# diffusion_on_synthetic_manifolds

November 23, 2025

## 1 Diffusion on Synthetic Manifolds

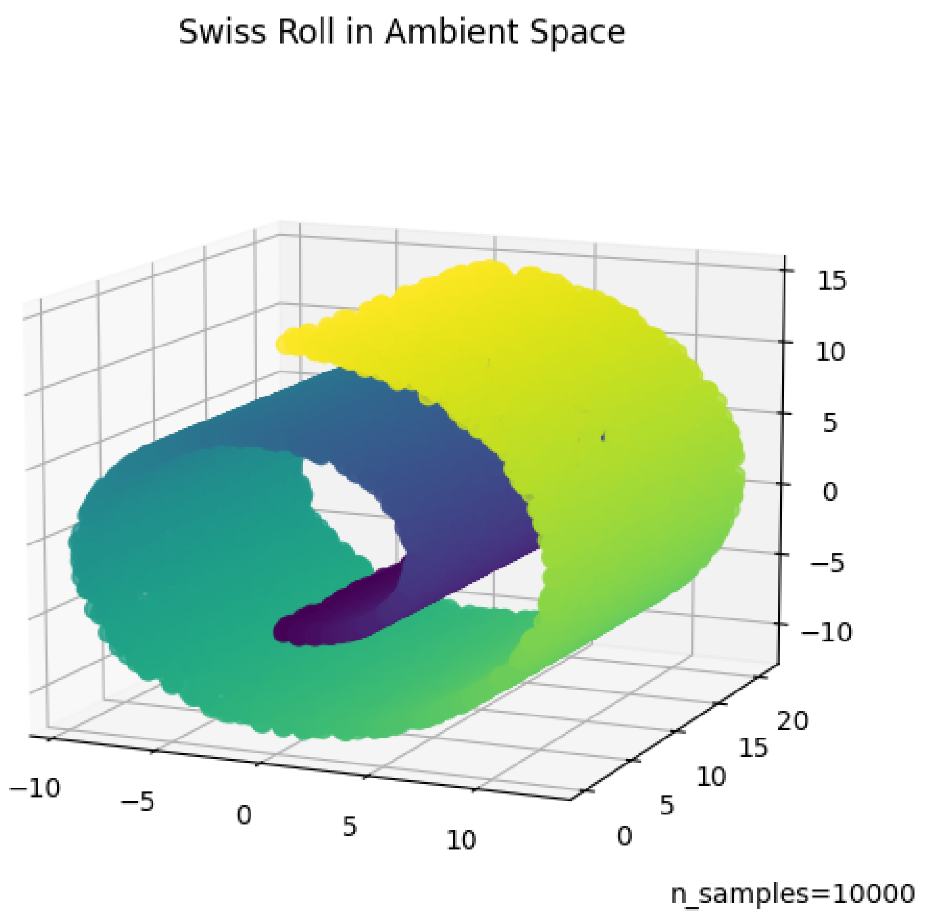
In this exploration, I will be creating a swiss roll data sampler and then will run diffusion on that synthetic data. The idea here is that diffusion is learning the manifold implicitly and what extent can we show this. If this is the case, can we use a diffusion model in a real setting to create synthetic data to better learn a manifold. Furthermore, is there anyway you could use a learned manifold to try and help the sample complexity of the diffusion model?

### 1.1 Is Diffusion implicitly learning the manifold of the data?

#### 1.1.1 ChatGPT created example

First let us create a sampler that will sample data from some swiss roll object, $x \in \mathbb{R}^d$ living on same $k$-dimensional swiss roll ($k < d$).

[9]:
```
from sklearn.datasets import make_swiss_roll
import matplotlib.pyplot as plt
import random
import numpy as np
import torch
# This class will abstract out all of the data setup. You will be able to just␣
↪create a synthetic dataset and sample from it
class Sampler():
    # dataset attribute
    X = None
    color = None

    # at init load in swiss roll dataset as default
    def __init__(self):
        self.create_swiss_roll_dataset(10000)

    # Create a swiss roll dataset in $R^3$
    def create_swiss_roll_dataset(self, n, random_state=None, noise=0):
        X, color = make_swiss_roll(n_samples=n, noise=noise,␣
↪random_state=random_state)
        self.X = X
        self.color = color
    # Sample a random datapoint in the dataset
    def sample(self):
        rand_int = random.randint(0, len(self.X))
        return self.X[rand_int]

    def visualize_swissroll(self):
        # this code is taken directly from the scikit learn documentation:␣
↪https://scikit-learn.org/stable/auto_examples/manifold/plot_swissroll.
↪html#sphx-glr-auto-examples-manifold-plot-swissroll-py
        fig = plt.figure(figsize=(8, 6))
        ax = fig.add_subplot(111, projection="3d")
        fig.add_axes(ax)
        ax.scatter(
            self.X[:, 0], self.X[:, 1], self.X[:, 2], c=self.color, s=50,␣
↪alpha=0.8
        )
        ax.set_title("Swiss Roll in Ambient Space")
        ax.view_init(azim=-66, elev=12)
        _ = ax.text2D(0.8, 0.05, s=f"n_samples={len(self.X)}", transform=ax.
↪transAxes)

sampler = Sampler()
sampler.visualize_swissroll()
```



Next is to define a noise scheduler. For simplicity I will define a linear schedule with $\sigma \in [0.01, 100]$

