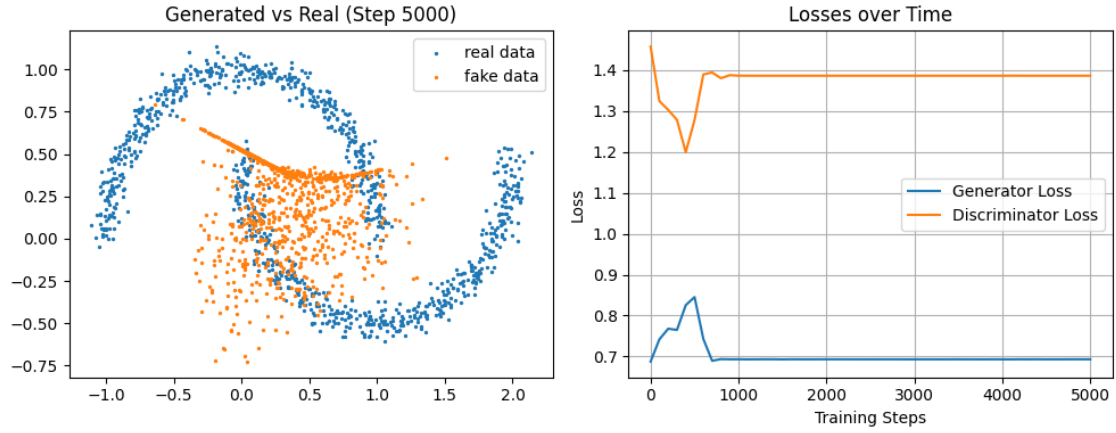
Step 5000: discriminator loss: 1.386e+00, generator loss: 6.929e-01

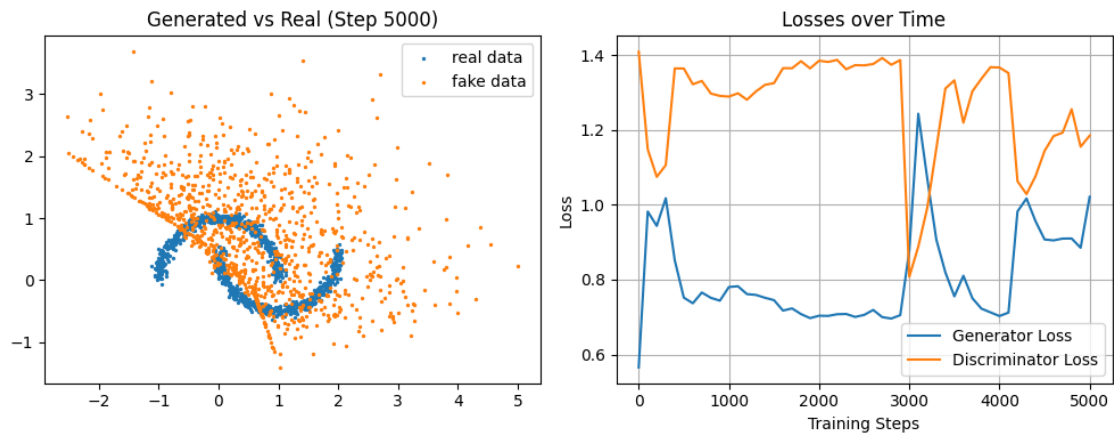
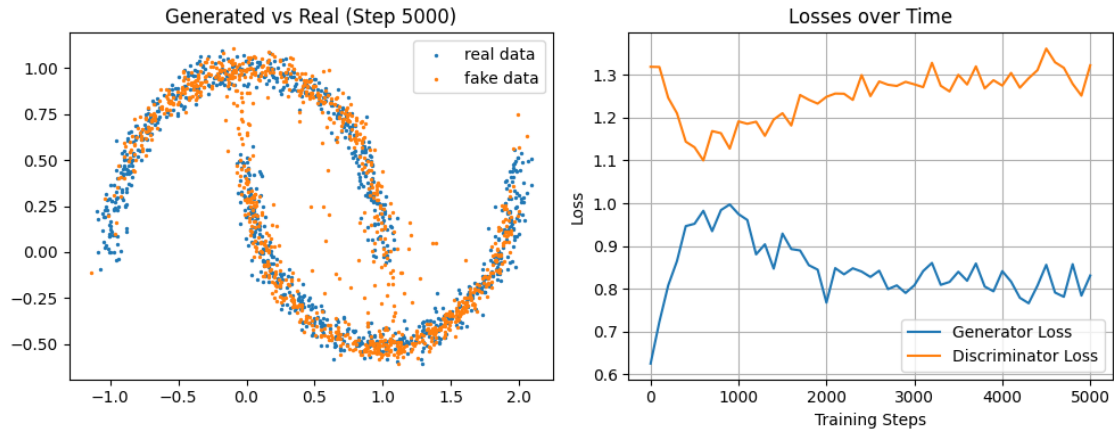


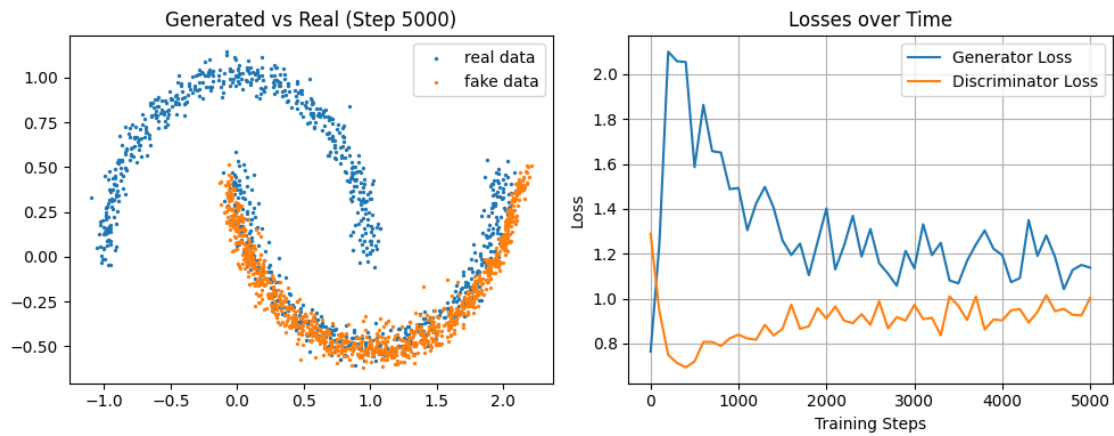
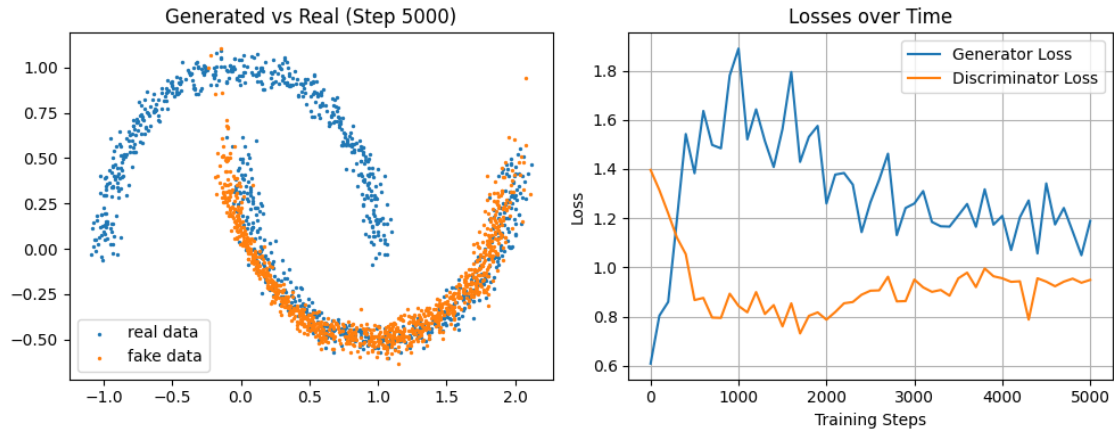
Parameters: TRAIN_RATIO : 1, depth_generator : 4, depth_discriminator : 4, width : 32

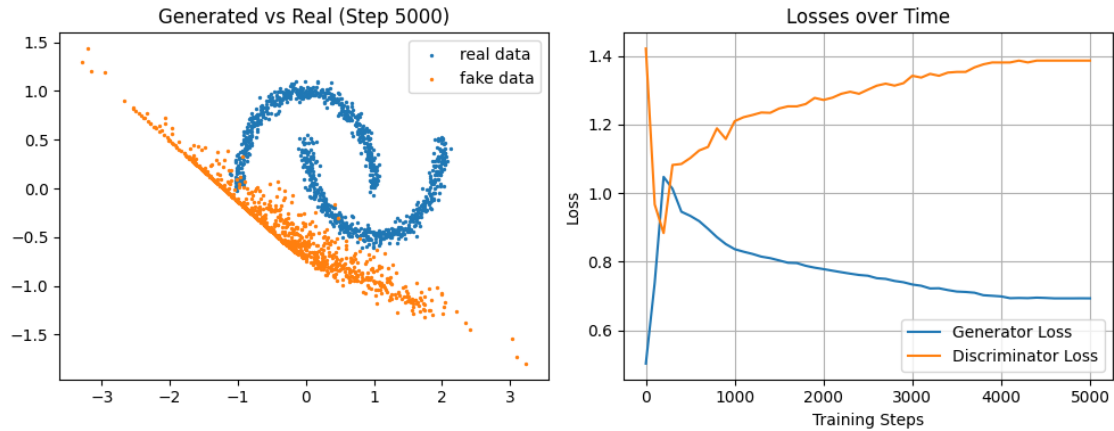
Step 5000: discriminator loss: 1.384e+00, generator loss: 7.250e-01











Parameters: TRAIN_RATIO : 5, depth_generator : 1, depth_discriminator : 4, width : 32

KeyboardInterrupt Traceback (most recent call last)

Cell In[60], line 6

```

4 for width in [4,32,128]:
5     print('Parameters: TRAIN_RATIO : {}, depth_generator : {},
↳depth_discriminator : {}, width : {}'.format(tr_ratio, depth_g, depth_d, width)
----> 6
↳train_gan(prior_n=10, gen_depth=depth_g, gen_width=width, disc_depth=depth_d, disc_width=width)
```

Cell In[56], line 76, in train_gan(prior_n, gen_depth, gen_width, disc_depth, disc_width, train_ratio, n_iter, batchlen, lr)

```

73     disc_loss = real_loss + fake_loss
75     disc_loss.backward()
---> 76     optim_disc.step()
78 # Train the generator
79 generator.zero_grad()
```

File /opt/anaconda3/envs/my_env/lib/python3.12/site-packages/torch/optim/optimizer.py

```

↳py:385, in Optimizer.profile_hook_step.<locals>.wrapper(*args, **kwargs)
380         else:
381             raise RuntimeError(
382                 f"{func} must return None or a tuple of (new_args,
↳new_kwargs), but got {result}."
383             )
--> 385 out = func(*args, **kwargs)
386 self._optimizer_step_code()
388 # call optimizer step post hooks
```

```

File /opt/anaconda3/envs/my_env/lib/python3.12/site-packages/torch/optim/optimizer.py:
  → py:76, in _use_grad_for_differentiable.<locals>._use_grad(self, *args, **kwargs)
    74     torch.set_grad_enabled(self.defaults['differentiable'])
    75     torch._dynamo.graph_break()
  → 76     ret = func(self, *args, **kwargs)
    77 finally:
    78     torch._dynamo.graph_break()

```

```

File /opt/anaconda3/envs/my_env/lib/python3.12/site-packages/torch/optim/adam.py:
  → 166, in Adam.step(self, closure)
    155     beta1, beta2 = group['betas']
    157     has_complex = self._init_group(
    158         group,
    159         params_with_grad,
    (... )
    163         max_exp_avg_sqs,
    164         state_steps)
  → 166     adam(
    167         params_with_grad,
    168         grads,
    169         exp_avgs,
    170         exp_avg_sqs,
    171         max_exp_avg_sqs,
    172         state_steps,
    173         amsgrad=group['amsgrad'],
    174         has_complex=has_complex,
    175         beta1=beta1,
    176         beta2=beta2,
    177         lr=group['lr'],
    178         weight_decay=group['weight_decay'],
    179         eps=group['eps'],
    180         maximize=group['maximize'],
    181         foreach=group['foreach'],
    182         capturable=group['capturable'],
    183         differentiable=group['differentiable'],
    184         fused=group['fused'],
    185         grad_scale=getattr(self, "grad_scale", None),
    186         found_inf=getattr(self, "found_inf", None),
    187     )
    189 return loss

```

```

File /opt/anaconda3/envs/my_env/lib/python3.12/site-packages/torch/optim/adam.py:
  → 316, in adam(params, grads, exp_avgs, exp_avg_sqs, max_exp_avg_sqs,
  → state_steps, foreach, capturable, differentiable, fused, grad_scale, found_inf
  → has_complex, amsgrad, beta1, beta2, lr, weight_decay, eps, maximize)
    313 else:
    314     func = _single_tensor_adam
  → 316 func(params,

```

```

317     grads,
318     exp_avgs,
319     exp_avg_sqs,
320     max_exp_avg_sqs,
321     state_steps,
322     amsgrad=amsgrad,
323     has_complex=has_complex,
324     beta1=beta1,
325     beta2=beta2,
326     lr=lr,
327     weight_decay=weight_decay,
328     eps=eps,
329     maximize=maximize,
330     capturable=capturable,
331     differentiable=differentiable,
332     grad_scale=grad_scale,
333     found_inf=found_inf)

```

File /opt/anaconda3/envs/my_env/lib/python3.12/site-packages/torch/optim/adam.py:

```

→441, in _single_tensor_adam(params, grads, exp_avgs, exp_avg_sqs,
→max_exp_avg_sqs, state_steps, grad_scale, found_inf, amsgrad, has_complex,
→beta1, beta2, lr, weight_decay, eps, maximize, capturable, differentiable)
    438     else:
    439         denom = (exp_avg_sq.sqrt() / bias_correction2_sqrt).add_(eps)
--> 441     param.addcddiv_(exp_avg, denom, value=-step_size)
    443 # Lastly, switch back to complex view
    444 if amsgrad and torch.is_complex(params[i]):

```

KeyboardInterrupt:

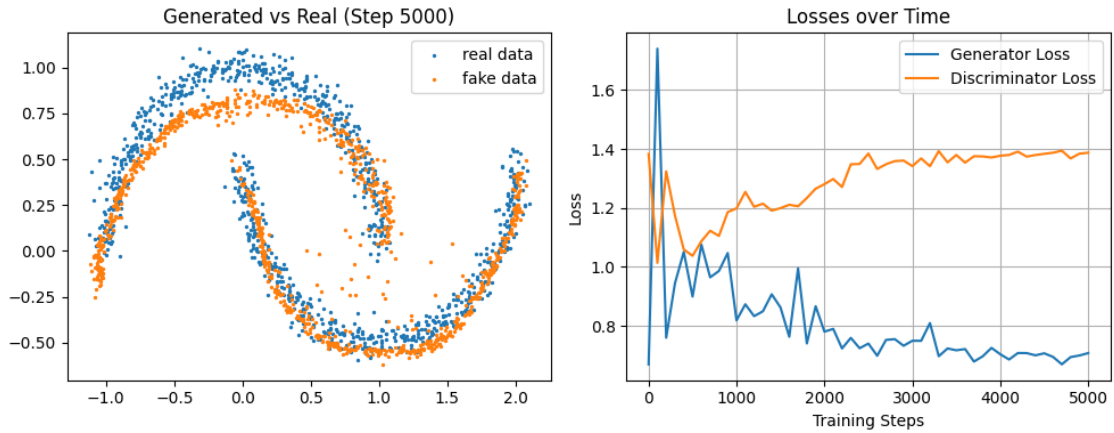
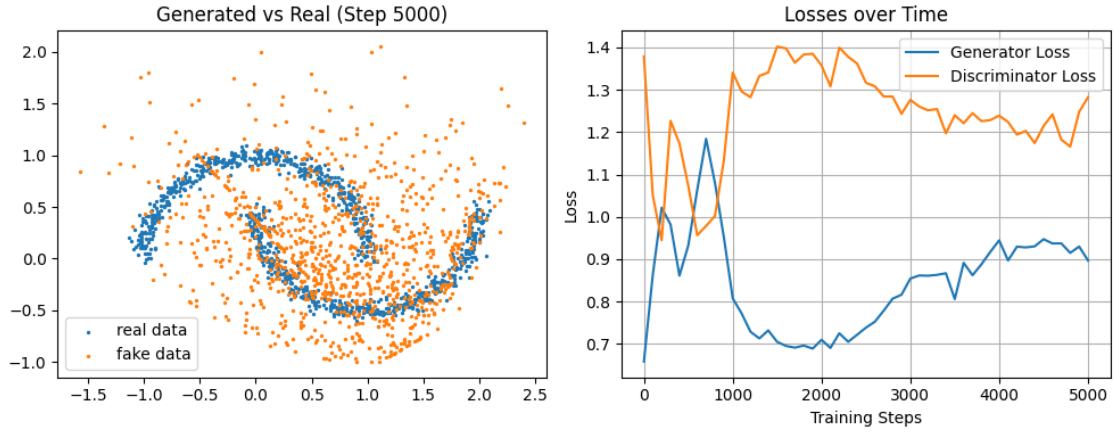
```

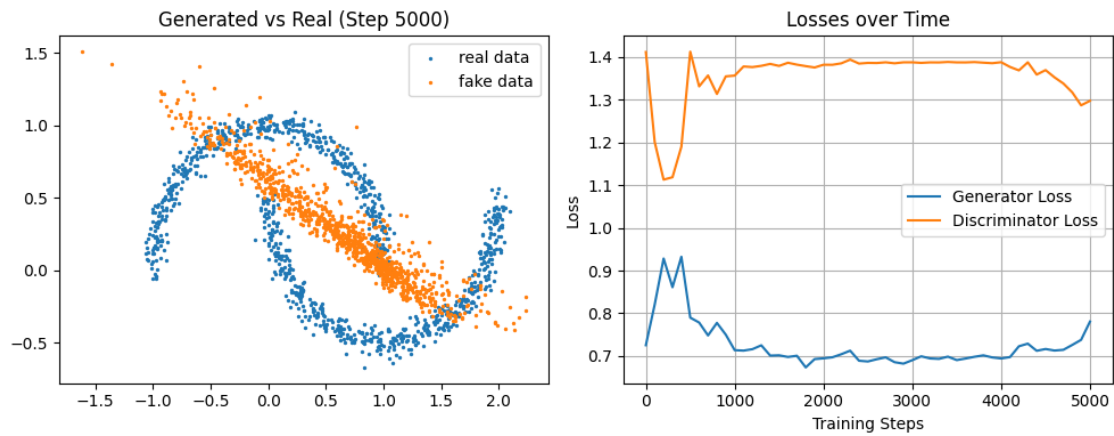
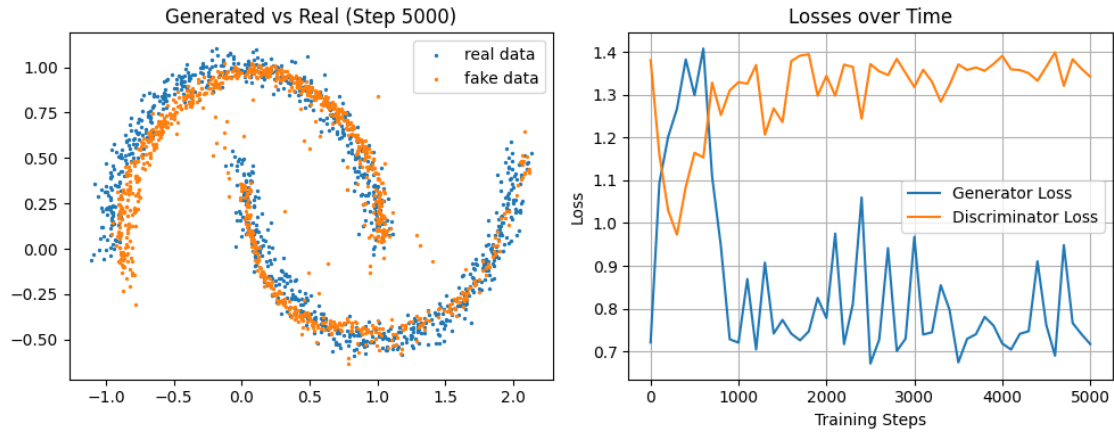
[66]: for tr_ratio in [3,5]:
      for depth in [4,2,1]:
          for width in [4,32,64]:
              print('Parameters: TRAIN_RATIO : {}, depth_generator : {},
→depth_discriminator : {}, width : {}'.format(tr_ratio, depth, depth, width))
              train_gan(prior_n=10, gen_depth=depth, gen_width=width,
→disc_depth=depth, disc_width=width, train_ratio=tr_ratio, n_iter = 5001)

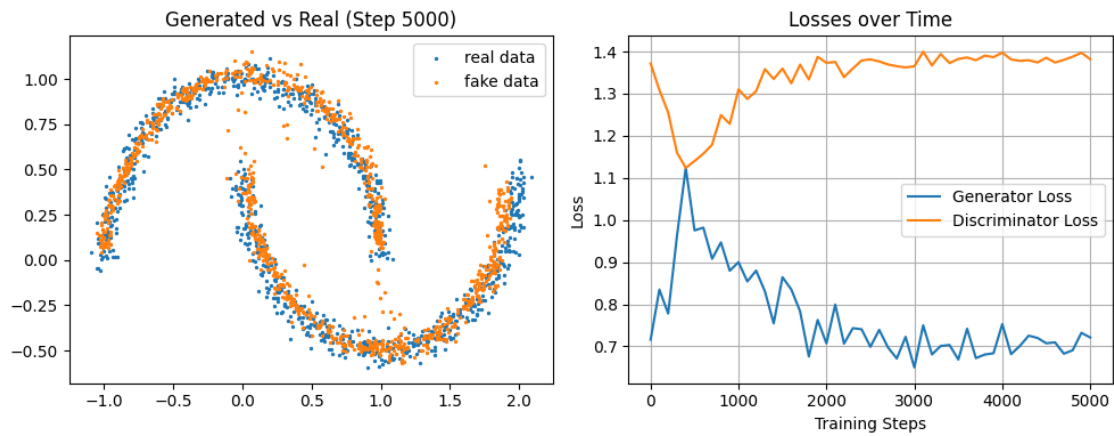
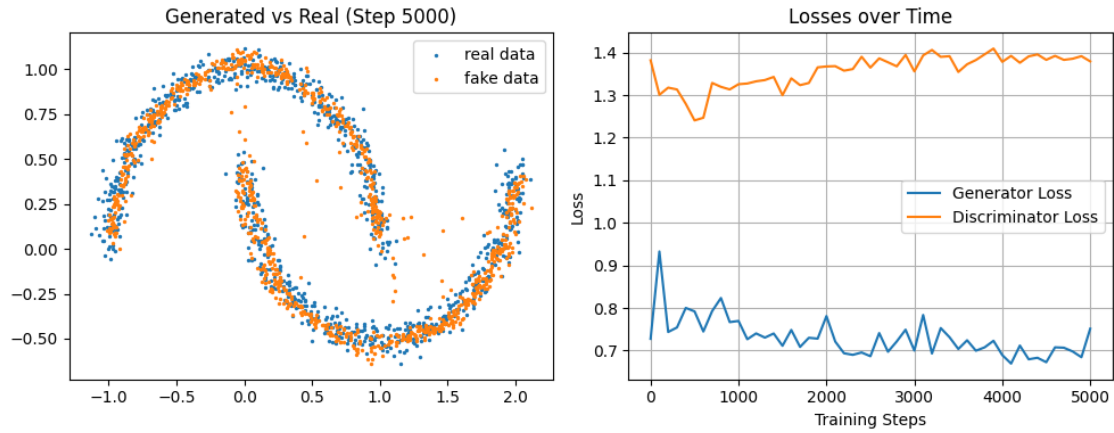
```

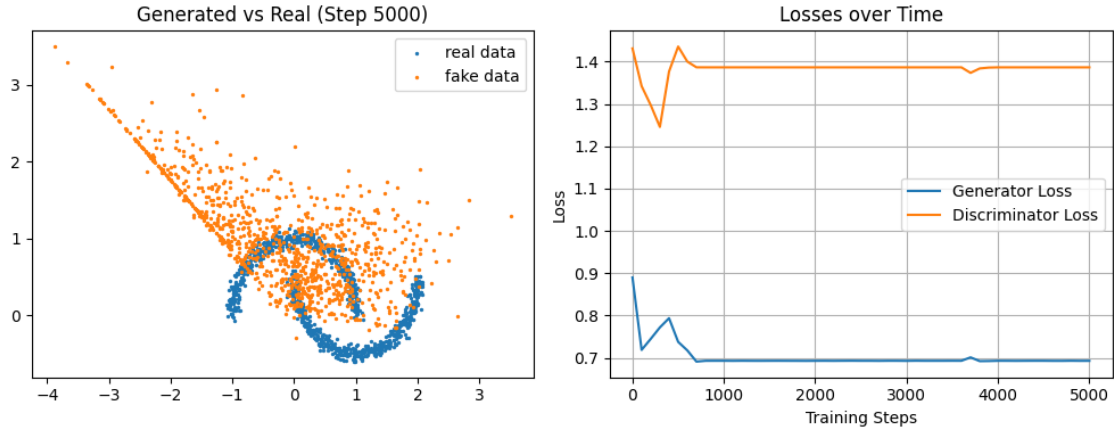
Parameters: TRAIN_RATIO : 3, depth_generator : 4, depth_discriminator : 4, width : 4

Step 5000: discriminator loss: 1.282e+00, generator loss: 8.968e-01



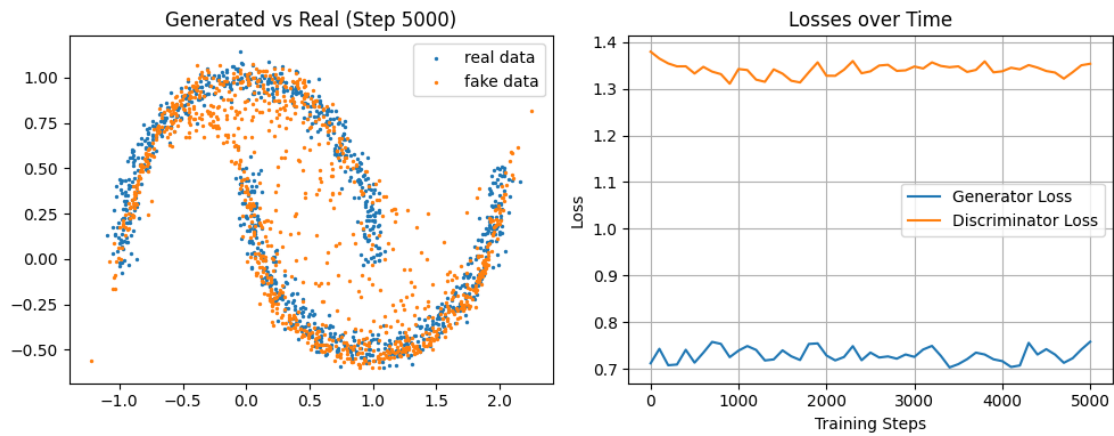






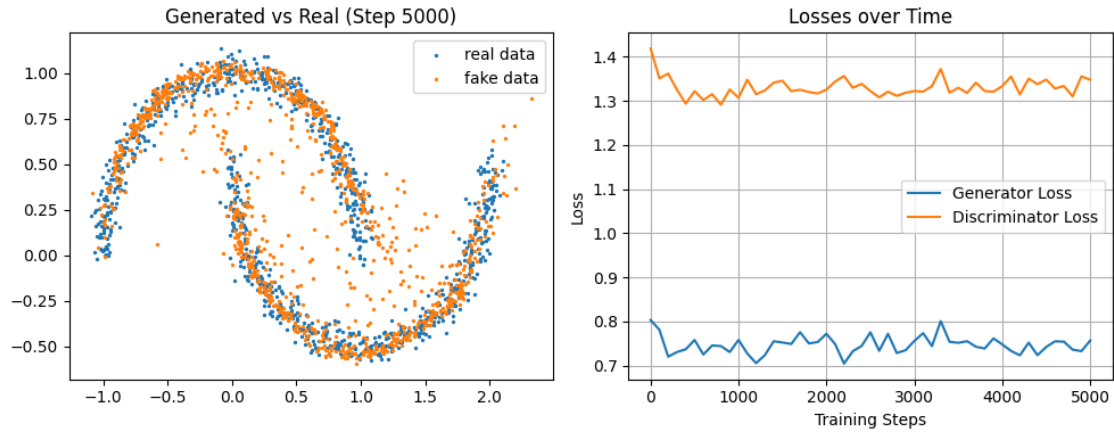
Parameters: TRAIN_RATIO : 3, depth_generator : 1, depth_discriminator : 1, width : 32

Step 5000: discriminator loss: 1.353e+00, generator loss: 7.580e-01



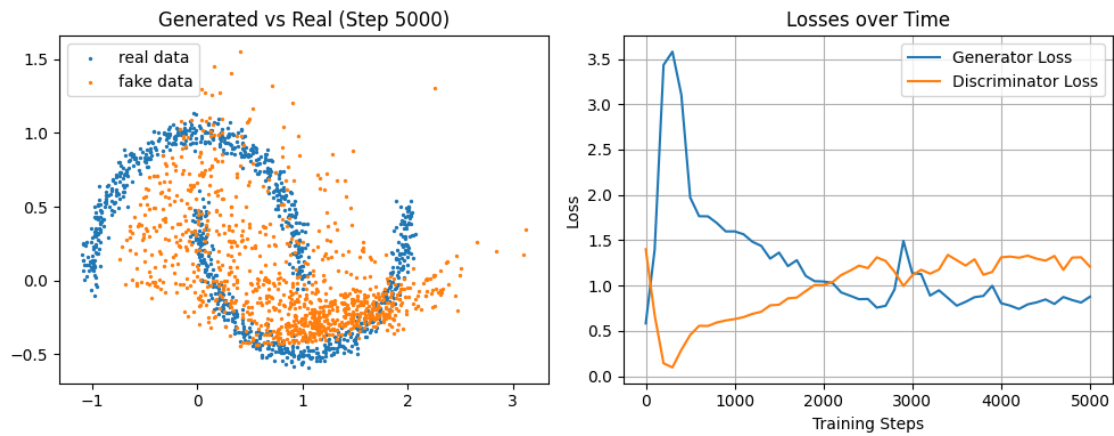
Parameters: TRAIN_RATIO : 3, depth_generator : 1, depth_discriminator : 1, width : 64

Step 5000: discriminator loss: 1.348e+00, generator loss: 7.566e-01



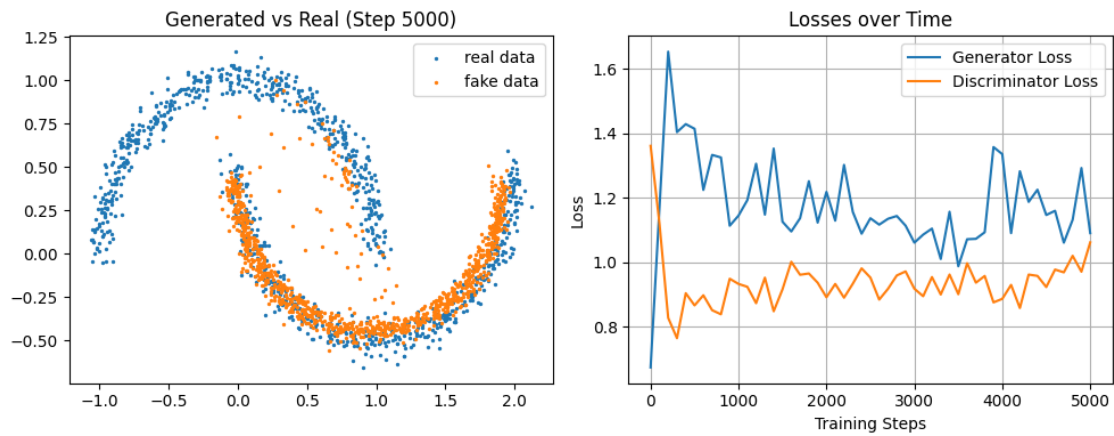
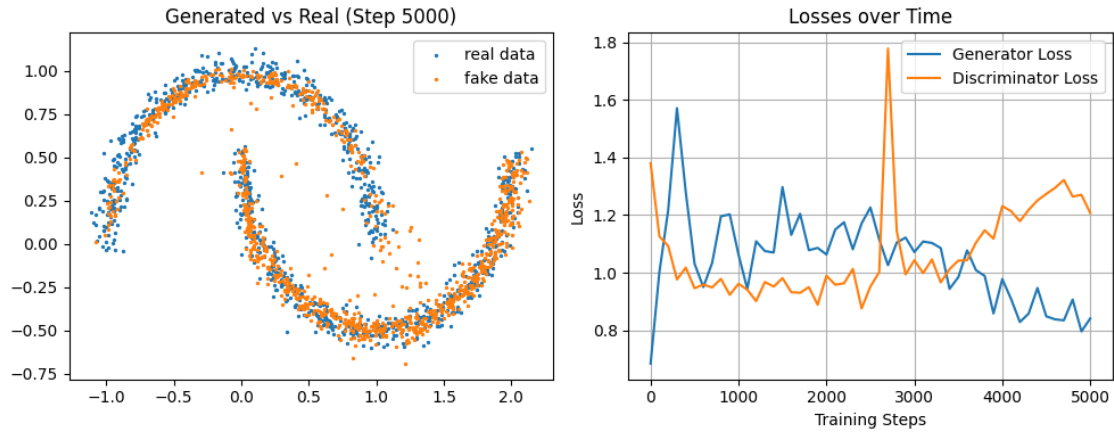
Parameters: TRAIN_RATIO : 5, depth_generator : 4, depth_discriminator : 4, width : 4

