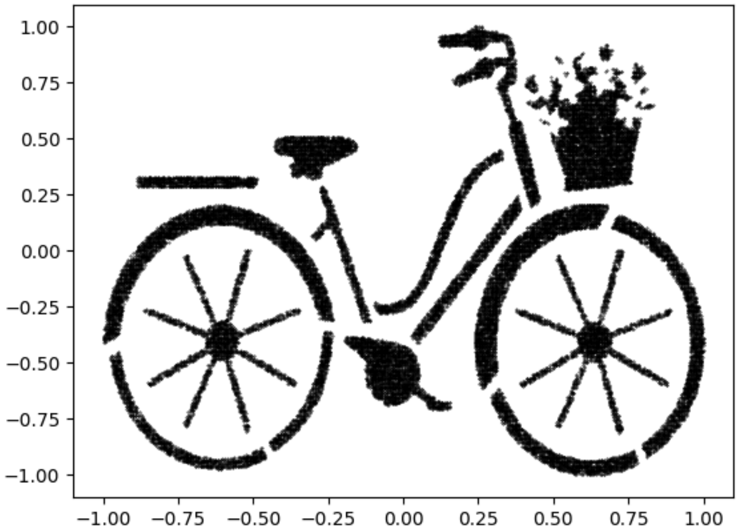
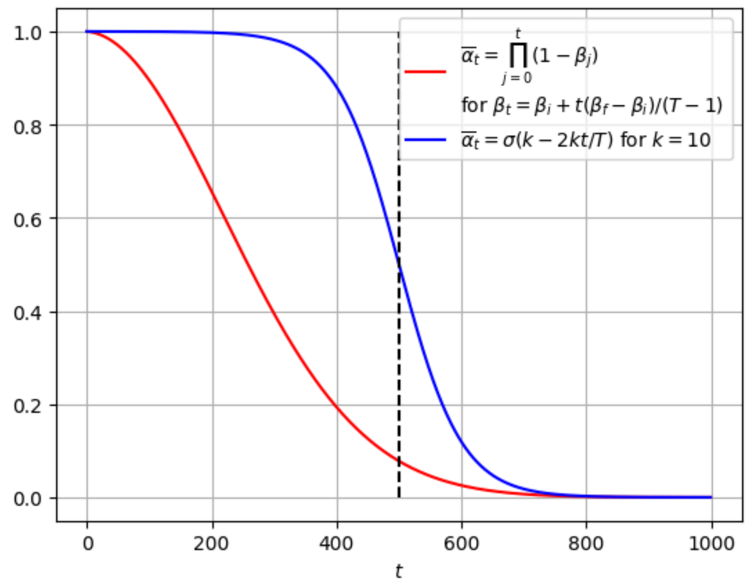
Bartosz Hanc

Zadanie 5.

Zadanie wykonałem korzystając z frameworka PyTorch. Po wczytaniu zbioru danych z pliku tekstowego narysowałem na płaszczyźnie 30 000 losowo wybranych przykładów.