[ ]:
```
import torch
import numpy as np
import random

class noise_scheduler:
    """Simple linear noise scheduler for diffusion-style experiments.

    Given a timestep t, this class returns a Gaussian noise level (standard
    deviation) from a linear schedule and can directly add noise to a data␣
 ↪point.
    """

    def __init__(self, num_steps: int = 10000,
                 min_noise: float = 0.01,
                 max_noise: float = 100,
                 device: str = "cpu"):
        """Create a linear noise schedule.

        Args:
            num_steps: How many timesteps in the schedule.
            min_noise: Smallest noise level (at t = 0).
            max_noise: Largest noise level (at t = num_steps - 1).
            device: Torch device ("cpu" or "cuda").
        """
        self.num_steps = num_steps
        self.device = device

        # Linearly spaced noise levels from min_noise to max_noise.
        # This will be our "schedule" of standard deviations.
        self.schedule = torch.linspace(min_noise, max_noise,
                                       steps=num_steps,
                                       device=device)

    def get_sigma(self, t: int) -> torch.Tensor:
        """Return the noise standard deviation for timestep t.

        Args:
            t: Integer timestep index in [0, num_steps - 1].
        """
        # Clamp t just in case it goes slightly out of range.
        t = max(0, min(int(t), self.num_steps - 1))
        return self.schedule[t]

    def gaussian_noise(self, x, t: int = None):
        """Add Gaussian noise to a single data point.

        Args:
            x: Data point. Can be a NumPy array or a torch tensor of shape
               (..., d). For your swiss roll this will be shape (3,) or (batch,␣
↪3).
            t: Optional timestep index. If None, a random t is sampled
               uniformly from [0, num_steps - 1].

        Returns:
            x_noisy: Noised version of x.
            noise: The actual Gaussian noise that was added.
            sigma: The noise standard deviation used.
            t: The timestep index that was used.
        """
        # If x is a NumPy array, convert it to a torch tensor.
        if not torch.is_tensor(x):
            x = torch.from_numpy(np.asarray(x)).float()
        # If no timestep is provided, pick one at random.
        if t is None:
            t = random.randint(0, self.num_steps - 1)

        # Get noise scale for this timestep.
        sigma = self.get_sigma(t)

        # Sample standard normal noise with the same shape as x,
        # then scale it by sigma.
        noise = torch.randn_like(x) * sigma

        # Add noise to the original data.
        x_noisy = x + noise

        return x_noisy, noise, sigma
```

Next is to create the model to predit the noise

[11]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F

class SimpleNoisePredictor(nn.Module):
    """
    Very simple MLP to predict Gaussian noise   from a noised point x_t and␣
 ↪timestep t.

    - Input:  x_t (noisy data), shape (3,) or (batch_size, 3)
              t   (timestep),    scalar int or tensor of shape (batch_size,)
    - Output: predicted noise _hat, same shape as x_t
    """

    def __init__(self, num_steps: int, input_dim: int = 3, hidden_dim: int =␣
 ↪128):
        super().__init__()

        # We need num_steps so we can normalize t to [0, 1]
        self.num_steps = num_steps

        # Input to the network is x_t (dim=3) plus time scalar (dim=1) -> 4 dims
        self.fc1 = nn.Linear(input_dim + 1, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, hidden_dim)
        self.fc3 = nn.Linear(hidden_dim, input_dim)  # output: predicted noise␣
 ↪(same dim as x)

        # Non-linearity
        self.act = nn.SiLU()  # you can also use nn.ReLU() if you want it even␣
↪simpler

    def forward(self, x: torch.Tensor, t):
        """
        Args:
            x: Noisy input point(s). Shape (3,) or (batch_size, 3).
            t: Timestep(s). Can be:
               - Python int
               - 0-D tensor
               - 1-D tensor of shape (batch_size,)
               - 2-D tensor of shape (batch_size, 1)

        Returns:
            eps_pred: Predicted noise with same shape as x.
        """

        # Ensure x has a batch dimension: (3,) -> (1, 3)
        if x.dim() == 1:
            x = x.unsqueeze(0)  # (1, 3)

        # Move x to some device
        device = x.device

        # Convert t to a tensor if it's not one already
        if not torch.is_tensor(t):
            t = torch.tensor(t, device=device)

        t = t.to(device).float()

        # Make t shape (batch_size, 1)
        if t.dim() == 0:
            # same t for the whole batch
            t = t.repeat(x.size(0))  # (batch_size,)
        if t.dim() == 1:
            t = t.unsqueeze(-1)  # (batch_size, 1)

        # Normalize t to [0, 1] for stability:
        # if num_steps = 10000, t_norm = t / 9999
        t_norm = t / (self.num_steps - 1)

        # Concatenate x_t and t along the feature dimension
        # x:      (batch_size, 3)
        # t_norm: (batch_size, 1)
        # -> x_input: (batch_size, 4)
        x_input = torch.cat([x, t_norm], dim=-1)
        # Simple 3-layer MLP
        h = self.act(self.fc1(x_input))
        h = self.act(self.fc2(h))
        eps_pred = self.fc3(h)  # (batch_size, 3)

        # If the original input was a single point (3,), return (3,) as well
        if eps_pred.size(0) == 1:
            return eps_pred.squeeze(0)

        return eps_pred
```

[12]:
```
import numpy as np
import torch
import torch.nn.functional as F
import matplotlib.pyplot as plt


class SwissRollDDPM:
    """
    Simple DDPM pipeline for a 3D swiss roll dataset.

    - Uses a beta schedule (linear) as in the original DDPM paper.
    - Trains a noise-prediction model  _ (x_t, t).
    - Samples new points by running the reverse diffusion process.
    """

    def __init__(
        self,
        sampler,
        num_steps: int = 1000,
        beta_start: float = 1e-4,
        beta_end: float = 0.02,
        hidden_dim: int = 128,
        lr: float = 1e-3,
        device: str | None = None,
    ):
        """
        Args:
            sampler: Your Sampler() instance that holds the swiss roll data.
            num_steps: Number of diffusion timesteps T.
            beta_start, beta_end: Linear beta schedule range.
            hidden_dim: Hidden dimension for SimpleNoisePredictor.
            lr: Learning rate for Adam.
            device: "cpu" or "cuda". If None, auto-detect.
        """
        self.sampler = sampler
        self.T = num_steps
        if device is None:
            device = "cuda" if torch.cuda.is_available() else "cpu"
        self.device = device

        # --------------------------
        # DDPM noise schedule setup
        # --------------------------
        # Linear betas from beta_start to beta_end
        self.betas = torch.linspace(beta_start, beta_end, self.T, device=self.
↪device)  # (T,)
        self.alphas = 1.0 - self.betas                                          ␣
↪         # (T,)
        self.alpha_bars = torch.cumprod(self.alphas, dim=0)                     ␣
↪         # (T,)

        # --------------------------
        # Model & optimizer
        # --------------------------
        # Simple MLP that predicts   given x_t and t
        self.model = SimpleNoisePredictor(
            num_steps=self.T, input_dim=3, hidden_dim=hidden_dim
        ).to(self.device)

        self.optimizer = torch.optim.Adam(self.model.parameters(), lr=lr)

    # --------------------------
    # Helper: sample a clean batch x_0 from swiss roll
    # --------------------------
    def _sample_clean_batch(self, batch_size: int) -> torch.Tensor:
        """
        Sample a batch of clean points x_0 from the swiss roll.