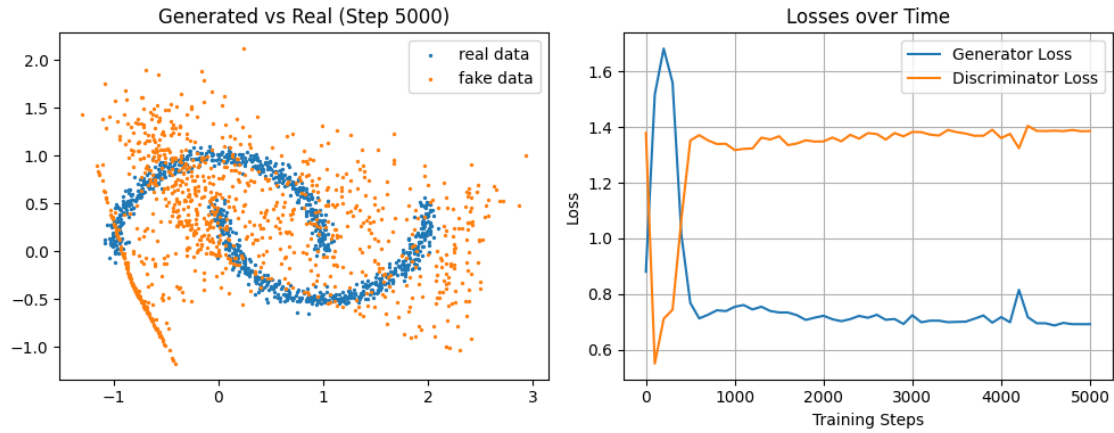
Step 5000: discriminator loss: 1.208e+00, generator loss: 8.766e-01



Parameters: TRAIN_RATIO : 5, depth_generator : 4, depth_discriminator : 4, width : 32

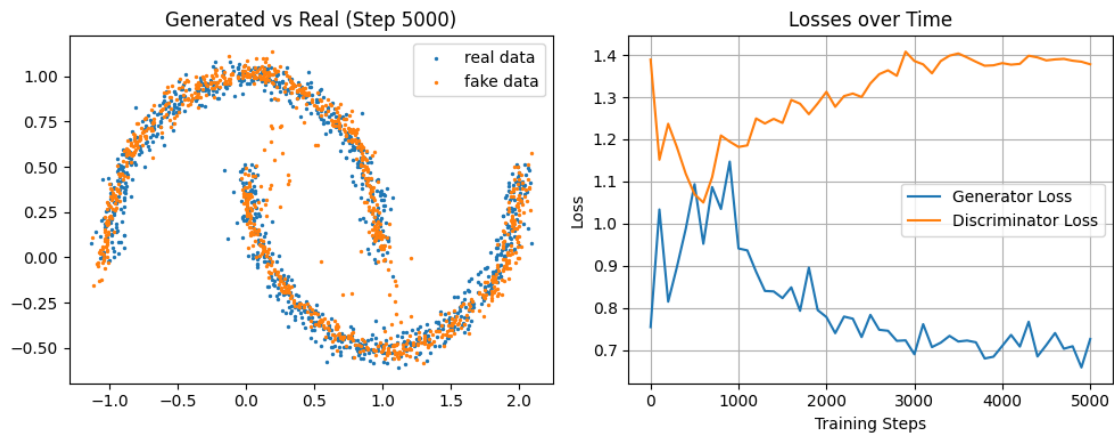
Step 5000: discriminator loss: 1.207e+00, generator loss: 8.410e-01





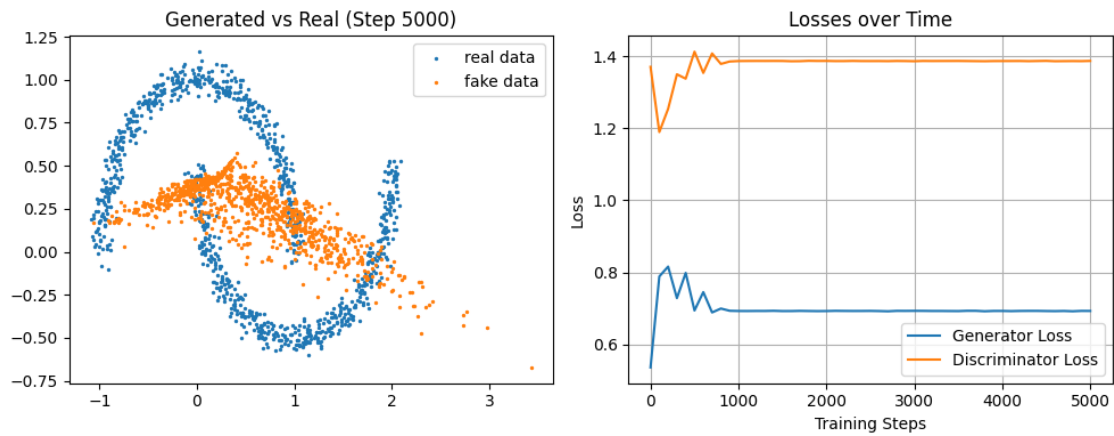
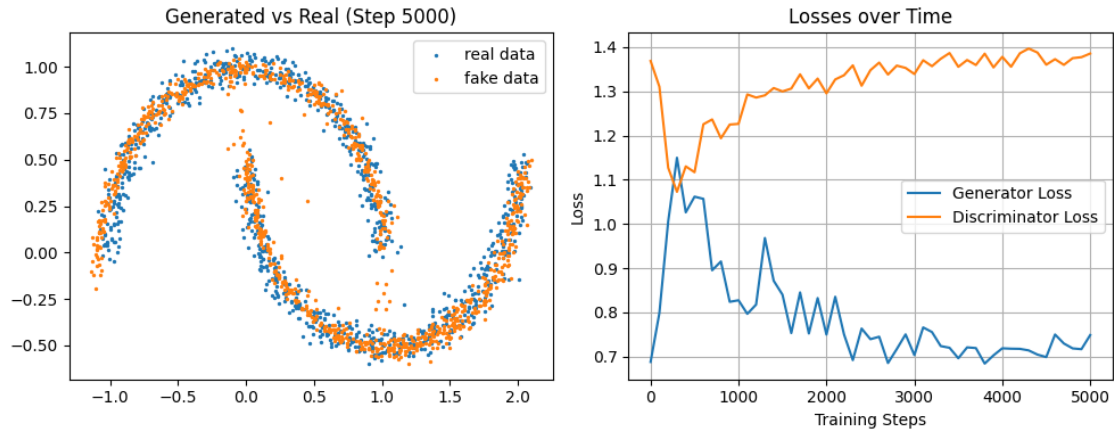
Parameters: TRAIN_RATIO : 5, depth_generator : 2, depth_discriminator : 2, width : 32

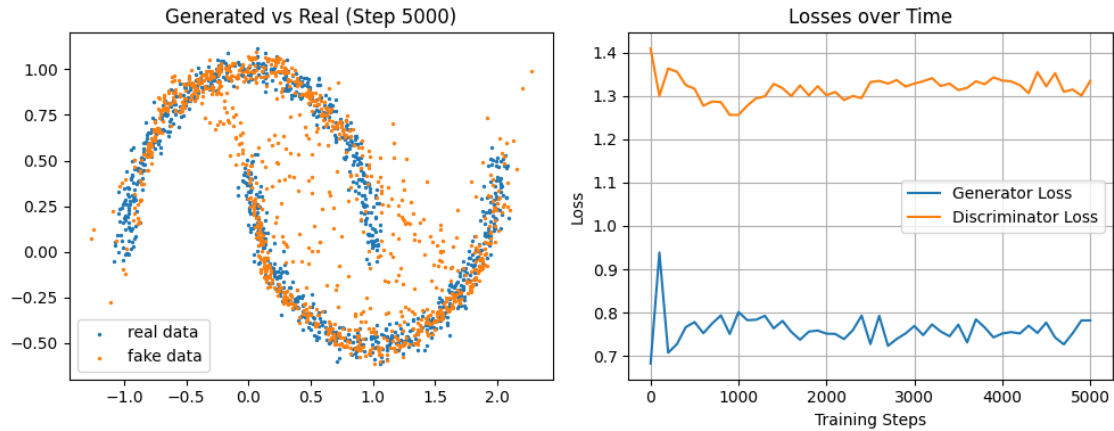
Step 5000: discriminator loss: 1.378e+00, generator loss: 7.260e-01



Parameters: TRAIN_RATIO : 5, depth_generator : 2, depth_discriminator : 2, width : 64

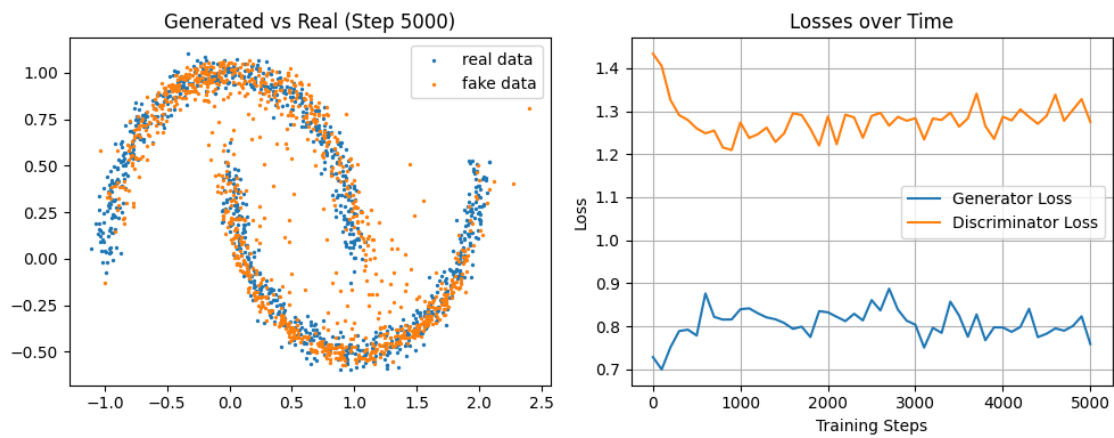
Step 5000: discriminator loss: 1.386e+00, generator loss: 7.489e-01





Parameters: TRAIN_RATIO : 5, depth_generator : 1, depth_discriminator : 1, width : 64

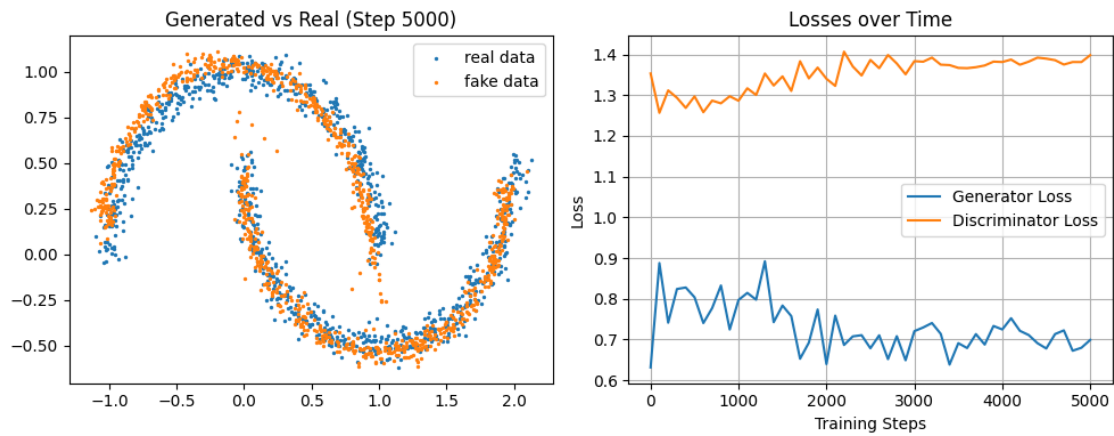
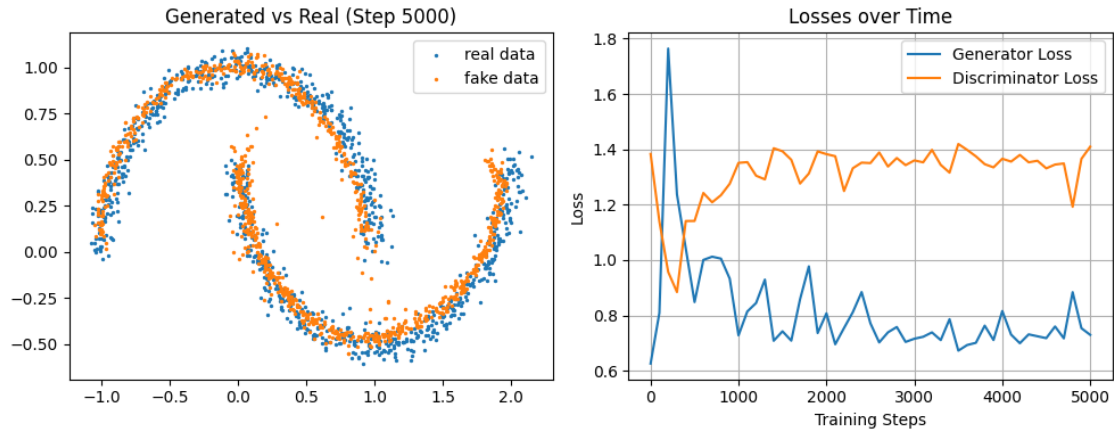
Step 5000: discriminator loss: 1.275e+00, generator loss: 7.591e-01

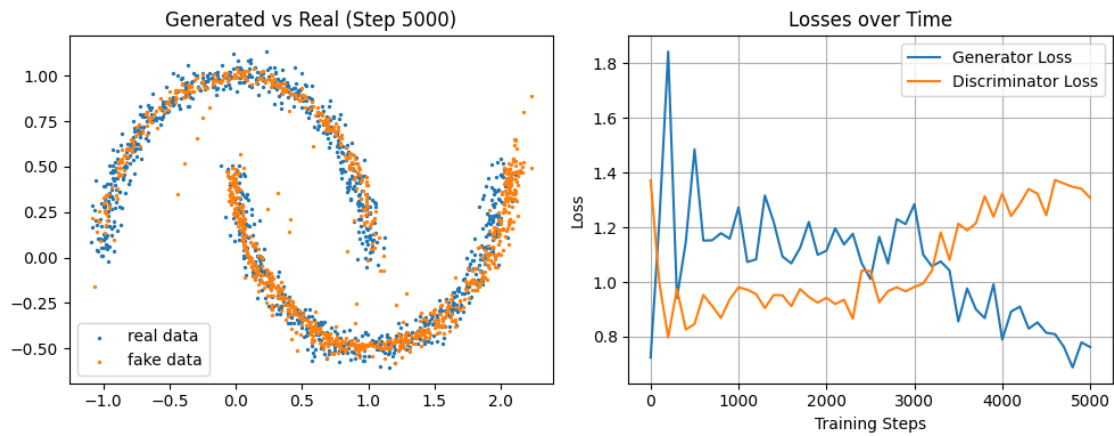
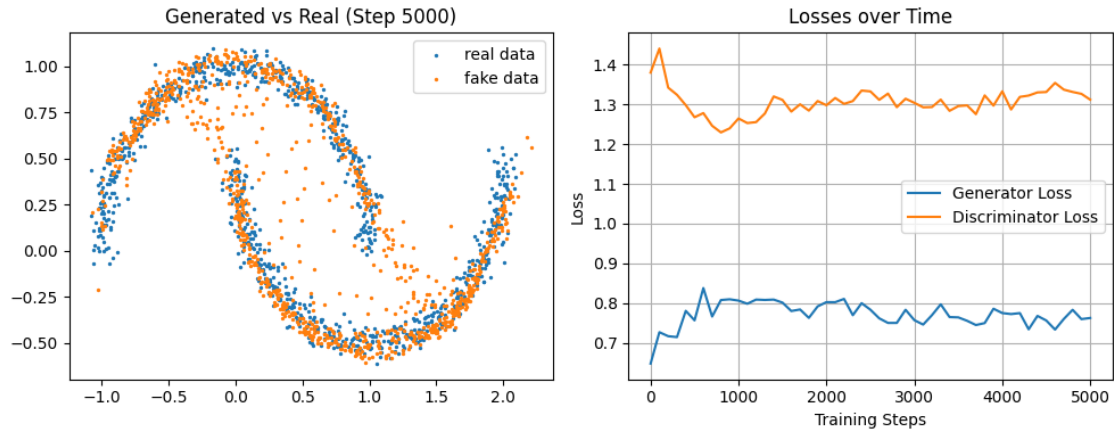


