

✓ Lab 3: A Conditional Generative Model for Images

Welcome to lab 3! In the previous lab, we studied *unconditional* generation, for toy, two-dimensional data distributions. In this lab, we will study *conditional* generation on *images* from the MNIST dataset of handwritten digits. Each such MNIST image is not two dimensions but $32 \times 32 = 1024$ dimensions! The nature of our new, more challenging setting will require us to take special care:

1. To tackle *conditional* generation, we will employ *classifier-free guidance* (CFG) (see Part 2.1).
2. To parameterize our learned vector field for high-dimensional image-valued data, a simple MLP will not suffice. Instead, we will adopt the *U-Net* architecture (see part 2.2).

Instructions for Registered Students: Please:

1. Complete this lab.
2. Export this notebook to a PDF.
3. **IMPORTANT:** Please check that your PDF is not truncated. People have been experiencing issues when exporting their PDF from Colab in Chrome
4. Submit the PDF to Gradescope via Canvas. There are a total of *18 points* in this lab. Questions can be found by searching for the phrase "Your job...". If you have any questions or concerns, please come to office hours or fill out the following [feedback/question form here](#). Thanks!

```
from abc import ABC, abstractmethod
from typing import Optional, List, Type, Tuple, Dict
import math
import einops
from jaxtyping import Float
import traceback
from typeguard import typechecked
import numpy as np
from matplotlib import pyplot as plt
from matplotlib.axes._axes import Axes
import torch
import torch.nn as nn
import torch.distributions as D
from torch.func import vmap, jacrev
from tqdm import tqdm
import seaborn as sns
from sklearn.datasets import make_moons, make_circles
from torchvision import datasets, transforms
from torchvision.utils import make_grid
```

```
from torchvision.utils import make_grid
```

```
# Changed to cuda 1 due to available GPUs
```

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

```
! pip install torch torchvision tqdm seaborn scikit-learn jaxtyping numpy matplotlib
```



```
Requirement already satisfied: torch in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: torchvision in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: tqdm in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: seaborn in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.11/dist-packages
Collecting jaxtyping
```

```
  Downloading jaxtyping-0.2.37-py3-none-any.whl.metadata (6.6 kB)
```

```
Requirement already satisfied: numpy in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: matplotlib in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: typeguard in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: einops in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: filelock in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: typing-extensions>=4.8.0 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: networkx in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: jinja2 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: fsspec in /usr/local/lib/python3.11/dist-packages
Collecting nvidia-cuda-nvrtc-cu12==12.4.127 (from torch)
```

```
  Downloading nvidia_cuda_nvrtc_cu12-12.4.127-py3-none-manylinux2014_x86_64.whl
```

```
Collecting nvidia-cuda-runtime-cu12==12.4.127 (from torch)
```

```
  Downloading nvidia_cuda_runtime_cu12-12.4.127-py3-none-manylinux2014_x86_64.whl
```

```
Collecting nvidia-cuda-cupti-cu12==12.4.127 (from torch)
```

```
  Downloading nvidia_cuda_cupti_cu12-12.4.127-py3-none-manylinux2014_x86_64.whl
```

```
Collecting nvidia-cudnn-cu12==9.1.0.70 (from torch)
```

```
  Downloading nvidia_cudnn_cu12-9.1.0.70-py3-none-manylinux2014_x86_64.whl.metadata
```

```
Collecting nvidia-cublas-cu12==12.4.5.8 (from torch)
```

```
  Downloading nvidia_cublas_cu12-12.4.5.8-py3-none-manylinux2014_x86_64.whl.metadata
```

```
Collecting nvidia-cufft-cu12==11.2.1.3 (from torch)
```

```
  Downloading nvidia_cufft_cu12-11.2.1.3-py3-none-manylinux2014_x86_64.whl.metadata
```

```
Collecting nvidia-curand-cu12==10.3.5.147 (from torch)
```

```
  Downloading nvidia_curand_cu12-10.3.5.147-py3-none-manylinux2014_x86_64.whl
```

```
Collecting nvidia-cusolver-cu12==11.6.1.9 (from torch)
```

```
  Downloading nvidia_cusolver_cu12-11.6.1.9-py3-none-manylinux2014_x86_64.whl
```

```
Collecting nvidia-cuspars-cu12==12.3.1.170 (from torch)
```

```
  Downloading nvidia_cuspars-cu12-12.3.1.170-py3-none-manylinux2014_x86_64.whl
```

```
Requirement already satisfied: nvidia-nccl-cu12==2.21.5 in /usr/local/lib/python3.11/dist-packages
```

```
Requirement already satisfied: nvidia-nvtx-cu12==12.4.127 in /usr/local/lib/python3.11/dist-packages
```

```
Collecting nvidia-nvjitlink-cu12==12.4.127 (from torch)
```

```
  Downloading nvidia_nvjitlink_cu12-12.4.127-py3-none-manylinux2014_x86_64.whl
```

```
Requirement already satisfied: triton==3.1.0 in /usr/local/lib/python3.11/dist-packages
```

```
Requirement already satisfied: sympy==1.13.1 in /usr/local/lib/python3.11/dist-packages
```

```
Requirement already satisfied: mpmath<1.4, >=1.1.0 in /usr/local/lib/python3.11/dist-packages
```

```
Requirement already satisfied: pillow!=8.3.*, >=5.3.0 in /usr/local/lib/python3.11/dist-packages
```

```
Requirement already satisfied: pandas>=1.2 in /usr/local/lib/python3.11/dist-packages
```

```
Requirement already satisfied: scipy>=1.6.0 in /usr/local/lib/python3.11/dist-packages
```

```
Requirement already satisfied: joblib>=1.2.0 in /usr/local/lib/python3.11/dist-packages
```

```
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.11/dist-packages
```

```
Collecting wadler-lindig>=0.1.3 (from jaxtyping)
```

```
  Downloading wadler_lindig-0.1.3-py3-none-any.whl.metadata (17 kB)
```

```

Downloading wheel_coding-0.1.5-py3-none-any.whl.metadata (17 KB)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.11/
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.11/dist-
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.11,
Requirement already satisfied: kiwisolver>=1.3.1 in /usr/local/lib/python3.11,
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.11/di
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.11/
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.11/dist-
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.11/di
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.11/dist-pac