        Uses sampler.X directly for speed and to avoid Python loops.
        Returns: x0 of shape (batch_size, 3) as a float32 tensor on self.device.
        """
        # sampler.X is a NumPy array of shape (N, 3)
        N = self.sampler.X.shape[0]
        idx = np.random.randint(0, N, size=batch_size)
        x0_np = self.sampler.X[idx]  # (batch_size, 3)
        x0 = torch.from_numpy(x0_np).float().to(self.device)
        return x0

    # --------------------------
    # Forward diffusion: q(x_t | x_0)
    # --------------------------
    def q_sample(self, x0: torch.Tensor, t: torch.Tensor, noise: torch.Tensor |␣
↪None = None):
        """
        Sample from q(x_t | x_0) using the closed-form DDPM formula:

            x_t = sqrt(alpha_bar_t) * x_0 + sqrt(1 - alpha_bar_t) *  ,

        where   ~ N(0, I).

        Args:
            x0: Clean data, shape (batch_size, 3).
            t:  Timesteps, shape (batch_size,) integer tensor in [0, T-1].
            noise: Optional pre-sampled noise (same shape as x0).

        Returns:
            x_t, noise
        """
        if noise is None:
            noise = torch.randn_like(x0)

        # alpha_bar_t: shape (batch_size,)
        alpha_bar_t = self.alpha_bars[t]  # index per example
        alpha_bar_t = alpha_bar_t.unsqueeze(-1)  # (batch_size, 1) for␣
↪broadcasting

        x_t = torch.sqrt(alpha_bar_t) * x0 + torch.sqrt(1.0 - alpha_bar_t) *␣
↪noise
        return x_t, noise

    # ---------------------------
    # Single training step
    # ---------------------------
    def training_step(self, batch_size: int) -> float:
        """
        Perform one gradient step on a random batch and random timesteps.

        Loss is E[ || _ (x_t, t) -  ||^2 ] as in DDPM.
        """
        self.model.train()

        # 1. Sample clean data x_0
        x0 = self._sample_clean_batch(batch_size)  # (B, 3)

        # 2. Sample random timesteps t ~ Uniform({0,...,T-1})
        t = torch.randint(0, self.T, (batch_size,), device=self.device).long()

        # 3. Sample noise   ~ N(0, I) and get x_t
        noise = torch.randn_like(x0)
        x_t, _ = self.q_sample(x0, t, noise=noise)

        # 4. Predict noise with the model
        eps_pred = self.model(x_t, t)  # (B, 3)

        # 5. Compute MSE loss between true and predicted noise
        loss = F.mse_loss(eps_pred, noise)

        # 6. Backprop and optimizer step
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        return loss.item()

    # ---------------------------
    # Full training loop
    # ---------------------------
    def train(self, num_steps: int = 10_000, batch_size: int = 128, log_every:␣
↪int = 500):
        """
        Train the diffusion model.

        Args:
            num_steps: Number of gradient updates.
            batch_size: Batch size.
            log_every: Print loss every this many steps.
        """
        for step in range(1, num_steps + 1):
            loss = self.training_step(batch_size)

            if step % log_every == 0 or step == 1:
                print(f"[step {step}/{num_steps}] loss = {loss:.6f}")

    # ---------------------------
    # Reverse process: p_ (x_{t-1} | x_t)
    # ---------------------------
    def p_sample(self, x_t: torch.Tensor, t: int) -> torch.Tensor:
        """
        Sample x_{t-1} given x_t using the DDPM update:

          _ (x_t, t) = 1/sqrt( _t) * ( x_t - _t / sqrt(1 - ¯_t) * _ (x_t, t) )
          x_{t-1} ~ N( _ (x_t, t), _t I )

        For t = 0, we just return the mean (no noise term).
        """
        # Create a batch of timesteps filled with the scalar t
        batch_size = x_t.shape[0]
        t_batch = torch.full((batch_size,), t, device=self.device, dtype=torch.
↪long)

        # Predict noise  _ (x_t, t)
        eps_theta = self.model(x_t, t_batch)  # (B, 3)

        beta_t = self.betas[t]         # scalar tensor
        alpha_t = self.alphas[t]       # scalar tensor
        alpha_bar_t = self.alpha_bars[t]  # scalar tensor

        # Compute the mean of p_ (x_{t-1} | x_t)
        #  _ (x_t, t) = 1/sqrt( _t) * ( x_t -  _t / sqrt(1 - ¯_t) *  _ (x_t, t) )
        coef1 = 1.0 / torch.sqrt(alpha_t)
        coef2 = beta_t / torch.sqrt(1.0 - alpha_bar_t)

        mean = coef1 * (x_t - coef2 * eps_theta)  # (B, 3)

        if t == 0:
            # No noise at the final step
            return mean

        # Add Gaussian noise with variance  _t
        noise = torch.randn_like(x_t)
        sigma_t = torch.sqrt(beta_t)
        x_prev = mean + sigma_t * noise
        return x_prev

    # --------------------------
    # Generate new samples x_0
    # --------------------------
    def sample(self, num_samples: int = 1000) -> np.ndarray:
        """
        Run the full reverse diffusion chain starting from x_T ~ N(0, I).

        Returns:
            samples: NumPy array of shape (num_samples, 3) on CPU.
        """
        self.model.eval()

        # Start from pure Gaussian noise at timestep T-1
        x_t = torch.randn(num_samples, 3, device=self.device)

        # Iterate backwards from T-1 to 0
        for t in reversed(range(self.T)):
            x_t = self.p_sample(x_t, t)
        # x_t is now an approximation of x_0
        x_0 = x_t.detach().cpu().numpy()
        return x_0

    # --------------------------
    # Visualization helper
    # --------------------------
    def visualize_generated(self, num_samples: int = 2000):
        """
        Generate samples and visualize them as a 3D scatter plot.
        """
        samples = self.sample(num_samples=num_samples)  # (N, 3)

        fig = plt.figure(figsize=(8, 6))
        ax = fig.add_subplot(111, projection="3d")
        fig.add_axes(ax)

        ax.scatter(
            samples[:, 0],
            samples[:, 1],
            samples[:, 2],
            s=5,
            alpha=0.8,
        )

        ax.set_title("Generated Samples from DDPM (Swiss Roll)")
        ax.view_init(azim=-66, elev=12)
        plt.show()
```

[14]:
```
# 1. Create the swiss roll data
sampler = Sampler()  # from your code

# 2. Set up the diffusion model
ddpm = SwissRollDDPM(
    sampler=sampler,
    num_steps=1000,
    beta_start=1e-4,
    beta_end=0.02,
    hidden_dim=128,
    lr=1e-3,
)