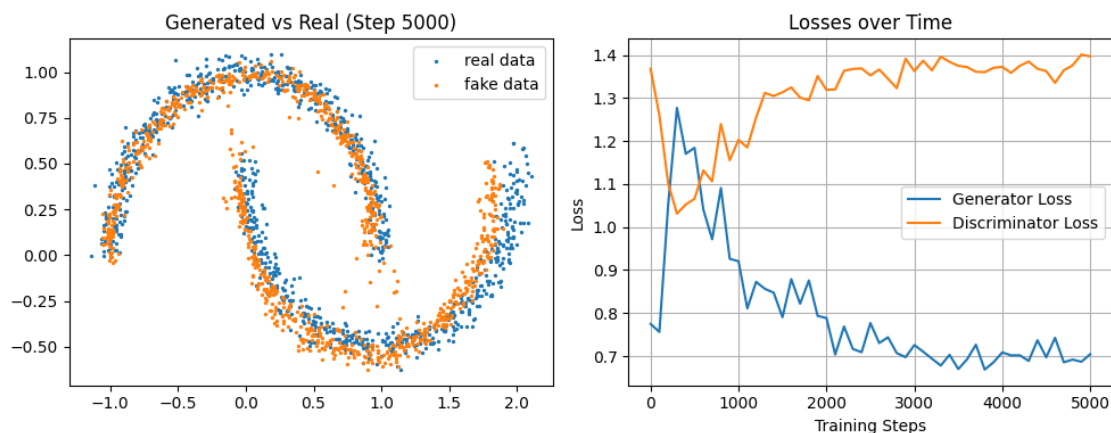
```
[69]: for tr_ratio in [3,5]:
      for depth in [4,2,1]:
          print('Parameters: TRAIN_RATIO : {}, depth_generator : {},
↳depth_discriminator : {}, width : {}'.format(tr_ratio, depth, depth, 128))
          train_gan(prior_n=10, gen_depth=depth, gen_width=width,
↳disc_depth=depth, disc_width=width, train_ratio=tr_ratio, n_iter = 5001)
```

Parameters: TRAIN_RATIO : 3, depth_generator : 4, depth_discriminator : 4, width : 128

Step 5000: discriminator loss: 1.410e+00, generator loss: 7.298e-01

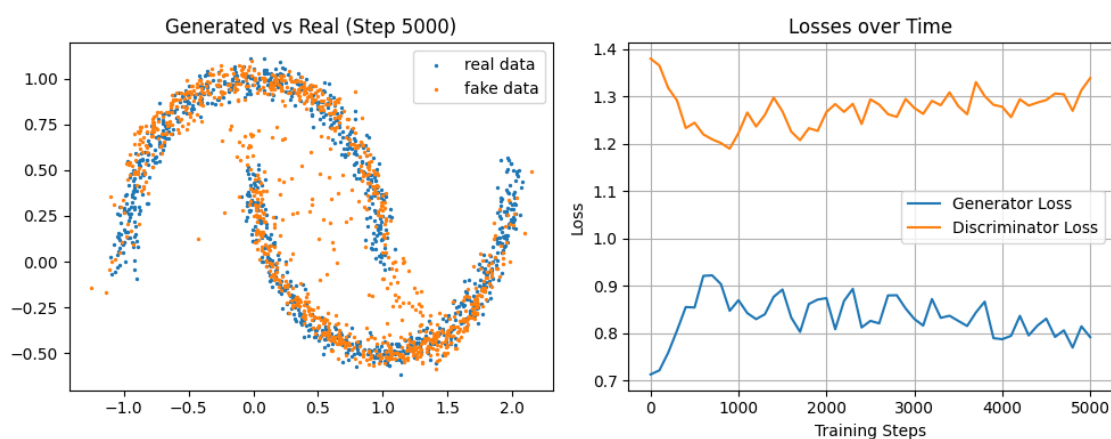






Parameters: TRAIN_RATIO : 5, depth_generator : 1, depth_discriminator : 1, width : 128

Step 5000: discriminator loss: 1.339e+00, generator loss: 7.916e-01



Keeping the generator and discriminator as it is doesn't provide good results. However, if we increase the number of layer in the generator as well as for the discriminator, the points generated look like the real points.

1.2.1 Train Ratio

The **train ratio** determines how many times the discriminator is updated for each generator update. A higher train ratio (>10) allows the discriminator to learn faster and potentially become more accurate in distinguishing real from fake samples. But, if the discriminator becomes too strong compared to the generator, it may lead to **vanishing gradients**, where the generator receives almost no feedback for improvement. On the other hand, a low train ratio (1) lead to poor sample quality. We can take something in between like 5.

1.2.2 Generator and Discriminator Depth

The **depth** of the generator and discriminator (the number of hidden layers) controls the representational capacity of each network. A **deeper generator** can model more complex mappings from the latent space to the data space, resulting to more realistic outputs. However, excessive depth in either network can lead to **overfitting** and increased training time.

If generator depth is inferior to discriminator depth, the model doesn't succeed to reconstruct the real dataset. It is better if it is the same. Ans if we increase it a little, like 2 or 4 depth, the generation is better but with 4, the computational cost is too much. Also, in some cases, it only captures one moon over the two. With little depth, it can gives quite good result but it is noisy.

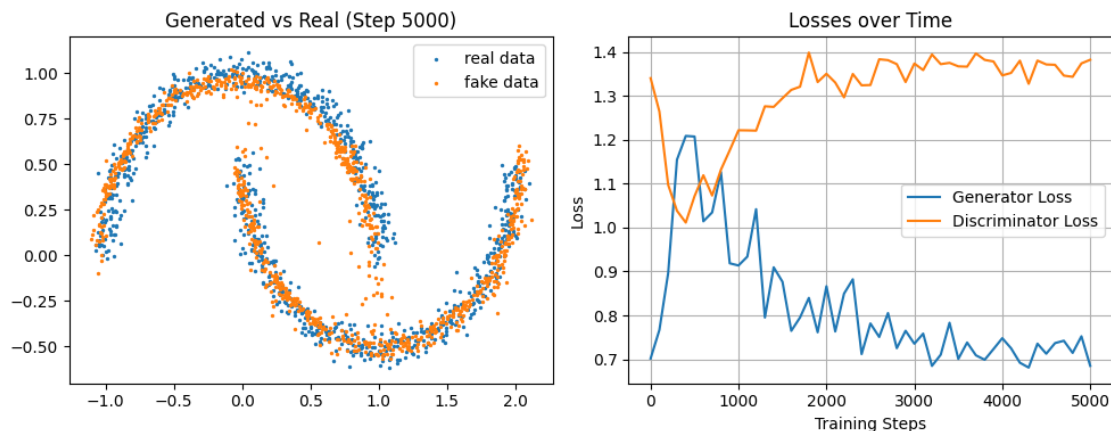
1.2.3 Generator and Discriminator Width

The **width** of a network refers to the number of neurons in each hidden layer. Increasing width increase the network ability to process and represent rich information, which is beneficial when the target data distribution is highly complex. For both the generator and discriminator, wider layers mean more expressive models, but also introduce risks like **overparameterization**, **slower convergence**, and **higher computational cost**. Thus, selecting appropriate widths is a trade-off between model expressiveness and training stability. Increasing it until 64 still gives good results.

The best option are with parameters : TRAIN_RATIO : 5, depth_generator : 2, depth_discriminator : 2, width : 64

```
[70]: train_gan(prior_n=10, gen_depth=2, gen_width=64, disc_depth=2, disc_width=64,
→train_ratio=5, n_iter = 5001)
```

Step 5000: discriminator loss: 1.382e+00, generator loss: 6.852e-01



For further readings on GANs, you can see the following papers:

- Generative Adversarial Networks (*Goodfellow et al.*): [arXiv:1406.2661](https://arxiv.org/abs/1406.2661)

- Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks (*Radford et al.*): [arXiv:1511.06434](https://arxiv.org/abs/1511.06434)
 - A Style-Based Generator Architecture for Generative Adversarial Networks (*Karras et al.*) [arxiv:1812.04948v3](https://arxiv.org/abs/1812.04948v3)
-

1.3 Variational AutoEncoders

Another well-known approach to generative modeling is embodied by Variational AutoEncoders (VAEs). While the generative model itself and the procedure to sample it is similar to GANs, the way it is trained is not.

The main goal of VAEs is to optimize the likelihood of the real data according to the generative model. In other words, maximize $\mathbb{E}_{p_D} [\log p_\theta(x)]$, which is equivalent to minimizing $D_{KL}(p_D||p_\theta)$.

$$D_{KL}(p_D||p_\theta) = \mathbb{E}_{p_D} \left[\log\left(\frac{p_D}{p_\theta}\right) \right]$$

Q8: Prove this equivalence.

We start with the definition of the Kullback-Leibler divergence between the data distribution p_D and the model distribution p_θ :

$$D_{KL}(p_D||p_\theta) = \mathbb{E}_{x \sim p_D} \left[\log \frac{p_D(x)}{p_\theta(x)} \right]$$

We can rewrite this expected value by splitting the logarithm:

$$D_{KL}(p_D||p_\theta) = \mathbb{E}_{x \sim p_D} [\log p_D(x)] - \mathbb{E}_{x \sim p_D} [\log p_\theta(x)]$$

Notice that $\mathbb{E}_{x \sim p_D} [\log p_D(x)]$ does not depend on the parameters θ . Hence, when optimizing with respect to θ , this term is effectively a constant.

Therefore, minimizing $D_{KL}(p_D||p_\theta)$ with respect to θ is equivalent to maximizing:

$$\mathbb{E}_{x \sim p_D} [\log p_\theta(x)]$$

In other words:

$$\boxed{\min_{\theta} D_{KL}(p_D||p_\theta) \iff \max_{\theta} \mathbb{E}_{x \sim p_D} [\log p_\theta(x)]}$$

This shows that minimizing the KL divergence $D_{KL}(p_D||p_\theta)$ is equivalent to maximizing the expected log-likelihood $\mathbb{E}_{p_D} [\log p_\theta(x)]$.

However, the classes of distributions for which $\log p_\theta(x)$ can be analytically computed and optimized is very restricted, and not suitable for real world problems. The main idea of the VAE is thus to introduce a latent variable z and decompose the distribution as: $p_\theta(x, z) = p_\theta(x|z)p(z)$. Here $p(z)$ is some fixed prior and $p_\theta(x|z)$ is a simple distribution whose parameters are the output of a neural network.

For example, you could have $p(z)$ be a standard $\mathcal{N}(0;1)$ and $p_\theta(x|z)$ be defined as a gaussian $\mathcal{N}(\mu_\theta(z); \sigma_\theta(z))$ where $\mu_\theta(z)$ and $\sigma_\theta(z)$ are created by the neural network you will train. In this case, the resulting distribution $p_\theta(x) = \int_z p_\theta(x|z)p(z)dz$ is an infinite mixture of Gaussians, which is a much more expressive class of distributions.

Now, this cannot stop here, as we are not able to analitically compute the density $p_\theta(x)$. The second main idea of the VAE is to introduce another, auxilliary distribution: $q_\phi(z|x)$, which will be modelled by a neural network similarly to $p_\theta(x|z)$. Introducing it allows us to create a lower bound for $\log p_\theta(x)$:

$$\log p_\theta(x) = \mathbb{E}_{z \sim q_\phi} \log p_\theta(x) = \mathbb{E}_{z \sim q_\phi} \left[\log p_\theta(x) \frac{q_\phi(z|x)}{q_\phi(z|x)} \right]$$

Following Bayes theorem, $p_\theta(x)p_\theta(z|x) = p_\theta(x, z) = p_\theta(x|z)p(z)$, so we get:

$$\log p_\theta(x) = \mathbb{E}_{z \sim q_\phi} \left[\log \frac{p_\theta(x|z)p(z)}{p_\theta(z|x)} \frac{q_\phi(z|x)}{q_\phi(z|x)} \right]$$

Re-organizing the terms:

$$\log p_\theta(x) = \mathbb{E}_{z \sim q_\phi} \log \frac{q_\phi(z|x)}{p_\theta(z|x)} + \mathbb{E}_{z \sim q_\phi} \log \frac{p(z)}{q_\phi(z|x)} + \mathbb{E}_{z \sim q_\phi} \log p_\theta(x|z)$$

This can be re-expressed like so:

$$\log p_\theta(x) = D_{KL}(q_\phi(z|x) || p_\theta(z|x)) - D_{KL}(q_\phi(z|x) || p(z)) + \mathbb{E}_{z \sim q_\phi} \log p_\theta(x|z)$$

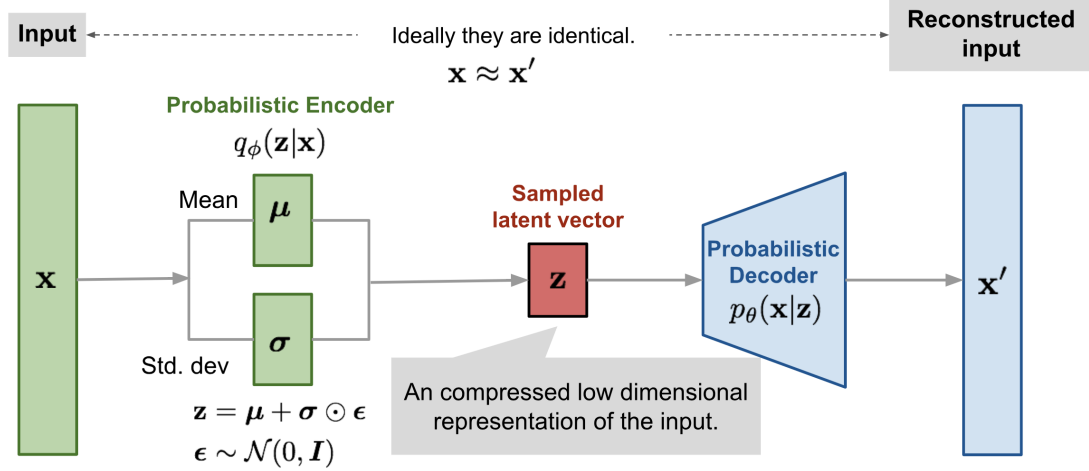
The 3 terms of this equality can be interpreted like so:

- the first term measures how much $q_\phi(z|x)$ is similar to $p_\theta(z|x)$, or in other words is a good inverse of $p_\theta(x|z)$
- the second term measures how similar $q_\phi(z|x)$ is from the latent prior $p(z)$
- the third term is linked to how likely p_θ is to yield the given x when z is sampled from $q_\phi(z|x)$ rather than $p(z)$

It is interesting to note that the first term, being a KL-divergence is always positive. As such the combination of the last two terms forms a lower bound of $\log p_\theta(x)$ which *can* be computed and used as a training objective. This bound is called the *Evidence Lower-Bound (ELBO)*. Simply flipping its sign can make it into a loss that can be minimized by gradient descent:

$$\mathcal{L}_{ELBO} = D_{KL}(q_\phi(z|x) || p(z)) + \mathbb{E}_{z \sim q_\phi} [-\log p_\theta(x|z)]$$

From this formulation comes the parallel with auto-encoders that give the VAE its name: $q_\phi(z|x)$ can be seen as a *probabilistic encoder* from the data x to the latent space z , and $p_\theta(x|z)$ can be seen as a *probabilistic decoder* from the latent space z to the data x . In this case the second term of $\mathcal{L}_{ELBO}$ is the loss measuring the reconstruction quality of the auto-encoder, and the first term can be seen as a regularization of the latent space.



Q9: We can see that $p(z)$ is never sampled during the training process, how can that be a problem?

When we generate new data after training, we will sample z from $p(z)$ rather than from $q_\phi(z|x)$. Because $p(z)$ is never actually used for sampling during training, the encoder might learn a latent representation that doesn't align well with this prior distribution. As a result, when we sample from $p(z)$, the decoder could receive z values it isn't familiar with, leading to poor or unrealistic outputs.

A typical choice to represent $q_\phi(z|x)$ is to use a diagonal Gaussian distribution $\mathcal{N}(\mu_\phi(x); \text{Diag}(\sigma_\phi^2(x)))$, which makes the KL-divergence term of $\mathcal{L}_{ELBO}$ analytically computable.

Q10: Assuming $p(z)$ is a $\mathcal{N}(0; Id)$ gaussian, what is the value of $D_{KL}(q_\phi(z|x)||p(z))$?

We want to compute the Kullback-Leibler divergence between two diagonal multivariate Gaussians:

$$q_\phi(z|x) = \mathcal{N}(\mu_\phi(x), \text{Diag}(\sigma_\phi^2(x))) \quad \text{and} \quad p(z) = \mathcal{N}(0, I).$$

We start with the general formula for the KL divergence between two Gaussians

$$D_{KL}(\mathcal{N}(\mu_1, \Sigma_1) \parallel \mathcal{N}(\mu_2, \Sigma_2)) = \frac{1}{2} \left[\text{tr}(\Sigma_2^{-1} \Sigma_1) + (\mu_2 - \mu_1)^\top \Sigma_2^{-1} (\mu_2 - \mu_1) - k + \log \left(\frac{\det \Sigma_2}{\det \Sigma_1} \right) \right]$$

where k is the dimensionality of z .