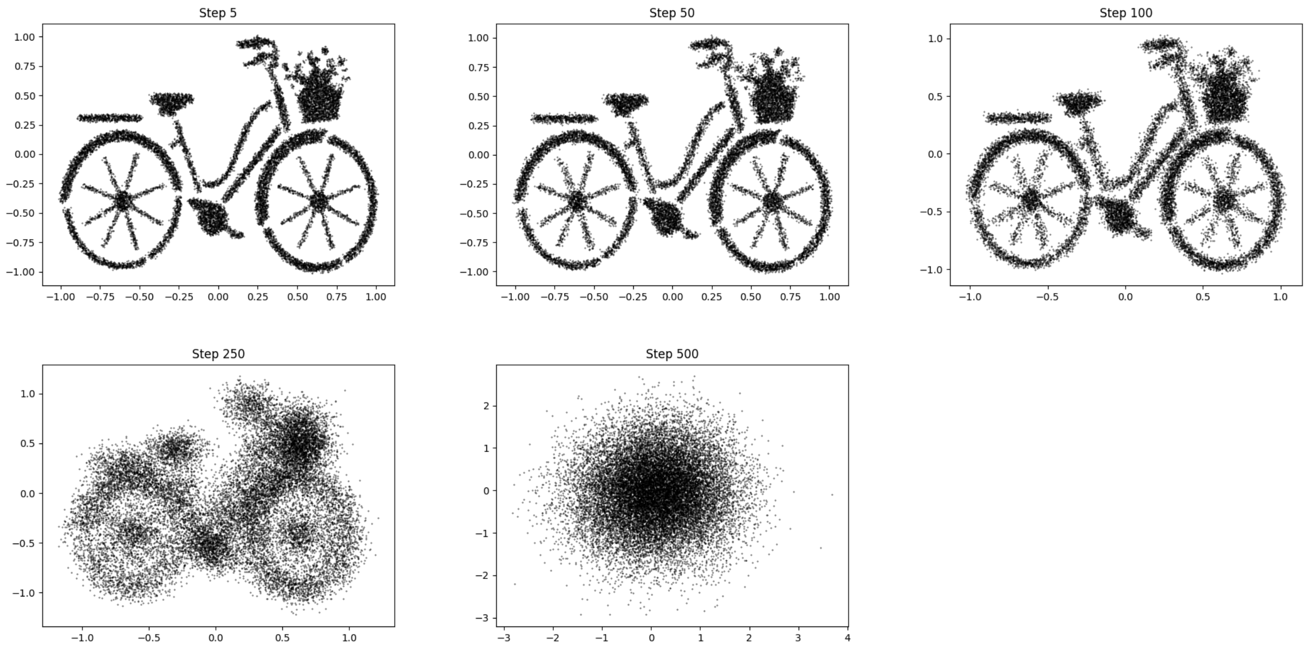
Następnie obliczyłem i narysowałem dwie różne sekwencje wariancji β_t każdego z kroków. W przypadku pierwszej sekwencji β_t zmienia się liniowo of β_i = *1e-4* do β_f = *2e-2* i jest opisane równaniem $\beta_t = \beta_i + t\,(\beta_f - \beta_i) / (T - 1)$ (założyłem, iż kroki czasowe liczone są od 0 do *T-1*), a sekwencja $\bar{a}_t$ jest obliczana jak w instrukcji. W przypadku drugiej sekwencji natomiast β_t jest wyznaczone pośrednio przez zdefiniowanie sekwencji $\bar{a}_t = \sigma(k - 2kt/T)$ i obliczenie $\beta_t = 1 - \bar{a}_t / \bar{a}_{t-1}$, gdzie σ jest standardową logistyczną funkcją sigmoidalną, a parametr k dobrałem ręcznie, aby uzyskana sekwencja miała preferowany kształt. Sekwencje $\bar{a}_t$ dla obu powyższych przypadków zamieściłem na poniższym wykresie.



Następnie zaimplementowałem funkcję realizującą dyfuzję “w przód”, która przyjmuje batch wektorów, krok czasowy t oraz sekwencję β_t i zwraca batch wektorów po t krokach dyfuzji. Poniżej zamieściłem implementację tej funkcji.

```python
from torch.distributions import MultivariateNormal as MVN

def diffuse(x: Tensor, t: int, β: Tensor) -> Tensor:
    assert 0 <= t < β.shape[0]
    ā = torch.cumprod(1 - β, dim=0)
    return MVN(torch.sqrt(ā[t]) * x, (1 - ā[t]) * torch.eye(2)).sample()
```

Poniżej zamieściłem efekt dyfuzji zbioru danych po 5, 50, 100, 250, i 500 krokach.