# 3. Train
ddpm.train(num_steps=100_000, batch_size=128, log_every=500)

# 4. Visualize generated samples
```

```python
ddpm.visualize_generated(num_samples=2000)
```

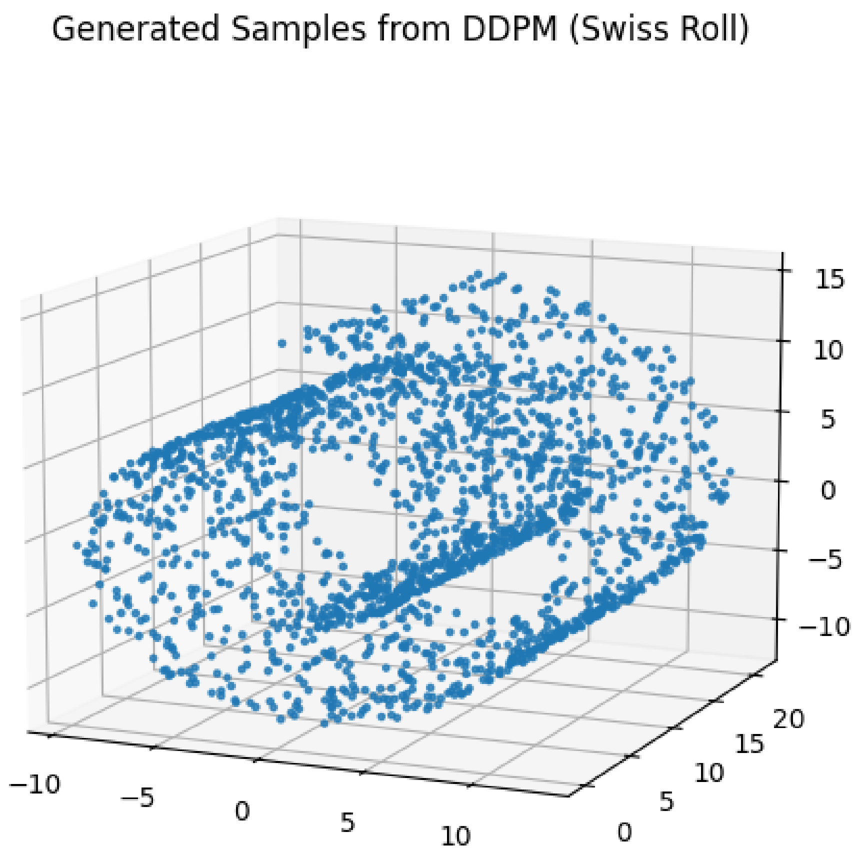
```
[step 1/100000] loss = 1.103717
[step 500/100000] loss = 0.765187
[step 1000/100000] loss = 0.533318
[step 1500/100000] loss = 0.626332
[step 2000/100000] loss = 0.554502
[step 2500/100000] loss = 0.577751
[step 3000/100000] loss = 0.592428
[step 3500/100000] loss = 0.567604
[step 4000/100000] loss = 0.551858
[step 4500/100000] loss = 0.555573
[step 5000/100000] loss = 0.508502
[step 5500/100000] loss = 0.613802
[step 6000/100000] loss = 0.601033
[step 6500/100000] loss = 0.475115
[step 7000/100000] loss = 0.561418
[step 7500/100000] loss = 0.671865
[step 8000/100000] loss = 0.421422
[step 8500/100000] loss = 0.610787
[step 9000/100000] loss = 0.702435
[step 9500/100000] loss = 0.517592
[step 10000/100000] loss = 0.590120
[step 10500/100000] loss = 0.600776
[step 11000/100000] loss = 0.605972
[step 11500/100000] loss = 0.437358
[step 12000/100000] loss = 0.505678
[step 12500/100000] loss = 0.454397
[step 13000/100000] loss = 0.542276
[step 13500/100000] loss = 0.522605
[step 14000/100000] loss = 0.548452
[step 14500/100000] loss = 0.470266
[step 15000/100000] loss = 0.480760
[step 15500/100000] loss = 0.591755
[step 16000/100000] loss = 0.535315
[step 16500/100000] loss = 0.470803
[step 17000/100000] loss = 0.512001
[step 17500/100000] loss = 0.508152
[step 18000/100000] loss = 0.526410
[step 18500/100000] loss = 0.482122
[step 19000/100000] loss = 0.578846
[step 19500/100000] loss = 0.431123
[step 20000/100000] loss = 0.513388
[step 20500/100000] loss = 0.641085
[step 21000/100000] loss = 0.548625
[step 21500/100000] loss = 0.582353
[step 22000/100000] loss = 0.462398
[step 22500/100000] loss = 0.434682
```

```
[step 23000/100000] loss = 0.477718
[step 23500/100000] loss = 0.439215
[step 24000/100000] loss = 0.452749
[step 24500/100000] loss = 0.532735
[step 25000/100000] loss = 0.476838
[step 25500/100000] loss = 0.418592
[step 26000/100000] loss = 0.477092
[step 26500/100000] loss = 0.519844
[step 27000/100000] loss = 0.426415
[step 27500/100000] loss = 0.575072
[step 28000/100000] loss = 0.407333
[step 28500/100000] loss = 0.517289
[step 29000/100000] loss = 0.438208
[step 29500/100000] loss = 0.442350
[step 30000/100000] loss = 0.510799
[step 30500/100000] loss = 0.402384
[step 31000/100000] loss = 0.613533
[step 31500/100000] loss = 0.575857
[step 32000/100000] loss = 0.463132
[step 32500/100000] loss = 0.470280
[step 33000/100000] loss = 0.492426
[step 33500/100000] loss = 0.484607
[step 34000/100000] loss = 0.499704
[step 34500/100000] loss = 0.540262
[step 35000/100000] loss = 0.415227
[step 35500/100000] loss = 0.471062
[step 36000/100000] loss = 0.468129
[step 36500/100000] loss = 0.472248
[step 37000/100000] loss = 0.403704
[step 37500/100000] loss = 0.480412
[step 38000/100000] loss = 0.450692
[step 38500/100000] loss = 0.423511
[step 39000/100000] loss = 0.498410
[step 39500/100000] loss = 0.454692
[step 40000/100000] loss = 0.473609
[step 40500/100000] loss = 0.459194
[step 41000/100000] loss = 0.490081
[step 41500/100000] loss = 0.462625
[step 42000/100000] loss = 0.507399
[step 42500/100000] loss = 0.418573
[step 43000/100000] loss = 0.517959
[step 43500/100000] loss = 0.581990
[step 44000/100000] loss = 0.372811
[step 44500/100000] loss = 0.439116
[step 45000/100000] loss = 0.463585
[step 45500/100000] loss = 0.426864
[step 46000/100000] loss = 0.618116
[step 46500/100000] loss = 0.449340
```