In our case:

- Since $p(z) = \mathcal{N}(0, I)$, we have $\mu_2 = 0$ and $\Sigma_2 = I$
- Since $q_\phi(z|x) = \mathcal{N}(\mu_\phi, \text{Diag}(\sigma_\phi^2))$, we have $\mu_1 = \mu_\phi$ and $\Sigma_1 = \text{Diag}(\sigma_\phi^2)$

We can now simplify the expression term by term:

- $\text{tr}(\Sigma_2^{-1} \Sigma_1) = \text{tr}(I \cdot \text{Diag}(\sigma_\phi^2)) = \sum_i \sigma_{\phi,i}^2$
- $(\mu_2 - \mu_1)^\top (\mu_2 - \mu_1) = \sum_i \mu_{\phi,i}^2$
- $\log \left(\frac{\det \Sigma_2}{\det \Sigma_1} \right) = -\log \det \Sigma_1 = -\sum_i \log \sigma_{\phi,i}^2$

- And k is simply the number of dimensions, i.e., $\sum_i 1$

Substituting these into the general formula, we obtain:

$$D_{\text{KL}}(q_\phi(z|x)||p(z)) = \frac{1}{2} \sum_i (\sigma_{\phi,i}^2(x) + \mu_{\phi,i}^2(x) - 1 - \log \sigma_{\phi,i}^2(x))$$

We will also model $p_\theta(x|z)$ as a diagonal Gaussian $\mathcal{N}(\mu_\theta(z); \text{Diag}(\sigma_\theta^2(z)))$.

Note: For the following, be careful about the difference between μ_ϕ, σ_ϕ which define the Gaussian distribution of the *encoder* q_ϕ and $\mu_\theta, \sigma_\theta$ which define the Gaussian distribution of the *decoder* p_θ .

Q11: What is the expression of $-\log p_\theta(x|z)$ for given x and z ?

We assume the decoder defines:

$$p_\theta(x | z) = \mathcal{N}(\mu_\theta(z), \text{Diag}(\sigma_\theta^2(z)))$$

The log-density of a diagonal Gaussian $\mathcal{N}(\mu, \text{Diag}(\sigma^2))$ evaluated at x is given by:

$$\log(\mathcal{N}(x; \mu, \text{Diag}(\sigma^2))) = -\frac{d}{2} \log(2\pi) - \frac{1}{2} \sum_{i=1}^d \log \sigma_i^2 - \sum_{i=1}^d \frac{(x_i - \mu_i)^2}{2 \sigma_i^2}$$

where d is the dimensionality of x .

Grouping the terms $\log(2\pi)$ and $\log(\sigma_i^2)$ together, we have:

$$\log(2\pi) + \log(\sigma_i^2) = \log(2\pi \sigma_i^2)$$

Thus, the negative log-likelihood becomes:

$$-\log p_\theta(x | z) = \frac{1}{2} \sum_i \left[\log(2\pi \sigma_{\theta,i}^2(z)) + \frac{(x_i - \mu_{\theta,i}(z))^2}{\sigma_{\theta,i}^2(z)} \right]$$

We will build and train a VAE using the same dataset as previously, in order to compare its behavior to GANs. For numerical stability, we will interpret the output of the encoder and decoder networks as $(\mu, \log \sigma^2)$, rather than (μ, σ) .

```
[9]: # Choose a value for the latent dimension
LATENT_N = 10

# Define the generator
class Encoder(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc_mu = nn.Linear(2, LATENT_N)
        self.fc_logvar = nn.Linear(2, LATENT_N)
```

```

    # encode a datapoint. This should return a couple of tensors (mu, logvar)
    → representing
    # the parameters of the Gaussian  $q_\phi(z | x)$ 
    def __call__(self, x):
        mu = self.fc_mu(x)
        logvar = self.fc_logvar(x)
        return (mu, logvar)

# Define the discriminator
class Decoder(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc_mu = nn.Linear(LATENT_N, 2)
        self.fc_logvar = nn.Linear(LATENT_N, 2)

    # decode a datapoint. This should return a couple of tensors (mu, logvar)
    → representing
    # the parameters of the Gaussian  $p_\theta(z | x)$ 
    def __call__(self, z):
        mu = self.fc_mu(z)
        logvar = self.fc_logvar(z)
        return (mu, logvar)

    def generate(self, batchlen):
        z = torch.normal(torch.zeros(batchlen, LATENT_N), 1.0)
        (mu, logvar) = self.__call__(z)
        return torch.normal(mu, torch.exp(0.5*logvar))

```

From this, the parameters of both networks are trained conjointly using the same loss $\mathcal{L}_{ELBO}$.

```

[10]: # Total number of training iterations for the VAE
N_ITER = 40001
# Batch size to use
BATCHLEN = 128

encoder = Encoder()
optim_enc = torch.optim.Adam(encoder.parameters(), lr=0.001, betas=(0.5,0.9))
decoder = Decoder()
optim_dec = torch.optim.Adam(decoder.parameters(), lr=0.001, betas=(0.5,0.9))

for i in range(N_ITER):
    encoder.zero_grad()
    decoder.zero_grad()

    x = generate_batch(BATCHLEN)

```

```

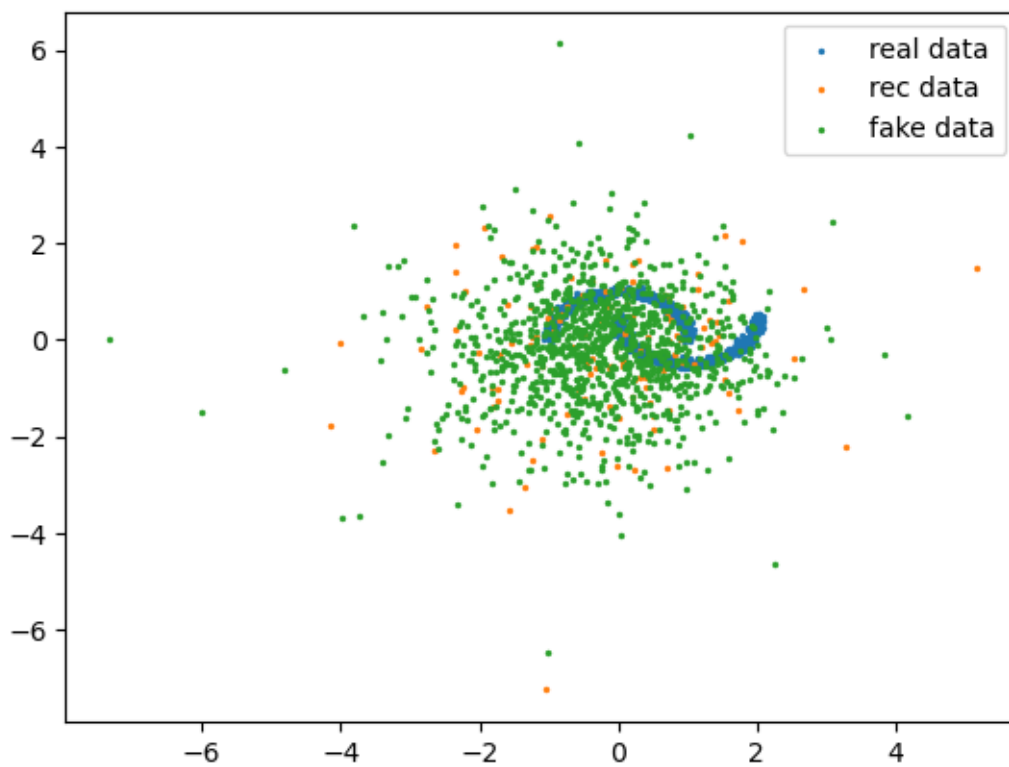
enc_mu, enc_logvar = encoder(x)
#
# COMPUTE THE KL PART OF THE LOSS HERE
#
loss_kl = 0.5 * torch.mean(
    torch.sum(
        torch.exp(enc_logvar) + enc_mu**2 - 1.0 - enc_logvar,
        dim=1
    )
)
#
# SAMPLE z FROM q(z/x) HERE
#
z = enc_mu + torch.exp(0.5 * enc_logvar) * torch.randn_like(enc_mu)

dec_mu, dec_logvar = decoder(z)
#
# COMPUTE THE RECONSTRUCTION PART OF THE LOSS HERE
#
loss_rec = 0.5 * torch.mean(
    torch.sum(
        dec_logvar + (x - dec_mu)**2 * torch.exp(-dec_logvar),
        dim=1
    ) + 2 * math.log(2*math.pi)
)

(loss_kl + loss_rec).backward()
optim_enc.step()
optim_dec.step()
if i%1000 == 0:
    print('step {}: KL: {:.3e}, rec: {:.3e}'.format(i, float(loss_kl),
→float(loss_rec)))
    # plot the result
    real_batch = generate_batch(1024)
    rec_batch = torch.normal(dec_mu, torch.exp(0.5*dec_logvar)).detach()
    fake_batch = decoder.generate(1024).detach()
    plt.scatter(real_batch[:,0], real_batch[:,1], s=2.0, label='real data')
    plt.scatter(rec_batch[:,0], rec_batch[:,1], s=2.0, label='rec data')
    plt.scatter(fake_batch[:,0], fake_batch[:,1], s=2.0, label='fake data')
    plt.legend()
    plt.show()

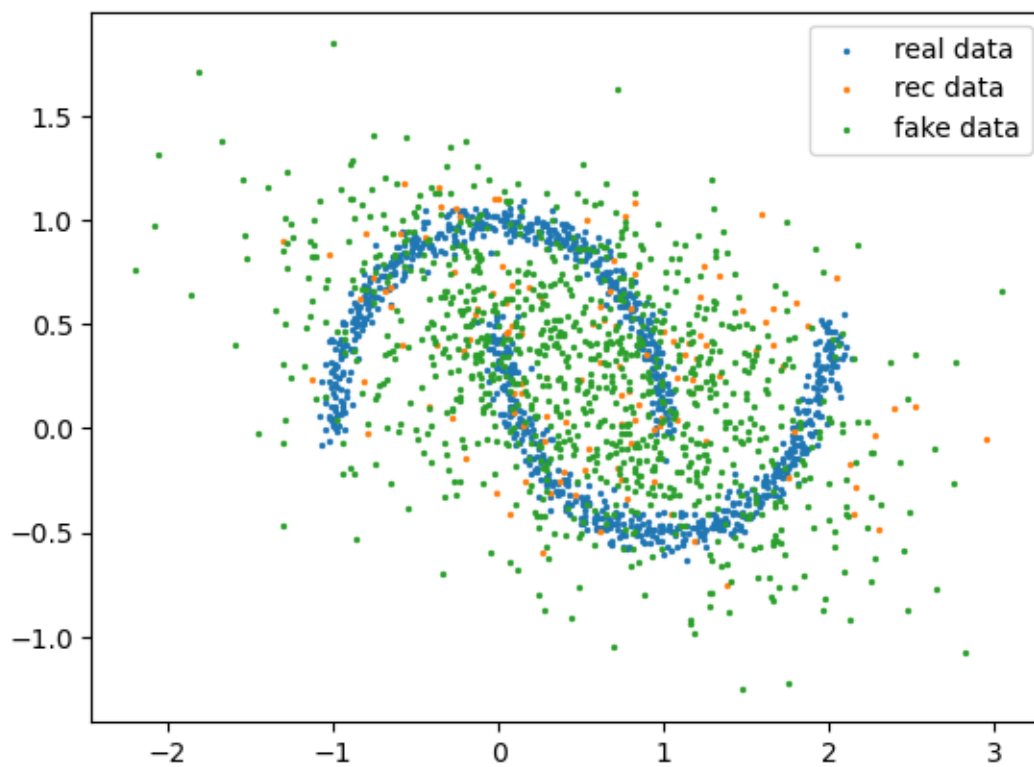
```

step 0: KL: 3.694e+00, rec: 3.294e+00

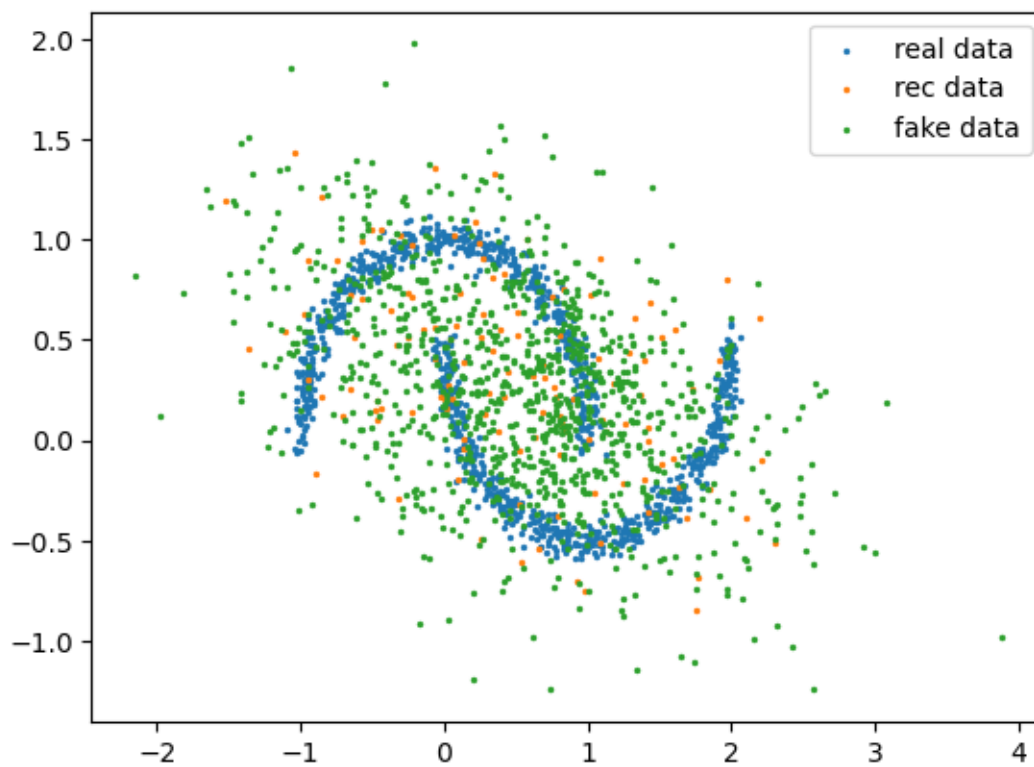


step 10000: KL: 4.857e-01, rec: 1.373e+00

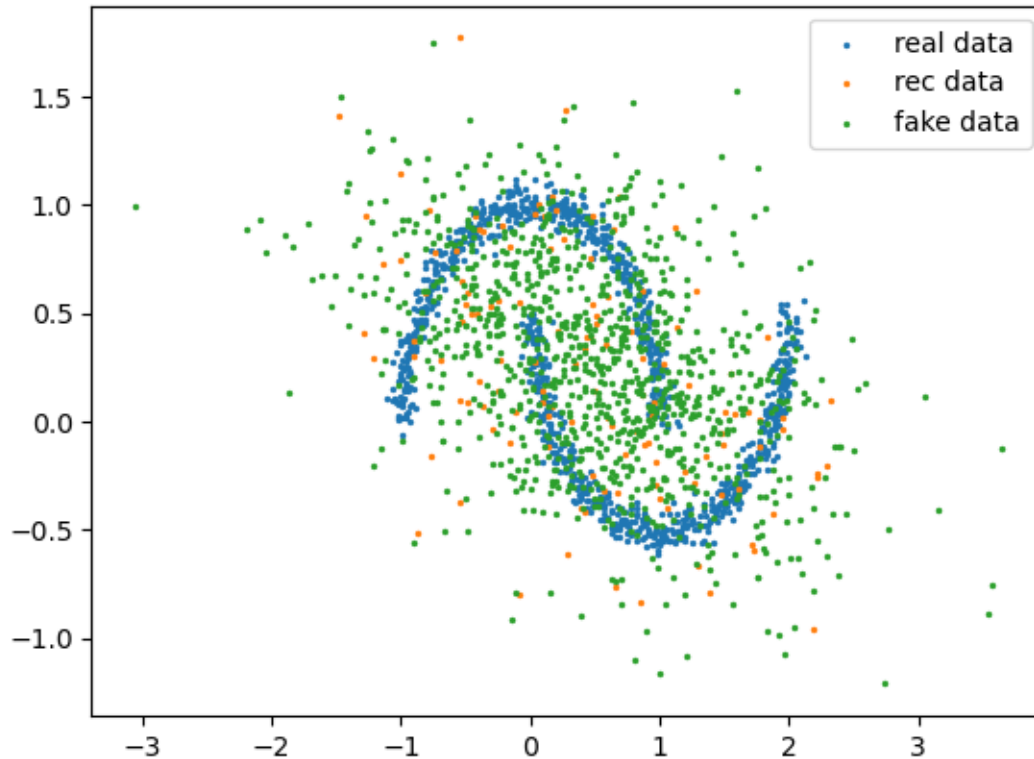
step 20000: KL: 5.057e-01, rec: 1.252e+00



step 30000: KL: 5.001e-01, rec: 1.485e+00



step 40000: KL: 5.030e-01, rec: 1.366e+00



Q12: Try hardcoding $\sigma_\theta(z)$ to some small value (like $1\text{E-}4$) rather than allowing the decoder to learn it. What does it change?