Następnie zaimplementowałem model predykcyjny zgodnie z architekturą podaną w instrukcji.

```python
class LearnableSinusoidalEmbedding(nn.Module):

    def __init__(self, output_dim: int = 128, latent_dim: int = 50, max_period: int = 10_000):
        super().__init__()
        self.lin1 = nn.Linear(latent_dim, output_dim)
        self.lin2 = nn.Linear(output_dim, output_dim)
        self.pe = lambda t: self.sinusoidal_positional_encoding(t, latent_dim, max_period)

    def sinusoidal_positional_encoding(self, t: Tensor | int, dim: int, max_period: int):
        assert dim % 2 == 0
        if type(t) is int:
            t = torch.tensor([[t]])
        freqs = 1 / torch.pow(max_period, 2 / dim * torch.arange(0, dim // 2))

        embeddings = torch.zeros(t.shape[0], dim)
        embeddings[:, 0::2] = torch.sin(freqs * t)
        embeddings[:, 1::2] = torch.cos(freqs * t)

        return embeddings

    def forward(self, t: Tensor | int) -> Tensor:
        return self.lin2(F.relu(self.lin1(self.pe(t))))
```

```
class DDPM(nn.Module):

    def __init__(self, dim: int = 2, latent_dim: int = 128):
        super().__init__()
        self.dim = dim
        self.emb = LearnableSinusoidalEmbedding(output_dim=latent_dim)
        self.lin1 = nn.Linear(dim, latent_dim)
        self.lin2 = nn.Linear(latent_dim, latent_dim)
        self.lin3 = nn.Linear(latent_dim, latent_dim)
        self.lin4 = nn.Linear(latent_dim, dim)

    def forward(self, x: Tensor, t: Tensor | int) -> Tensor:
        t = self.emb(t)
        x = F.relu(t + self.lin1(x))
        x = F.relu(t + self.lin2(x))
        x = F.relu(t + self.lin3(x))
        x = self.lin4(x)
        return x

    @torch.no_grad()
    def sample(self, size: int, T: int, β: Tensor, x0: Tensor | None = None) -> Tensor:
        assert x0 is None or x0.shape == (size, self.dim)
        assert β.shape[0] >= T

        self.eval()

        mvn = MVN(loc=torch.zeros(self.dim), covariance_matrix=torch.eye(self.dim))
        a = 1 - β
        ā = torch.cumprod(a, dim=0)
        x = mvn.sample((size,)) if x0 is None else x0

        for t in trange(T - 1, -1, -1):
            z = mvn.sample((size,)) if t > 0 else torch.zeros((size, self.dim))
            ϵ_θ = self(x, t)
            n = torch.sqrt(β[t]) * z
            x = 1 / torch.sqrt(a[t]) * (x - (1 - a[t])/ torch.sqrt(1 - ā[t]) * ϵ_θ) + n

        return x
```

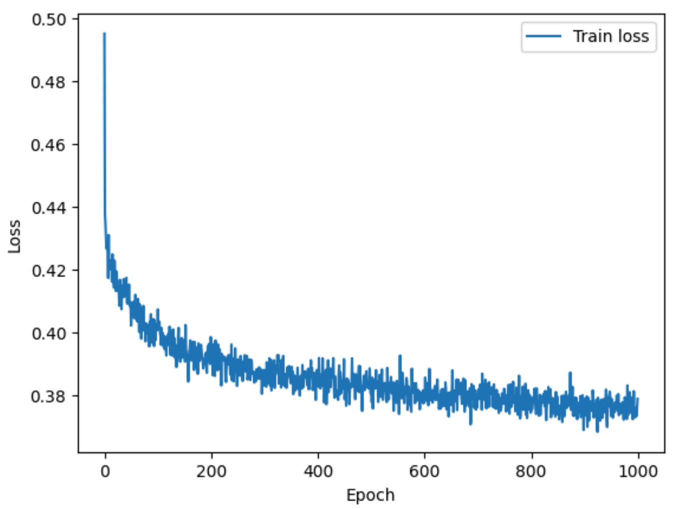
Do właściwego modelu DDPM dodałem również funkcję implementującą algorytm generacji. Po zdefiniowaniu liczby epok, optymalizatora z odpowiednią stałą uczącą, wielkości batcha oraz funkcji straty

```
epochs = 1000
log_period = 10
batch_size = 64

ddpm = DDPM(latent_dim=128)
criterion = F.mse_loss
optimizer = optim.Adam(ddpm.parameters(), lr=1e-4)
```

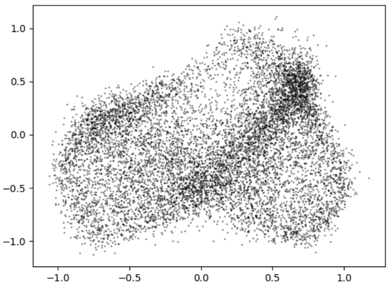
i przetestowaniu, iż model zwraca odpowiednie dane, następnym krokiem była implementacja pętli treningowej zgodnie z algorytmem DDPM.