```
[step 47000/100000] loss = 0.553180
[step 47500/100000] loss = 0.521653
[step 48000/100000] loss = 0.512340
[step 48500/100000] loss = 0.442329
[step 49000/100000] loss = 0.512836
[step 49500/100000] loss = 0.525813
[step 50000/100000] loss = 0.483353
[step 50500/100000] loss = 0.511071
[step 51000/100000] loss = 0.489108
[step 51500/100000] loss = 0.482543
[step 52000/100000] loss = 0.465241
[step 52500/100000] loss = 0.463083
[step 53000/100000] loss = 0.418070
[step 53500/100000] loss = 0.436991
[step 54000/100000] loss = 0.448345
[step 54500/100000] loss = 0.490275
[step 55000/100000] loss = 0.503422
[step 55500/100000] loss = 0.441451
[step 56000/100000] loss = 0.478661
[step 56500/100000] loss = 0.506621
[step 57000/100000] loss = 0.509634
[step 57500/100000] loss = 0.456413
[step 58000/100000] loss = 0.439071
[step 58500/100000] loss = 0.479341
[step 59000/100000] loss = 0.397926
[step 59500/100000] loss = 0.430890
[step 60000/100000] loss = 0.429841
[step 60500/100000] loss = 0.482890
[step 61000/100000] loss = 0.473771
[step 61500/100000] loss = 0.428949
[step 62000/100000] loss = 0.371051
[step 62500/100000] loss = 0.514289
[step 63000/100000] loss = 0.439919
[step 63500/100000] loss = 0.497655
[step 64000/100000] loss = 0.457001
[step 64500/100000] loss = 0.372673
[step 65000/100000] loss = 0.446049
[step 65500/100000] loss = 0.372027
[step 66000/100000] loss = 0.453831
[step 66500/100000] loss = 0.498761
[step 67000/100000] loss = 0.430270
[step 67500/100000] loss = 0.512454
[step 68000/100000] loss = 0.485791
[step 68500/100000] loss = 0.356401
[step 69000/100000] loss = 0.458892
[step 69500/100000] loss = 0.495172
[step 70000/100000] loss = 0.494406
[step 70500/100000] loss = 0.446237
```

```
[step 71000/100000] loss = 0.416387
[step 71500/100000] loss = 0.488306
[step 72000/100000] loss = 0.345560
[step 72500/100000] loss = 0.482300
[step 73000/100000] loss = 0.566690
[step 73500/100000] loss = 0.436884
[step 74000/100000] loss = 0.470106
[step 74500/100000] loss = 0.497590
[step 75000/100000] loss = 0.558484
[step 75500/100000] loss = 0.446212
[step 76000/100000] loss = 0.537137
[step 76500/100000] loss = 0.485704
[step 77000/100000] loss = 0.576250
[step 77500/100000] loss = 0.418841
[step 78000/100000] loss = 0.387424
[step 78500/100000] loss = 0.522504
[step 79000/100000] loss = 0.550985
[step 79500/100000] loss = 0.494085
[step 80000/100000] loss = 0.483919
[step 80500/100000] loss = 0.490667
[step 81000/100000] loss = 0.409289
[step 81500/100000] loss = 0.411036
[step 82000/100000] loss = 0.461117
[step 82500/100000] loss = 0.447544
[step 83000/100000] loss = 0.451863
[step 83500/100000] loss = 0.449378
[step 84000/100000] loss = 0.301577
[step 84500/100000] loss = 0.519905
[step 85000/100000] loss = 0.450514
[step 85500/100000] loss = 0.493273
[step 86000/100000] loss = 0.521779
[step 86500/100000] loss = 0.364811
[step 87000/100000] loss = 0.501973
[step 87500/100000] loss = 0.481274
[step 88000/100000] loss = 0.559089
[step 88500/100000] loss = 0.490527
[step 89000/100000] loss = 0.423767
[step 89500/100000] loss = 0.424489
[step 90000/100000] loss = 0.381260
[step 90500/100000] loss = 0.473374
[step 91000/100000] loss = 0.539499
[step 91500/100000] loss = 0.387206
[step 92000/100000] loss = 0.448522
[step 92500/100000] loss = 0.476756
[step 93000/100000] loss = 0.474326
[step 93500/100000] loss = 0.420166
[step 94000/100000] loss = 0.386300
[step 94500/100000] loss = 0.459074
```