Below is a modified code inspired by the previous approach. We add multiple layers to both the encoder and decoder. In this version, the decoder fixes its sampling variance to 10^{-4} when generating or reconstructing data, forcing the network to focus on learning the mean function precisely.

```
[13]: LATENT_N = 10

class Encoder(nn.Module):
    def __init__(self, n_layers=6, hidden_size=128):
        super().__init__()
        self.n_layers = n_layers
        self.hidden_size = hidden_size

        # Build a sequence of layers for the encoder
        self.layers = [nn.Linear(2, self.hidden_size), nn.ReLU()]
        for _ in range(1, self.n_layers):
            self.layers.append(nn.Linear(self.hidden_size, self.hidden_size))
            self.layers.append(nn.ReLU())
        self.fc = nn.Sequential(*self.layers)
```

```

        # Final layers to produce mu and logvar
        self.fc_mu = nn.Linear(self.hidden_size, LATENT_N)
        self.fc_logvar = nn.Linear(self.hidden_size, LATENT_N)

    def forward(self, x):
        x = self.fc(x)
        mu = self.fc_mu(x)
        logvar = self.fc_logvar(x)
        return mu, logvar

class Decoder(nn.Module):
    def __init__(self, n_layers=6, hidden_size=128):
        super().__init__()
        self.n_layers = n_layers
        self.hidden_size = hidden_size

        # Build a sequence of layers for the decoder
        self.layers = [nn.Linear(LATENT_N, self.hidden_size), nn.ReLU()]
        for _ in range(1, self.n_layers):
            self.layers.append(nn.Linear(self.hidden_size, self.hidden_size))
            self.layers.append(nn.ReLU())
        self.fc = nn.Sequential(*self.layers)

        # Final layers to produce mu and logvar
        self.fc_mu = nn.Linear(self.hidden_size, 2)
        self.fc_logvar = nn.Linear(self.hidden_size, 2)

    def forward(self, z):
        z = self.fc(z)
        mu = self.fc_mu(z)
        logvar = self.fc_logvar(z)
        return mu, logvar

    def generate(self, batchlen):
        # Sample z from N(0, I)
        z = torch.normal(torch.zeros(batchlen, LATENT_N), 1.0)
        (mu, logvar) = self.forward(z)
        # Hardcode sigma = 1e-4
        return torch.normal(mu, 1e-4)

# Notre exemple :

N_ITER = 10001
BATCHLEN = 128

encoder = Encoder()
decoder = Decoder()

```

```

optim_enc = torch.optim.Adam(encoder.parameters(), lr=0.001, betas=(0.5,0.9))
optim_dec = torch.optim.Adam(decoder.parameters(), lr=0.001, betas=(0.5,0.9))

for i in range(N_ITER):
    encoder.zero_grad()
    decoder.zero_grad()

    x = generate_batch(BATCHLEN)

    enc_mu, enc_logvar = encoder(x)
    # KL term
    loss_kl = 0.5 * torch.sum(torch.exp(enc_logvar) + enc_mu.pow(2) - 1 -
    ↪ enc_logvar)

    # Sample z via reparameterization trick
    enc_sig = torch.exp(0.5*enc_logvar)
    z = enc_mu + enc_sig * torch.randn((BATCHLEN, LATENT_N))

    dec_mu, dec_logvar = decoder(z)
    # Reconstruction term (same formula as before)
    loss_rec = 0.5 * torch.sum(
        dec_logvar + (x - dec_mu).pow(2) * torch.exp(-dec_logvar) + LATENT_N *
    ↪ np.log(2 * np.pi)
    )

    # Combine and update
    (0.25 * loss_kl + loss_rec).backward()
    optim_enc.step()
    optim_dec.step()

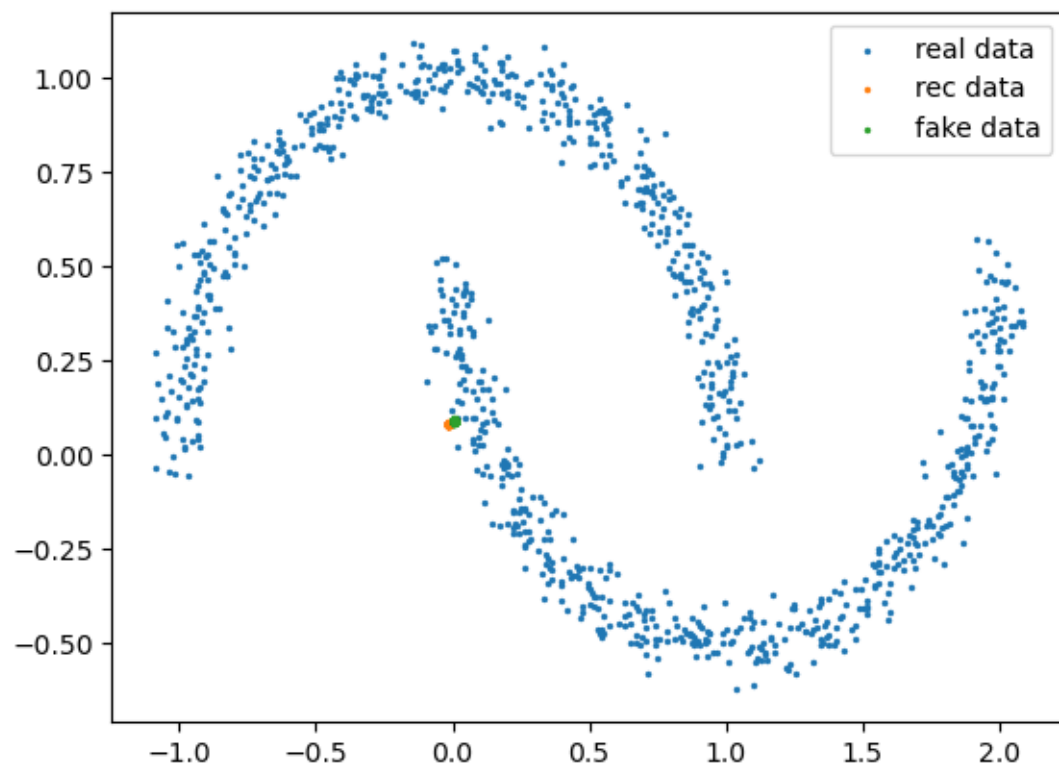
    # Display progress
    if i % 1000 == 0:
        print(f'step {i}: KL: {float(loss_kl):.3e}, rec: {float(loss_rec):.3e}')

        with torch.no_grad():
            real_batch = generate_batch(1024)
            rec_batch = torch.normal(dec_mu, 1e-4)
            fake_batch = decoder.generate(1024)

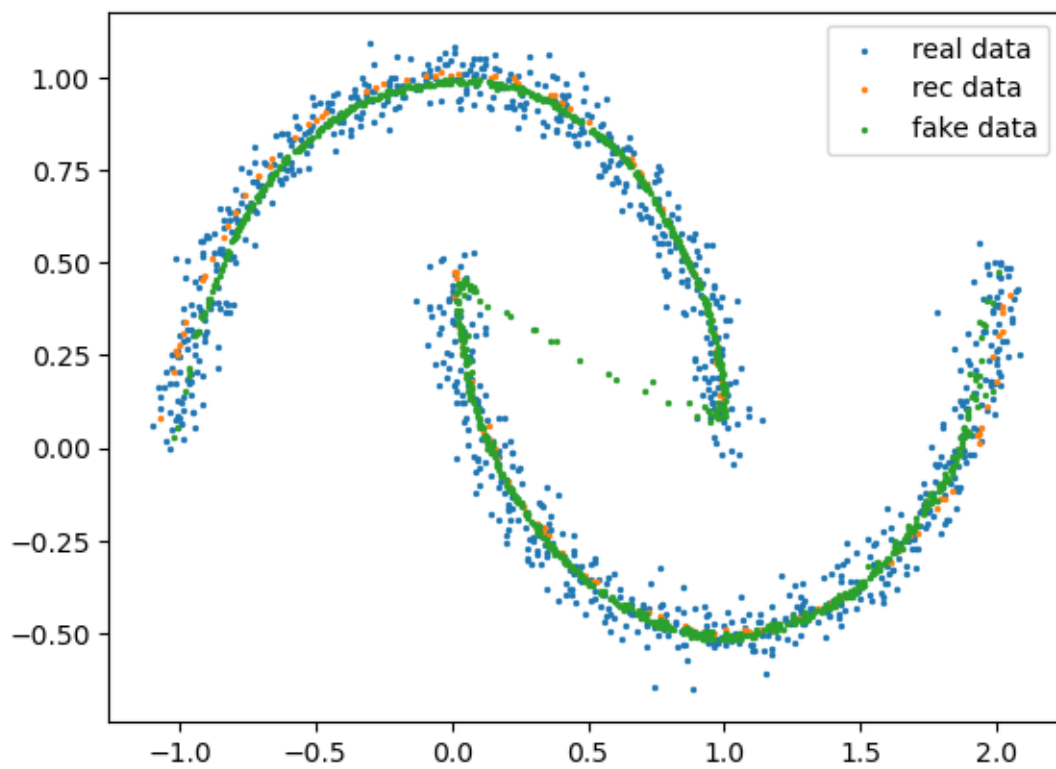
        plt.scatter(real_batch[:,0], real_batch[:,1], s=2.0, label='real data')
        plt.scatter(rec_batch[:,0], rec_batch[:,1], s=2.0, label='rec data')
        plt.scatter(fake_batch[:,0], fake_batch[:,1], s=2.0, label='fake data')
        plt.legend()
        plt.show()

```

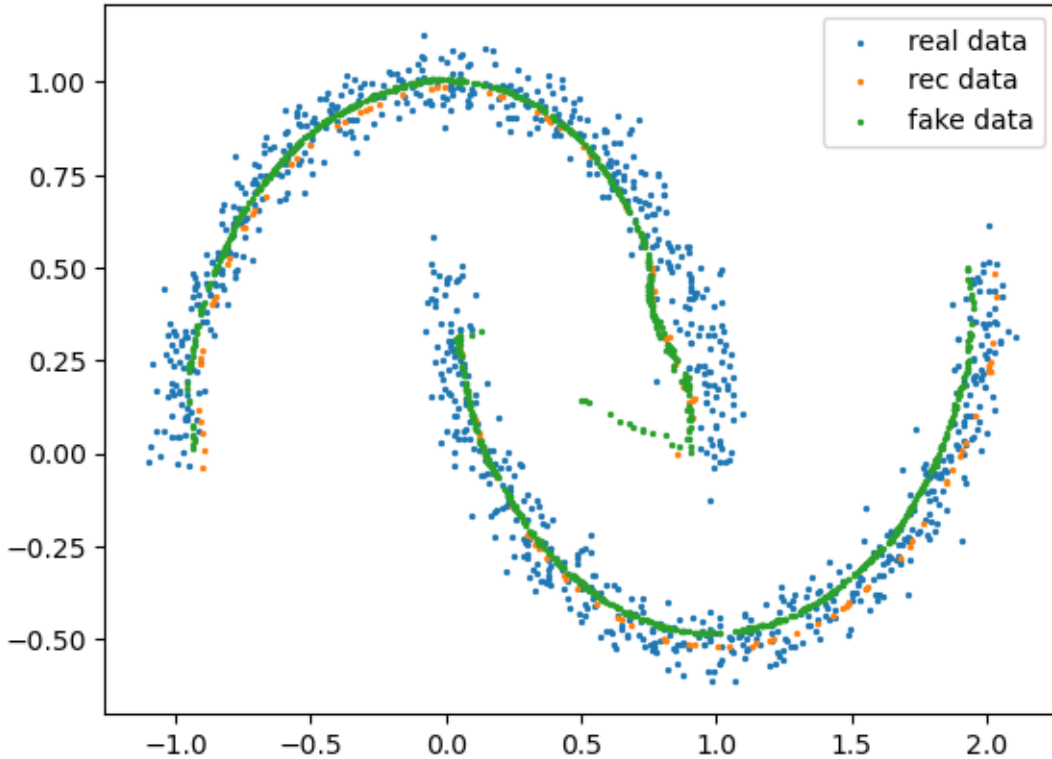
step 0: KL: 2.423e+00, rec: 2.433e+03



step 5000: KL: 5.063e+02, rec: 1.781e+03



step 10000: KL: 5.199e+02, rec: 1.772e+03



Observation:

* We observe that when $\sigma_{\theta}(z)$ is hardcoded to a very small value (e.g., $1E-4$), the decoder is forced to generate outputs almost identical to the learned mean.

- In practice, we tested this configuration over 40,000 epochs. Up until around 33,000 epochs, the model converged very well, and the generated (fake) data closely matched the target distributions.
- However, around 34,000 epochs, instability set in: the model began to “explode,” and the fake data started drifting erratically. These fluctuations created noticeable “waves” in the convergence, as the fixed small variance exacerbated any slight imprecision in the learned mean.
- This behavior suggests that while the model benefits from rapid initial convergence due to the forced closeness of outputs to the mean, the rigidity introduced by a fixed small variance can lead to significant reconstruction errors and erratic behavior in later stages of training.

Q13: How do the power of encoder and decoder affect the overall training of the VAE?

The encoder’s capacity determines how well it can learn a latent representation of the data. If the encoder is too small, it might fail to capture enough information, leading to poor reconstructions.

A larger encoder can better embed the data but may also overfit if not regularized

properly.

The decoder's capacity controls how well the model can reconstruct data from latent codes. A smaller decoder might struggle to generate realistic samples, while an overly large decoder might overfit or ignore the KL term.

In summary, both the encoder and decoder need a balanced capacity. Too little power makes it difficult to capture or reconstruct the data, while too much can lead to overfitting and unstable training.

1.4 Flow Matching

Another approach to generative modeling is given by Normalizing flows. The idea is to learn a mapping f from the data distribution p_D (defined over a space $\mathcal{X}$) to a known distribution (typically, a normal distribution) from which we know how to sample. Two key points are to be noted: 1. The mapping needs to be bijective (i.e., the network needs to be invertible). 2. The exact likelihood of a data point should be easily computable.

For the implementation we will use Flow Matching, an approach for training Continuous Normalizing Flows based on regressing vector fields of fixed conditional probability paths. Before diving into Flow Matching, let's first clarify:

- A **vector field** is a mathematical object that assigns a vector to every point in a space. We can think of it as a function that describes the direction and speed of movement at every possible location.
- A **flow** is the trajectory that a point follows when it moves according to the directions given by the vector field over time. In general, data points evolve when guided by a vector field.

Let us denote $\mathbb{R}v : [0, 1] \times \mathbb{R}^d \rightarrow \mathbb{R}^d$ a time-dependent vector field and $\phi : [0, 1] \times \mathbb{R}^d \rightarrow \mathbb{R}^d$ a flow. A flow is related to a vector field v_t as:

$$\begin{aligned}\frac{d}{dt}\phi_t(x) &= v_t(\phi_t(x)) \\ \phi_0(x) &= x\end{aligned}$$

The main idea of Flow Matching is to use a neural network to model the vector field v_t , which implies that ϕ_t will also be a parametric model simulating a Continuous Normalizing Flow (CNF). In the context of generative modeling, the CNF allows us to transform a simple prior distribution (e.g. Gaussian) p_z into a more complex one p_x , which will be hopefully close enough to the data distribution p_D . To comply with the time-dependent framework, we assume a transition between p_z and p_x in a range $[0, 1]$ so $p_z = p_0$ and $p_x = p_1$. We proceed by using the push-forward equation:

$$p_t = [\phi_t] * p_0$$

where $\forall x \in \mathbb{R}^2$:

$$[\phi_t] * p_0(x) := p_0(\phi_t^{-1}(x)) \det \left[\frac{\delta \phi_t^{-1}}{\delta x}(x) \right]$$

So the flow ϕ acts as the mapping f in this context. If you are not familiar with images of distributions, the definition is here: https://en.wikipedia.org/wiki/Pushforward_measure ; it can

be defined as the measure that satisfies this change of variables: $\int_{z \sim p_N} g(f^{-1}(z)) dz = \int_{x \sim p_G} g(x) dx$ for all functions g . Another notation for this is $\int_z g(f^{-1}(z)) dp_N(z) = \int_x g(x) dp_G(x)$.