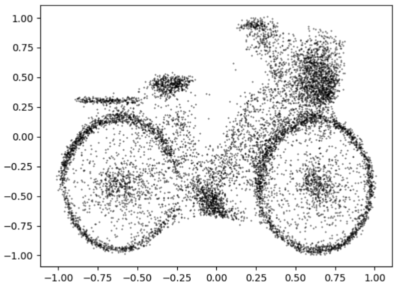
```
loss_hist = []
mvn = MVN(loc=torch.zeros(2), covariance_matrix=torch.eye(2))
sample_size = 10_000
x0 = mvn.sample((sample_size,))

for epoch in (pbar := trange(epochs)):
    ddpm.train()
    running_loss = 0.0

    for x_batch in make_batch(X, batch_size):
        optimizer.zero_grad()

        t = torch.tensor(np.random.choice(T, size=(batch_size, 1)))
        ϵ = mvn.sample((batch_size,))
        ϵ_θ = ddpm(torch.sqrt(ā[t]) * x_batch + torch.sqrt(1 - ā[t]) * ϵ, t)

        loss = criterion(ϵ_θ, ϵ)
        loss.backward()

        optimizer.step()

        running_loss += loss.item()

    epoch_loss = (running_loss * batch_size) / len(X)
    loss_hist.append(epoch_loss)
    pbar.set_description(f"Epoch [{epoch+1}/{epochs}] | Loss: {epoch_loss:.4f}")

    if epoch == 0 or (epoch + 1) % log_period == 0:
        show(ddpm.sample(sample_size, T, β, x0), save_path=f"./results/r{epoch+1}.png")
        torch.save(
        # ...
        )
```

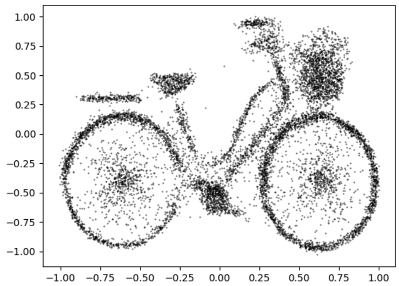
W pętli treningowej zaimplementowałem również okresowe zapisywanie stanu modelu oraz efektu procedury generacji dla ustalonego przed pętlą początkową batcha wektorów wylosowanych z rozkładu normalnego wykorzystywanych w procesie odszumiania jako początkowe wartości. Poniżej zamieściłem uzyskany wykres wartości funkcji straty w kolejnych epokach.



Poniżej zamieściłem otrzymane efekty procedury generacji dla modelu po 50, 500 i 1000 epokach treningu. Do rozwiązania dołączyłem również plik .gif przedstawiający rozkład punktów dla kolejnych kroków czasowych procesu generacji dla wytrenowanego modelu, który zbiega do rozkładu przedstawionego na trzecim z poniższych wykresów.







Na podstawie wykresu wartości funkcji straty oraz powyższych rezultatów generacji dla modelu po różnych krokach czasowych można stwierdzić, iż spadek jakości generowanych rozkładów jest z pewnością związany z różnicami w funkcji loss, chociaż nie jest to raczej zależność wprost proporcjonalna. Pomimo, iż ciężko jest dokładnie ilościowo opisać różnice w jakości generowanych rozkładów, to można zauważyć, iż wbrew minimalnej różnicy między wartościami funkcji straty po 500 i 1000 epok, rozkład generowany przez model po 1000 epok jest znacznie bardziej szczegółowy.