```
[step 95000/100000] loss = 0.405718
[step 95500/100000] loss = 0.456781
[step 96000/100000] loss = 0.413887
[step 96500/100000] loss = 0.407953
[step 97000/100000] loss = 0.448682
[step 97500/100000] loss = 0.550710
[step 98000/100000] loss = 0.350137
[step 98500/100000] loss = 0.431670
[step 99000/100000] loss = 0.651962
[step 99500/100000] loss = 0.467459
[step 100000/100000] loss = 0.591693
```

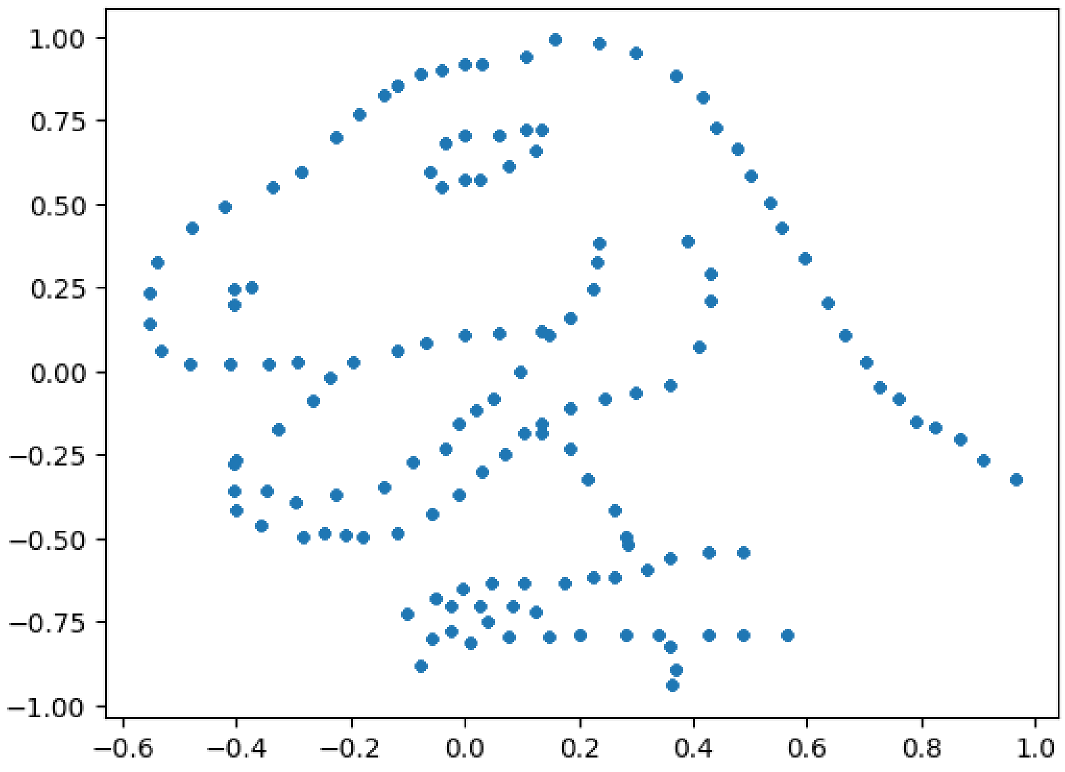


### 1.1.2 Smalldiffusion example

See reference - https://github.com/yuanchenyang/smalldiffusion/blob/main/examples/toyexample.ipynb

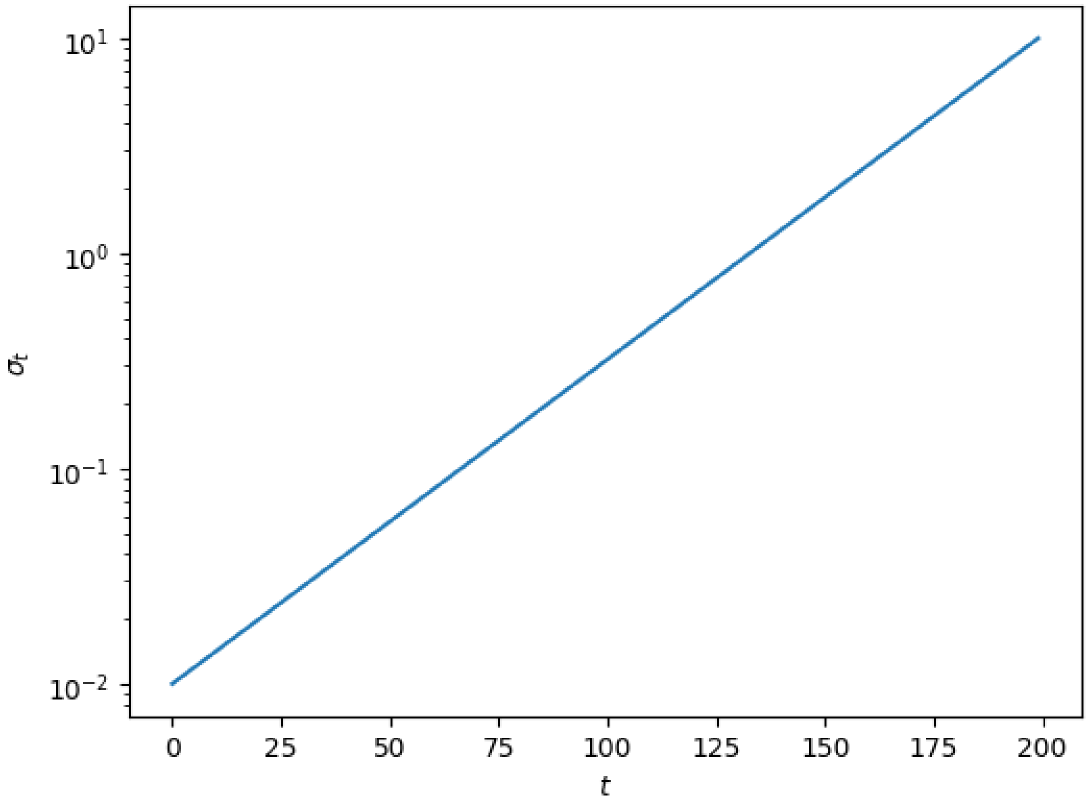
[15]:
```
import numpy as np
import torch
import matplotlib.pyplot as plt
from torch.utils.data import DataLoader
from smalldiffusion import (
    TimeInputMLP, ScheduleLogLinear, training_loop, samples,
    DatasaurusDozen, Swissroll
)

def plot_batch(batch):
    batch = batch.cpu().numpy()
    plt.scatter(batch[:,0], batch[:,1], marker='.')