Thus Flow Matching aims to learn the true vector field u_t :

$$L_{CFM}(\theta) = \mathbb{E}_{t, p_D(x_1), p_t(x|x_1)} \|v_t(x) - u_t(x|x_1)\|^2$$

> **Attention! Mind the difference in notation between the true vector u and its estimation v .**

1.4.1 Optimal Transport Conditional Vector Fields

We will use the simplest approach where the flow ϕ is a linear displacement.

Let $x_1 \sim p_D(x_1)$ be a point from the dataset. We can define the conditional flow ψ_t (conditional version of ϕ_t) to be the Optimal Transport displacement map between the two Gaussians $p_0(x|x_1)$ and $p_1(x|x_1)$, that is:

$$\psi_t(x) = (1 - (1 - \sigma_{min})t)x + tx_1 \quad (1)$$

(we can prove that ψ_t satisfies the push-forward equation).

In the case of Optimal Transport conditional VFs the conditional flow matching objective loss can be written as follows:

$$L_{CFM}(\theta) = \mathbb{E}_{t, p_D(x_1), p(x_0)} \|v_t(\psi_t(x_0)) - \frac{d}{dt}\psi_t(x_0)\|^2$$

Q14: Express the derivative of $\psi_t(x_0)$, to simplify the formula of $L_{CFM}(\theta)$.

(Write your answer here)

Q15: Calculate the determinant of the Jacobian of the flow $\phi_t(x)$.

(Write your answer here)

Q16: Using the Jacobian determinant, prove that $\psi_t(x)$ is indeed invertible.

(Write your answer here)

Q17: Prove that $\psi_t(x)$ satisfies the push-forward equation. That means that it applies scaling and shifting to the distribution $p_0(x)$.

(Write your answer here)

We sample $x_1 \sim p_D(x_1)$ (samples from the dataset), $x_0 \sim p(x_0)$ (samples from the prior distribution) and $t \sim \text{Unif}([0, 1])$. We need to apply a little trick for time sampling since the batch covers the range $[0, 1]$ uniformly with a random offset. Then we compute an approximation of the CFM Loss on the batch:

$$L_{batch}(\theta) = \frac{1}{N} \sum_{i=0}^{N-1} \|v_t(\psi_t(x_0^{(i)})) - (x_1^{(i)} - (1 - \sigma_{min})x_0^{(i)})\|^2$$

Q18: Fill the Coupling layer structure below by defining suitable functions `sfun` and `tfun` (using a final `tanh` in `sfun` is highly recommended) and the inverse pass

```
[ ]: class OptimalTransportFlowMatching:
    def __init__(self, sig_min: float = 0.001) -> None:
        super().__init__()
        self.sig_min = sig_min
        self.eps = 1e-5

    def psi_t(self, x: torch.Tensor, x_1: torch.Tensor, t: torch.Tensor) -> torch.Tensor:
        """ Conditional Flow """
        #
        # COMPUTE THE CONDITIONAL FLOW HERE
        #
        return None

    def loss(self, v_t: nn.Module, x_1: torch.Tensor) -> torch.Tensor:
        """ Compute loss """
        # t ~ Unif([0, 1])
        # the batch covers the range [0,1] uniformly with a random offset
        t = (torch.rand(1, device=x_1.device) + torch.arange(len(x_1),
->device=x_1.device) / len(x_1)) % (1 - self.eps)
        t = t[:, None].expand(x_1.shape) # turn 2d

        #
        # SAMPLE THE PRIOR DISTRIBUTION HERE
        # x_0 ~ p(x_0)
        #

        #
        # CALCULATE THE BATCH LOSS HERE
        #
        return None
```

1.4.2 Conditional Vector Field

We add the Conditional Vector Field class which aims to model the Optimal Transport Conditional Vector Field v_t . This approximation $v_t(\cdot; \theta)$ of the true v_t will be optimized by tuning the parameters θ of ConditionalVectorField.

Once optimized, we need to sample from the learned flow, thus we define the encode and decode functions: - **encode** maps the data distribution $p_D(x_1) \approx p_1(x_1)$ to the prior distribution $p(x_0)$. - **decode** maps the prior distribution $p(x_0)$ to the data distribution $p_1(x_1) \approx p_D(x_1)$.

A flow ϕ_t and its corresponding vector field v_t verify the following equations:

$$\frac{d}{dt}\phi_t(x) = v_t(\phi_t(x))$$

$$\phi_0(x) = x$$

Thus, to encode we integrate the first equation from $t = 1$ to $t = 0$ and the decode function is the same but going from $t = 0$ to $t = 1$. We use the Euler method to encode and decode.

```
[ ]: class ConditionalVectorField(nn.Module):
    def __init__(self, net: nn.Module, n_steps: int = 100, dt: float = 0.01) -> None:
        super().__init__()
        self.net = net
        self.n_steps = n_steps
        self.dt = dt

    def forward(self, t: torch.Tensor, x: torch.Tensor) -> torch.Tensor:
        return self.net(t, x)

    def wrapper(self, t: torch.Tensor, x: torch.Tensor) -> torch.Tensor:
        t = t * torch.ones(len(x), device=x.device)
        return self(t, x)

    def encode(self, x_1: torch.Tensor) -> torch.Tensor:
        x = x_1
        t = torch.ones(x.shape[0], device=x.device) # Start from t=1
        for _ in range(self.n_steps):
            v = self.wrapper(t, x)
            x = x - self.dt * v # Reverse Euler update
            t = t - self.dt # Decrement time
        return x

    def decode(self, x_0: torch.Tensor) -> torch.Tensor:
        x = x_0
        t = torch.zeros(x.shape[0], device=x.device)
        for _ in range(self.n_steps):
            v = self.wrapper(t, x)
            x = x + self.dt * v # Euler update
            t = t + self.dt # Step increment
        return x
```

1.4.3 Neural Network for VF approximation

We want to approximate v_t with a parametric model $v_t(\cdot; \theta)$. Since we deal with point coordinates and want to keep the neural network simple for this example, we use a simple Multilayer Perceptron.

However the vector field v is conditioned on time t . Thus, we create a sinusoidal positional encoding of t thanks to sine and cosine functions to have a time representation that is easier to exploit than just its value between 0 and 1.

When computing $v_t(x)$ we simply concatenate the sinusoidal positional encoding of t with x (input coordinates), and then process the obtained tensor with the MLP's layers.

```
[ ]: class Net(nn.Module):
    def __init__(self, in_dim: int, out_dim: int, h_dims: list[int],
        ↪n_frequencies:int) -> None:
        super().__init__()
        ins = [in_dim + 2 * n_frequencies] + h_dims
        outs = h_dims + [out_dim]
        self.n_frequencies = n_frequencies

        self.layers = nn.ModuleList([
            nn.Sequential(nn.Linear(in_d, out_d), nn.LeakyReLU()) for in_d,
        ↪out_d in zip(ins, outs)
        ])
        self.top = nn.Sequential(nn.Linear(out_dim, out_dim))

    def time_encoder(self, t: torch.Tensor) -> torch.Tensor:
        freq = 2 * torch.arange(self.n_frequencies, device=t.device) * torch.pi
        t = freq * t[..., None]
        return torch.cat((t.cos(), t.sin()), dim=-1)

    def forward(self, t: torch.Tensor, x: torch.Tensor) -> torch.Tensor:
        t = self.time_encoder(t)
        x = torch.cat((x, t), dim=-1)

        for l in self.layers:
            x = l(x)
        return self.top(x)
```

1.4.4 Train

```
[ ]: # globals
BATCH_SIZE = 128
LOG_INTERVAL = 50
N_EPOCHS = 10
INPUT_SIZE = 2
OUTPUT_SIZE = 2
HIDDEN_SIZE = 256
N_LAYERS = 5
TIME_FREQUENCY = 10

[ ]: # data loading
train_data = generate_batch(50000)
test_data = generate_batch(1000)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
loader_kwargs = {"num_workers": 1, "pin_memory": True} if device == "cuda" else
    ↪{}
train_loader = torch.utils.data.DataLoader(train_data, batch_size=BATCH_SIZE,
    ↪shuffle=True)
```

```
test_loader = torch.utils.data.DataLoader(test_data, batch_size=BATCH_SIZE,
↳shuffle=True)
```

```
[ ]: model = OptimalTransportFlowMatching()
net = Net(INPUT_SIZE, OUTPUT_SIZE, [HIDDEN_SIZE]*N_LAYERS, TIME_FREQUENCY).
↳to(device)
v_t = ConditionalVectorField(net)

losses = []
# configure optimizer
optimizer = torch.optim.Adam(v_t.parameters(), lr=1e-3)
n_epochs = 5000

for epoch in range(N_EPOCHS):
    for batch_idx, x_inputs in enumerate(train_loader):
        x_1 = x_inputs.to(device)
        # compute loss
        loss = model.loss(v_t, x_1)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        losses += [loss.detach()]
        if batch_idx % LOG_INTERVAL == 0:
            print("Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}".format(
                epoch, batch_idx * len(x_inputs), len(train_data),
                100. * batch_idx / len(train_loader), loss / len(x_inputs)))
```

1.4.5 Sampling

To sample $\hat{x}_1 \sim p_D(x_1)$, we first sample $x_0 \sim p(x)$, drawing samples from the prior distribution, and then apply the decode function from the ConditionalVectorField class.

```
[ ]: #
# SAMPLE THE PRIOR AND DECODE HERE
#
```

```
[ ]: #
# PLOT THE DENSITY HERE
#
```

Q19: Explain why Normalizing flows do not fit well when there are many clusters in the data.

(Write your answer here)

1.5 Denoising Diffusion Model

Recently Diffusion Models received a lot of attention. **Diffusion Models Beat GANs on Image Synthesis** (*Dhariwal, Nichol*) : [arXiv:2105.05233](https://arxiv.org/abs/2105.05233) shows significant improvement on image generation with a model claimed to be easier to train than GANs.

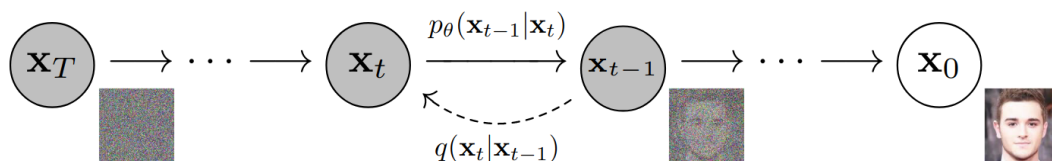
It is the key of the image generating process behind Latent Diffusion models such as Dall.e, Imagen or StableDiffusion.

We are going to implement a model inspired by the paper **Denoising Diffusion Probabilistic Models** (*Ho et al.*): [arXiv:2006.11239](https://arxiv.org/abs/2006.11239)

Denoising diffusion probabilistic model introduce noise into data and gradually learn to reverse this process for generating new samples. It involves two Markov chains, one that forwards data to noise, and another that reverses this process by converting the noise back to data. The forward chain is usually designed to transform any data distribution into a simpler prior distribution, such as a standard Gaussian distribution, while the reverse Markov chain is parametrized by a neural network and learns to reverse this process. To generate new data points, we first sample a random vector from the prior distribution, then applies ancestral sampling through the reverse Markov chain.

References:

- Survey on Diffusion Models : <https://arxiv.org/pdf/2209.00796.pdf>
- <https://lilianweng.github.io/posts/2021-07-11-diffusion-models/>
- <https://blog.alexalemi.com/diffusion.html>
- <https://sander.ai/posts/>



q and p_θ being the respective kernels of the two chains.

Forward Pass For our purpose we are going to use a Gaussian Noise. The forward Kernel will have the following shape :

$$q(x_t|x_{t-1}) = \mathcal{N}(x_t; \sqrt{1 - \beta_t}x_{t-1}, \beta_t I)$$

with $\mathcal{N}(x; \mu, \sigma I)$ a conditional gaussian :

$$y \sim \mathcal{N}(x; \mu, \sigma I) \equiv y|x \sim \mathcal{N}(\mu, \sigma I)$$

Q20: What does β_t represent in the formula ?

(Write your answer here)

In our example the sequence β_t will be chosen linearly wrt t .

We introduce $\alpha_t := 1 - \beta_t$ and $\bar{\alpha}_t = \prod_{s=1}^t \alpha_s$ we can sample forward directly at timestep t from the original image :

$$q(\mathbf{x}_{1:T}|\mathbf{x}_0) = \prod_{t=1}^T q(\mathbf{x}_t|\mathbf{x}_{t-1})$$

$$q(\mathbf{x}_t|\mathbf{x}_0) = \mathcal{N}(\mathbf{x}_t; \sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1 - \bar{\alpha}_t)I)$$

i.e.

$$\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1 - \bar{\alpha}_t}\epsilon \quad \text{where } \epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \quad (2)$$

we will then fix

$$\sigma_t = \sqrt{1 - \bar{\alpha}_t}$$

Q22: Implement the forward pass using $\bar{\alpha}_t$ (Hint : you may need to precalculate the $\bar{\alpha}_t$ from β_t)

```
[ ]: def bcast_right(x: torch.Tensor, ndim: int) -> torch.Tensor:
    """Util function for broadcasting to the right."""
    if x.ndim > ndim:
        raise ValueError(f'Cannot broadcast a value with {x.ndim} dims to {ndim}
        ↪dims.')
    elif x.ndim < ndim:
        difference = ndim - x.ndim
        return x.view(x.shape + (1,) * difference)
    else:
        return x
```

```
[ ]: class DiscreteDDPMPProcess:
    """A Gaussian diffusion process:  $q(x_t|x_0) = N(\text{sqrt\_alpha\_bar}(t)*x_0,$ 
    ↪ $\text{sigma}(t)^2 * I)$ ,
    which implies the following transition from  $x_0$  to  $x_t$ :

     $x_t = \text{sqrt\_alpha\_bar}(t) x_0 + \text{sigma}(t) \text{eps}$ ,  $\text{eps} \sim N(0, I)$ .

    Diffusion processes differ in how they specify  $\text{sqrt\_alpha\_bar}(t)$  and/or
    ↪ $\text{sigma}(t)$ .
    Here we follow the DDPM paper.

    """
    def __init__(
        self,
        num_diffusion_timesteps: int = 1000,
        beta_start: float = 0.0001,
        beta_end: float = 0.02,
    ):
        self._num_diffusion_timesteps = num_diffusion_timesteps
        self._beta_start = beta_start
```

```

        self._beta_end = beta_end
        self._betas = np.linspace(self._beta_start, self._beta_end, self.
→_num_diffusion_timesteps)

        alphas_bar = self._get_alphas_bar()
        #####
        # TO COMPLETE #
        #####
        self._sqrt_alphas_bar = ? #put in dtype=torch.float32
        self._sigmas = ? #put in dtype=torch.float32

    @property
    def tmin(self):
        return 1

    @property
    def tmax(self):
        return self._num_diffusion_timesteps

    def _get_alphas_bar(self) -> np.ndarray:
        #####
        # TO COMPLETE #
        #####
        alphas_bar = ?

        # we can add this 1 in front to simplify indexing,
        # and to make alpha[0]=1 and sigma[0]=0.
        # these values at t=0 will be needed later when generating samples
        alphas_bar = np.concatenate([1., alphas_bar])

        return alphas_bar

    def sqrt_alpha_bar(self, t: torch.Tensor) -> torch.Tensor:
        return self._sqrt_alphas_bar[t.long()]

    def sigma(self, t: torch.Tensor) -> torch.Tensor:
        return self._sigmas[t.long()]

    def sample(self, x0: torch.Tensor, t: torch.Tensor, eps: torch.Tensor) ->
→torch.Tensor:
        """Draws samples from the forward diffusion process  $q(x_t|x_0)$ ."""
        #####
        # TO COMPLETE #
        #####
        return

```

```
[ ]: # data loading
test_data = generate_batch(1000)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

[ ]: process = DiscreteDDPMPProcess(num_diffusion_timesteps=1000)
ts = torch.Tensor(np.linspace(process.tmin, process.tmax, num=process.tmax,
    dtype=np.int32))

[ ]: fig, ax = plt.subplots(1, 2, figsize=(4 * 2, 2))

# plot how sqrt_alpha_bar and sigma change over time
ax[0].plot(ts.numpy(), process.sigma(ts), label=r'$\sigma$')
ax[0].plot(ts.numpy(), process.sqrt_alpha_bar(ts), label=r'$\sqrt{\bar{\alpha}}$')
ax[0].set_xlabel('time')
ax[0].legend()

# It is often helpful to reason in terms of signal-to-noise ratio:
# SNR = sqrt_alpha_bar(t)^2 / sigma(t)^2 or more conveniently, its logarithm.
# High SNR = little noise, low SNR = a lot of noise.
lambda_ = [2. * np.log(process.sqrt_alpha_bar(t) / process.sigma(t)) for t in ts]
ax[1].plot(ts.numpy(), lambda_, c='g', label=r'$\log$ SNR')
ax[1].set_xlabel('time')
ax[1].legend()

plt.show()

[ ]: # Effect on the distribution
torch_test_data = torch.Tensor(test_data)
def apply_on_dataset(dataset, t):
    dataset_t = torch.zeros_like(dataset)
    for i in range(dataset.shape[0]):
        point = dataset[i]
        eps = torch.randn_like(point)
        point_t = process.sample(point, t, eps)
        dataset_t[i] = point_t
    return dataset_t

fig, axes = plt.subplots(1, 5, figsize=(20, 5))
axes[0].scatter(torch_test_data[:, 0], torch_test_data[:, 1])
for i in range(1, 5):
    t = torch.Tensor([i * 50]).type(torch.int64)
    dataset_t = apply_on_dataset(torch_test_data, t)
    axes[i].scatter(dataset_t[:, 0], dataset_t[:, 1])
    axes[i].set_title(f"After {t.detach().item()} walks")
fig.suptitle("Influence of the forward pass on the distribution")
plt.show()
```

Backward Process Hypothetically, We would like to train the reverse kernel $p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t)$, with parameters θ parametrized by a neural network

$$p_\theta(\mathbf{x}_{0:T}) := p(\mathbf{x}_T) \prod_{t=1}^T p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t)$$

with $p(\mathbf{x}_T)$ a Gaussian prior, $p(\mathbf{x}_T) \sim \mathcal{N}(0, I)$. We would use $p(\mathbf{x}_T)$ to sample new data points for generation.

Variational Lower Bound In the paper, they used the knowledge of the forward pass and design a Gaussian reverse kernel

$$p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t) = \mathcal{N}(\mathbf{x}_{t-1}; \mu_\theta(\mathbf{x}_t, t), \Sigma_\theta(\mathbf{x}_t, t))$$

with μ_θ and Σ_θ parametrized by a neural network

We can write the previous forward kernel

$$\begin{aligned} q(\mathbf{x}_t \mid \mathbf{x}_0) &= \mathcal{N}(\mathbf{x}_t; \sqrt{\bar{\alpha}_t} \mathbf{x}_0, (1 - \bar{\alpha}_t) \mathbf{I}) \\ &= \sqrt{\bar{\alpha}_t} \mathbf{x}_0 + \epsilon \sqrt{1 - \bar{\alpha}_t} \end{aligned}$$

with $\epsilon \sim \mathcal{N}(0, \mathbf{I})$

The natural loss would be to optimize under the variational lower bound L_t such that :

$$\begin{aligned} L_{\text{vlb}} &:= L_0 + L_1 + \dots + L_{T-1} + L_T \\ L_0 &:= -\log p_\theta(x_0 \mid x_1) \\ L_{t-1} &:= D_{KL}(q(x_{t-1} \mid x_t, x_0) \parallel p_\theta(x_{t-1} \mid x_t)) \\ L_T &:= D_{KL}(q(x_T \mid x_0) \parallel p(x_T)) \end{aligned}$$

One can show that an alternative loss can be used that only take one parametrized estimator ϵ_θ . Hint: KL divergences between Gaussian distributions can be computed analytically

$$L_{\text{simple}}(\theta) := \mathbb{E}_{t, \mathbf{x}_0, \epsilon} \left[\left\| \epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_t} \mathbf{x}_0 + \sqrt{1 - \bar{\alpha}_t} \epsilon, t) \right\|^2 \right]$$

Let's start implementing the network that predicts $\hat{\epsilon}(x_t, t; \theta)$. Note that, as stated by its definition, ϵ is the cumulated noise from $t = 0$ and not just the noise added at the last step.

```
[ ]: import torch.nn as nn

class ResidualMLP(nn.Module):
    """MLP with residual connections."""

    def __init__(
        self,
        n_blocks: int,
        n_hidden: int,
```

```

        n_out: int,
        activation: str,
        name: str = None
    ):
        super(ResidualMLP, self).__init__()
        self._n_blocks = n_blocks
        self._n_hidden = n_hidden
        self._n_out = n_out
        self._activation = getattr(nn.functional, activation)

        self.linear_input = nn.Linear(n_out, n_hidden)
        self.linear_time = nn.ModuleList([nn.Linear(n_hidden, n_hidden) for _ in
        ↪range(self._n_blocks)])
        self.linear_hidden = nn.Sequential(
            nn.ModuleList([nn.Linear(n_hidden, n_hidden) for _ in range(self.
        ↪_n_blocks)]),
            nn.ModuleList([nn.Linear(n_hidden, n_hidden) for _ in range(self._n_blocks)])
        )
        self.linear_output = nn.Linear(n_hidden, n_out)

    def forward(self, xt: torch.Tensor, time: torch.Tensor) -> torch.Tensor:
        x = self.linear_input(xt)

        for _ in range(self._n_blocks):
            h=self._activation(x)
            h=self.linear_hidden[_][0](h)
            h=h+self.linear_time[_](time)
            h=self._activation(h)
            h=self.linear_hidden[_][1](h)
            x=x+h

        outputs = self.linear_output(x)

        return outputs

```

Time embeddings As we can see above, the denoiser network needs to get timestep t as an input.

However, feeding integers $t = 1, 2, \dots, T$ into the network will not work.

Transformers, when faced with the same problem, proposed to embed timesteps into d -dimensional vectors of sinusoids:

$$e = [\sin(w_1 t) \quad \cos(w_1 t) \quad \dots \quad \sin(w_{d/2} t) \quad \cos(w_{d/2} t)]$$

with $w_i = 1/10000^{\frac{2i}{d}}$.

Here is nice blogpost with more intuition: https://kazemnejad.com/blog/transformer_architecture_positional_encoding/

```
[ ]: class SinusoidalTimeEmbedding(nn.Module):
    """Time (positional) embedding as in Transformers."""

    def __init__(self, num_features: int, name: str = None):
        super(SinusoidalTimeEmbedding, self).__init__()
        self._num_features = num_features

    def forward(self, inputs: torch.Tensor) -> torch.Tensor:
        assert len(inputs.shape) == 1
        half_dim = self._num_features // 2
        e = torch.log(torch.tensor(10000.0)) / (half_dim - 1)
        embedding = torch.exp(-e * torch.arange(half_dim).float()).to(device)
        embedding = inputs.view(-1, 1) * embedding
        embedding = torch.cat([torch.cos(embedding), torch.sin(embedding)],
        ↪dim=-1)
        if self._num_features % 2 == 1:
            embedding = nn.functional.pad(embedding, (0, 1))
        return embedding
```

Now we need to put the two modules together:

```
[ ]: import dataclasses

@dataclasses.dataclass
class NetConfig:
    resnet_n_blocks: int = 2
    resnet_n_hidden: int = 256
    resnet_n_out: int = 2
    activation: str = 'elu'
    time_embedding_dim: int = 256

class Net(nn.Module):
    """Combines MLP and time embeddings."""
    def __init__(self, net_config: NetConfig, name: str = None):
        super(Net, self).__init__()

        self._time_encoder = SinusoidalTimeEmbedding(net_config.
        ↪time_embedding_dim)
        self._predictor = ResidualMLP(
            n_blocks=net_config.resnet_n_blocks,
            n_hidden=net_config.resnet_n_hidden,
            n_out=net_config.resnet_n_out,
            activation=net_config.activation
        )
```

```

def forward(self, noisy_data: torch.Tensor, time: torch.Tensor) -> torch.
↳Tensor:
    time_embedding = self._time_encoder(time)
    outputs = self._predictor(noisy_data, time_embedding)
    return outputs

```

Time sampler To compute our loss function, for each example, we need to sample a random timestep. In this trivial case of uniform sampling between step 1 and T, it might be an overkill to implement it as a separate class. It would be more helpful if you're going to explore more exotic diffusion models.

```

[ ]: from typing import Sequence

class UniformDiscreteTimeSampler:

    def __init__(self, tmin: int, tmax: int):
        self._tmin = tmin
        self._tmax = tmax

    def sample(self, shape: Sequence[int]) -> torch.Tensor:
        return torch.randint(low=self._tmin, high=self._tmax, size=shape)

```

Full DDPM model Q.23 Implement the Simplified loss given a model ϵ_θ

```

[ ]: class DiffusionModel(nn.Module):
    """Diffusion model."""

    def __init__(self, diffusion_process, time_sampler, net_config, data_shape):
        super(DiffusionModel, self).__init__()

        self._process = diffusion_process
        self._time_sampler = time_sampler
        self._net_config = net_config
        self._data_shape = data_shape
        self.net_fwd = Net(net_config)

    def loss(self, x0: torch.Tensor) -> torch.Tensor:
        """Computes MSE between the true noise and predicted noise,
        i.e. the goal of the network is to correctly predict eps from a noisy_
        ↳observation
        xt = alpha(t) * x0 + sigma(t)**2 * eps"""

        #####
        # TO COMPLETE #
        #####

        t = ? # sample time

```

```

    eps = ? # sample noise

    xt = ? # corrupt the data

    net_outputs = ? # get net outputs

    loss = ? # compute MSE loss between predicted and true noise

    return loss

#Used for sampling
def _reverse_process_step(
    self,
    xt: torch.Tensor,
    t: int,
) -> Tuple[torch.Tensor, torch.Tensor]:
    """Computes parameters of a Gaussian  $p_{\theta}(x_{t-1}|x_t)$ ."""

    t = t * torch.ones((xt.shape[0],), dtype=torch.int32, device=xt.device)

    #####
    # TO COMPLETE #
    #####

    eps_pred = ? # predict epsilon from x_t

    sqrt_alpha_t = ? # use self._sqrt_alpha_bar
    inv_sqrt_alpha_t = bcast_right(1.0 / sqrt_alpha_t, xt.ndim)

    beta_t = ?
    beta_t = bcast_right(beta_t, xt.ndim)

    inv_sigma_t = ?
    inv_sigma_t = bcast_right(inv_sigma_t, xt.ndim)

    mean = ?

    # DDPM instructs to use either the variance of the forward process
    # or the variance of  $q(x_{t-1}|x_t, x_0)$ . Former is easier.
    std = ?

    eps = ?

    return ?

```

```

def sample(self, x0, sample_size):
    """To generate samples from DDPM, we follow the reverse process.
    At each step of the chain, we sample  $x_{t-1}$  from  $p(x_{t-1} | x_t, x0\_pred)$ 
    →until we get to  $x_0$ ."""
    with torch.no_grad():
        x = torch.randn((sample_size,) + self._data_shape, device=x0.device)
    →#sample pure noise

    #####
    # TO COMPLETE #
    #####

    return x

```

Given the noise obtained by the model $\epsilon_\theta(\mathbf{x}_t, t)$, you can sample $\mu_\theta(\mathbf{x}_t, t)$ in order to get a denoised sample point.

$$\mu_\theta(\mathbf{x}_t, t) = \frac{1}{\sqrt{\alpha_t}} \left(\mathbf{x}_t - \frac{\beta_t}{\sqrt{1 - \bar{\alpha}_t}} \epsilon_\theta(\mathbf{x}_t, t) \right)$$

The `_reverse_process_step` function sample $x_{t-1} \sim p_\theta(\mathbf{x}_{t-1} | \mathbf{x}_t)$ where

$$p_\theta(\mathbf{x}_{t-1} | \mathbf{x}_t) = \mathcal{N}(\mathbf{x}_{t-1}; \mu_\theta(\mathbf{x}_t, t), \Sigma_\theta(\mathbf{x}_t, t))$$

and we set

$$\Sigma_\theta(\mathbf{x}_t, t) = \beta_t \mathbf{I}$$

Thus

$$x_{t-1} = \mu_\theta(\mathbf{x}_t, t) + \sqrt{\beta_t} \epsilon$$

where

$$\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$$

Q.24 Build the sampling function that allows you from inputs $\mathbf{x}_t$ and t to obtain $\mu_\theta(\mathbf{x}_t, t)$ and x_{t-1}

Training

```

[ ]: from torch import optim

# create the model
diffusion_process = DiscreteDDPMPProcess(num_diffusion_timesteps=1000)
time_sampler = UniformDiscreteTimeSampler(diffusion_process.tmin,
    →diffusion_process.tmax)
model = DiffusionModel(diffusion_process, time_sampler, net_config=NetConfig(),
    →data_shape=(2,)).to(device)
optimizer = optim.Adam(model.parameters(), lr=3e-4)

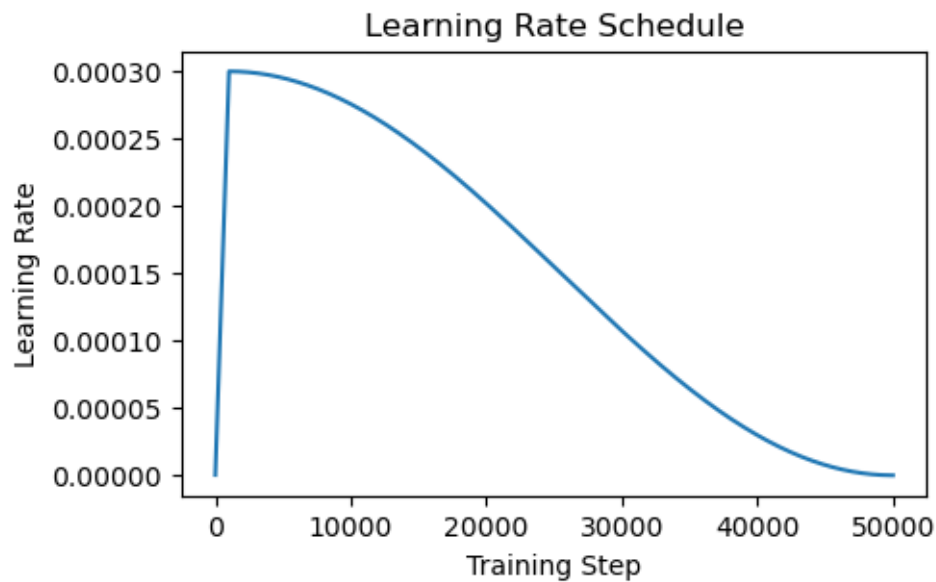
```

We will use a learning rate scheduler with a warmup.

You will have to install the library `pytorch_warmup` with “`pip install -U pytorch_warmup`”:

https://github.com/Tony-Y/pytorch_warmup

Our learning rate schedule will have the following shape.



```
[ ]: import pytorch_warmup as warmup

training_steps = 50000

warmup_period=1000 #this is the increasing part of the learning rate schedule
num_steps = training_steps
t0 = num_steps // 1
lr_min = 3e-12
max_step = t0 * 1 + warmup_period #you can also create cycle but we won't touch
    → it here

lr_scheduler = optim.lr_scheduler.CosineAnnealingWarmRestarts(
    optimizer, T_0=t0, T_mult=1, eta_min=lr_min)

warmup_scheduler = warmup.LinearWarmup(optimizer, warmup_period)
```

Training loop Q.25 Train $\epsilon_\theta(x, t)$ that take as entry a transformed vector x and t a number of time x passed through the forward process and return a direct estimate of the noise ϵ

```
[ ]: BATCH_SIZE=1024

for step in range(training_steps):
    #generate batch on the fly

    #####
```

```

# TO COMPLETE #
#####

if step % 1000 == 0:
    print(f'Step: {step}, Loss: {loss:.5f}')

if step % 10000 == 0:
    with torch.no_grad():
        samples = ? # create new samples
        plt.figure(figsize=(3, 3))
        plt.scatter(samples[:, 0], samples[:, 1], s=1, alpha=0.5)
        plt.show()

with warmup_scheduler.dampening():
    if warmup_scheduler.last_step + 1 >= warmup_period:
        lr_scheduler.step()
    if warmup_scheduler.last_step + 1 >= max_step:
        break

```

Ablation study Q.26 Perform ablations experiments on hyperparameters, such as: learning rate (try constant learning rate with different values), batch size, the number of timesteps, the positional embedding for timestep (remove it), the hidden size, number of epochs, skip connections

For a given ablation experiment (they ought to be done separately!), you should plot generated samples at few given timesteps (if compute resources is limited, do it for one timestep).

Note: you don't have to do all of these experiments, but the bigger is your group, the more you should do.

[]:

(Write your answer here)

Generation/Sampling Q.27 Generate samples and plot them at different timestep of the chain. Check how real & synthetic distribution overlap, compare the densities

[]:

1.6 Conclusion

Q.28: As a conclusion, how would you compare the advantages and shortcomings of GANs, VAEs, Flow Matching and Denoising Models ?

- **GANs:** Often create very sharp and realistic samples but can be unstable during training and may suffer from mode collapse.

- **VAEs**: Are easier and more stable to train, yield an interpretable latent space, but sometimes produce blurrier samples.

We only experimented with **GANs** and **VAEs** in our work, so our comparison is limited to these two approaches. While they both show promise for generative tasks, they have contrasting trade-offs in terms of sample quality versus training stability and latent-space interpretation.