def moving_average(x, w):
    return np.convolve(x, np.ones(w), 'valid') / w
```

[21]:
```
# Try replacing dataset with 'dino', 'bullseye', 'h_lines', 'x_shape', etc.
dataset = DatasaurusDozen(csv_file='./data/DatasaurusDozen.tsv', dataset='dino')
# Or use the SwissRoll dataset
# dataset = Swissroll(np.pi/2, 5*np.pi, 100)
loader = DataLoader(dataset, batch_size=2130)
plot_batch(next(iter(loader)))
```



[22]:
```
schedule = ScheduleLogLinear(N=200, sigma_min=0.01, sigma_max=10)
plt.plot(schedule.sigmas)
plt.xlabel('$t$')
plt.ylabel('$\sigma_t$')
plt.yscale('log')
```

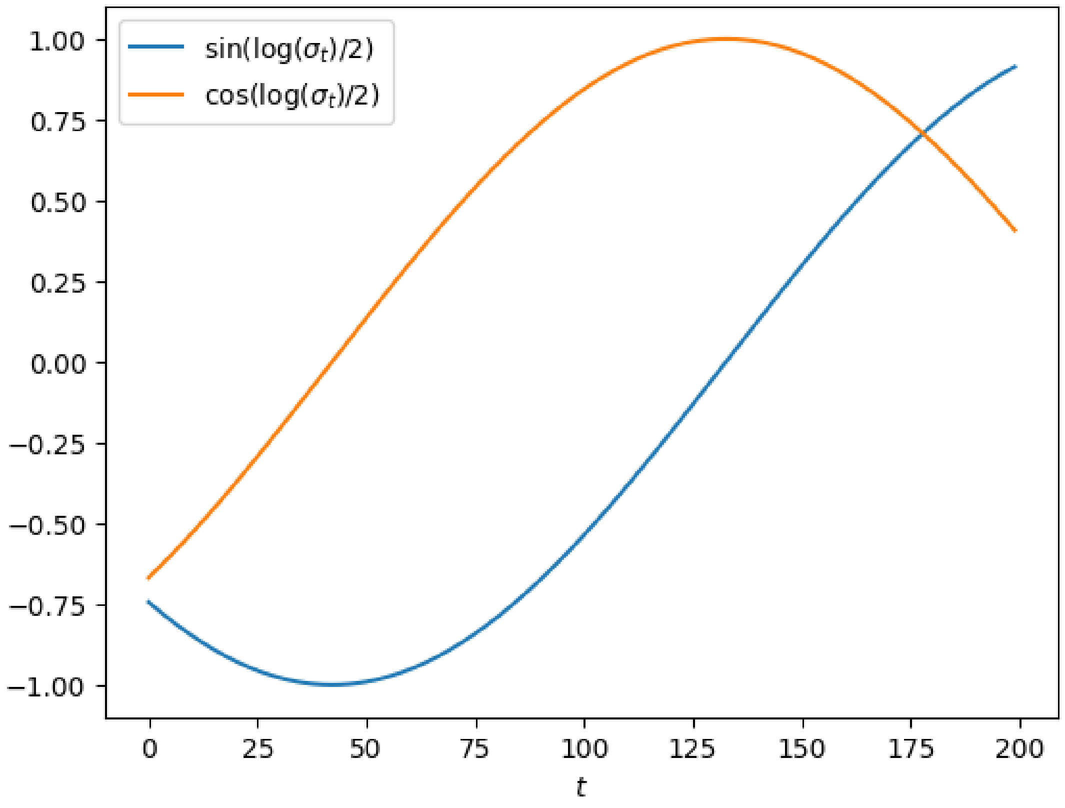
```
<>:4: SyntaxWarning: invalid escape sequence '\s'
<>:4: SyntaxWarning: invalid escape sequence '\s'
/var/folders/rd/mp__2p510kndbjc_f7sc4jz00000gn/T/ipykernel_88359/445830193.py:4:
SyntaxWarning: invalid escape sequence '\s'
  plt.ylabel('$\sigma_t$')
```



[23]:
```
from smalldiffusion.model import get_sigma_embeds
sx, sy = get_sigma_embeds(len(schedule), schedule.sigmas).T
plt.plot(sx, label='$\sin(\log(\sigma_t)/2)$')
plt.plot(sy, label='$\cos(\log(\sigma_t)/2)$')
plt.xlabel('$t$')
plt.legend()
plt.show()
```

```
<>:3: SyntaxWarning: invalid escape sequence '\s'
<>:4: SyntaxWarning: invalid escape sequence '\c'
```

```
<>:3: SyntaxWarning: invalid escape sequence '\s'
<>:4: SyntaxWarning: invalid escape sequence '\c'
/var/folders/rd/mp__2p510kndbjc_f7sc4jz00000gn/T/ipykernel_88359/1963255254.py:3
: SyntaxWarning: invalid escape sequence '\s'
  plt.plot(sx, label='$\sin(\log(\sigma_t)/2)$')
/var/folders/rd/mp__2p510kndbjc_f7sc4jz00000gn/T/ipykernel_88359/1963255254.py:4
: SyntaxWarning: invalid escape sequence '\c'
  plt.plot(sy, label='$\cos(\log(\sigma_t)/2)$')
```



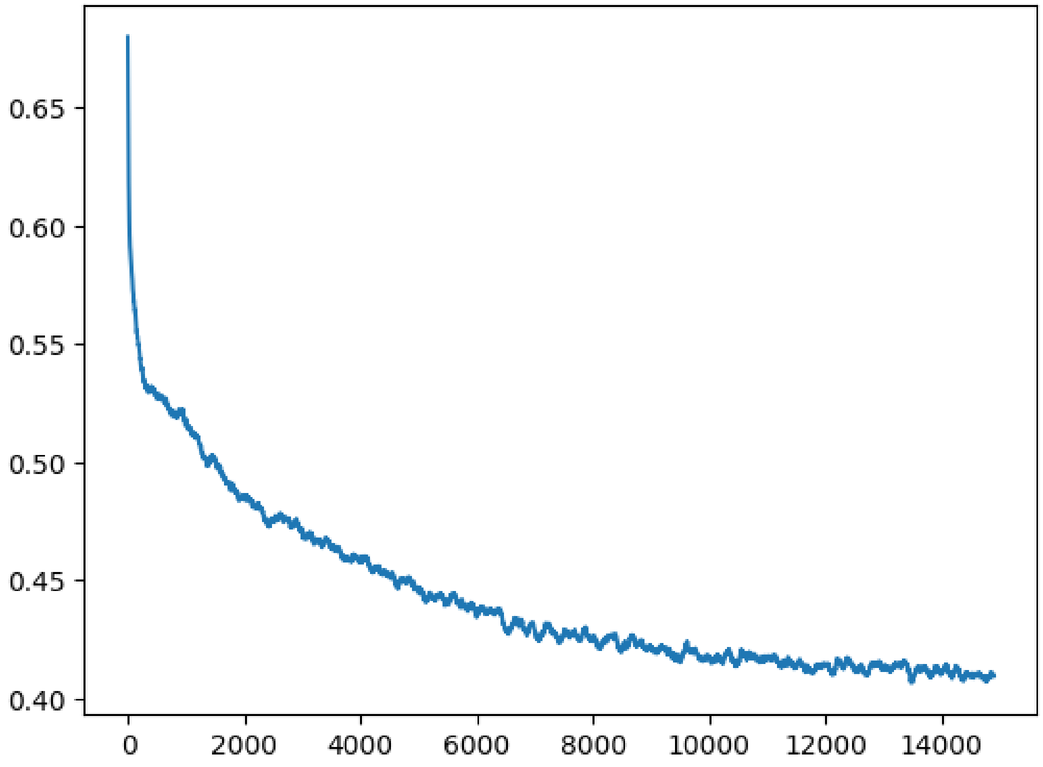
[24]:
```
model = TimeInputMLP(hidden_dims=(16,128,128,128,128,16))
print(model)
```

```
TimeInputMLP(
  (net): Sequential(
    (0): Linear(in_features=4, out_features=16, bias=True)
    (1): GELU(approximate='none')
    (2): Linear(in_features=16, out_features=128, bias=True)
    (3): GELU(approximate='none')
    (4): Linear(in_features=128, out_features=128, bias=True)
    (5): GELU(approximate='none')
    (6): Linear(in_features=128, out_features=128, bias=True)
    (7): GELU(approximate='none')
    (8): Linear(in_features=128, out_features=128, bias=True)
    (9): GELU(approximate='none')
    (10): Linear(in_features=128, out_features=16, bias=True)
    (11): GELU(approximate='none')
    (12): Linear(in_features=16, out_features=2, bias=True)
  )
)
```

[25]:
```
trainer = training_loop(loader, model, schedule, epochs=15000, lr=1e-3)
losses = [ns.loss.item() for ns in trainer]
```

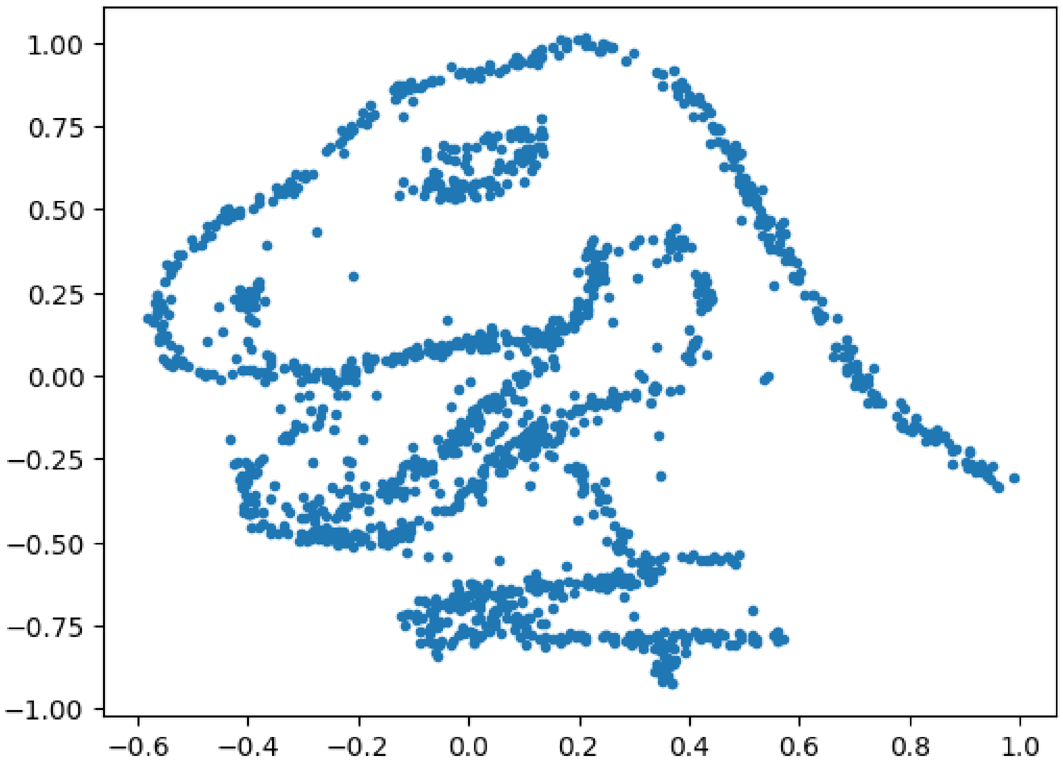
```
100%|      | 15000/15000 [01:07<00:00, 221.61it/s]
```

[26]:
```
plt.plot(moving_average(losses, 100))
plt.show()
```



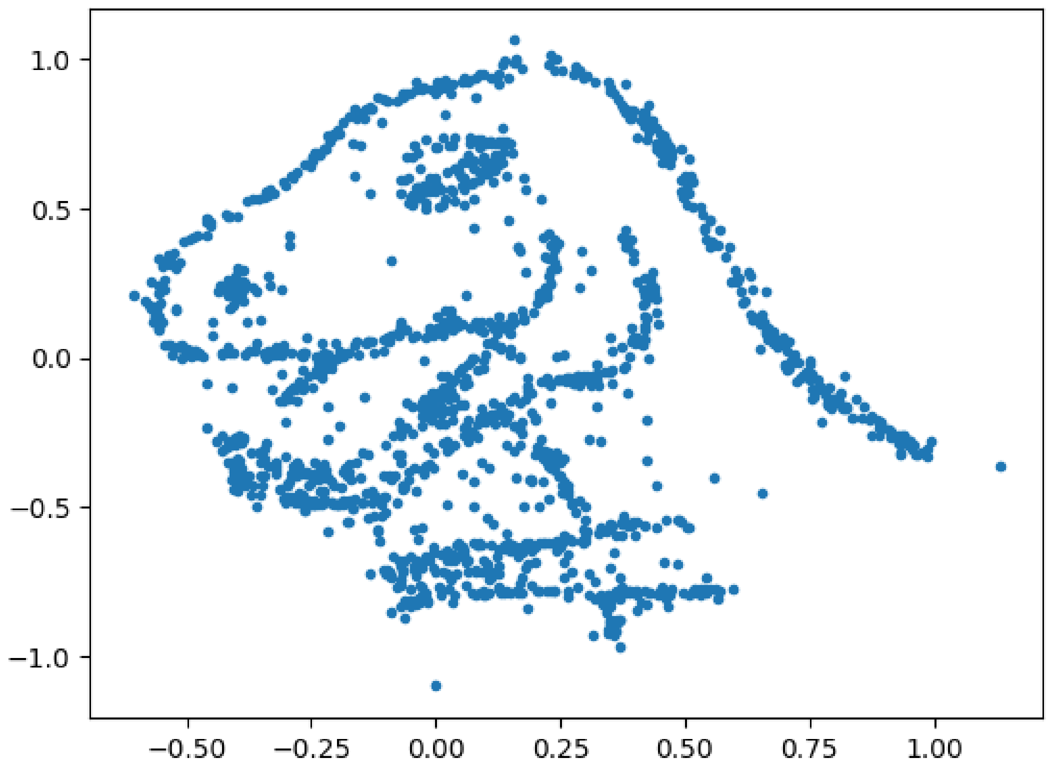
DDPM Sampling

[27]:
```
# For DDPM sampling, change to gam=1, mu=0.5
# For DDIM sampling, change to gam=1, mu=0
*xts, x0 = samples(model, schedule.sample_sigmas(20), batchsize=1500, gam=1,␣
 ↪mu=0.5)
plot_batch(x0)
```



DDIM Sampling

[28]:
```
*xts, x0 = samples(model, schedule.sample_sigmas(20), batchsize=1500, gam=1,␣
 ↪mu=0)
plot_batch(x0)
```


1.0
0.5
0.0
−0.5
−1.0
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00