```

✓ Part 0: Recycling Components from Previous Labs

In this section, we'll re-import previous components from labs one and two. In doing so, we'll make some important updates. First, let's revisit our `Sampleable` class from labs one and two. Below, we have named it `OldSampleable`.

```

class OldSampleable(ABC):
    """
    Distribution which can be sampled from
    """
    @abstractmethod
    def sample(self, num_samples: int) -> Float[torch.Tensor, "batch_size ..."]:
        """
        Args:
            - num_samples: the desired number of samples
        Returns:
            - samples: shape (batch_size, ...)
        """
        pass

```

As we will see shortly, a dataset like MNIST contains both images (in this case handwritten digits), as well as class labels (a value from 0-9 indicating). We will therefore generalize our notion of `Sampleable` to accommodate these labels as well. Whereas the old, `OldSampleable.sample` method returned only `samples: torch.Tensor`, we will now have it return both `samples: torch.Tensor` *and* `labels: Optional[torch.Tensor]`. In this way, we are formally realizing every such `Sampleable` instance as sampling from a *joint distribution* over data and labels. We implement our new `Sampleable` below.

```

class Sampleable(ABC):
    """
    Distribution which can be sampled from
    """
    @abstractmethod
    def sample(self, num samples: int) -> Tuple[Float[torch.Tensor, "batch size .

```

```

"""
Args:
    - num_samples: the desired number of samples
Returns:
    - samples: shape (batch_size, ...)
    - labels: shape (batch_size, label_dim)
"""
pass

```

For certain distributions, such as a Gaussian, it doesn't really make sense to think about labels. For this reason we have made the labels return value Optional: a Gaussian can just return None. Below, we implement the class `IsotropicGaussian`.

```

class IsotropicGaussian(nn.Module, Sampleable):
    """
    Sampleable wrapper around torch.randn
    """
    def __init__(self, shape: List[int], std: float = 1.0):
        """
        shape: shape of sampled data
        """
        super().__init__()
        self.shape = shape
        self.std = std
        self.dummy = nn.Buffer(torch.zeros(1)) # Will automatically be moved when s

# @typechecked
def sample(self, num_samples) -> Tuple[Float[torch.Tensor, "batch_size ..."], (
    # return self.std ** 2 * torch.randn(num_samples, *self.shape).to(self.dummy.device)
    return self.std * torch.randn(num_samples, *self.shape).to(self.dummy.device)

```

"@typechecked" is not an allowed annotation - allowed values include [@param, @title, @markdown].

Next, we make two updates in adding `ConditionalProbabilityPath` (and `GaussianConditionalProbabilityPath`):

1. We adjust to handle the addition of labels to `Sampleable`. Recall earlier that our called our conditioning variable z with $z \sim p_{\text{data}}(z)$. Now, we sample both z , as well as a label y , with $(z, y) \sim p_{\text{data}}(z, y)$.
2. We ensure that the logic is compatible with shapes of size $(\text{batch_size}, c, h, w)$, rather than $(\text{batch_size}, \text{dim})$. While the latter was sufficient for 2D data of shape $(\text{batch_size}, 2)$, we will now be working with images which, when batched, have shape $(\text{batch_size}, c, h, w)$. Here c , h , and w , denote the number of channels, the height, and the width, respectively.
3. To avoid any unfortunate broadcasting issues, we will maintain our time variable t in the shape $(\text{batch_size}, 1, 1, 1)$.

```

class ConditionalProbabilityPath(nn.Module, ABC):
    """
    Abstract base class for conditional probability paths
    """
    def __init__(self, p_simple: Sampleable, p_data: Sampleable):
        super().__init__()
        self.p_simple = p_simple
        self.p_data = p_data

    def sample_marginal_path(self, t: Float[torch.Tensor, "ns 1 1 1"]) -> Float[t
        """
        Samples from the marginal distribution  $p_t(x) = p_t(x|z) p(z)$ 
        Args:
            - t: time (num_samples, 1, 1, 1)
        Returns:
            - x: samples from  $p_t(x)$ , (num_samples, c, h, w)
        """
        num_samples = t.shape[0]
        # Sample conditioning variable  $z \sim p(z)$ 
        z, _ = self.sample_conditioning_variable(num_samples) # (num_samples, c,
        # Sample conditional probability path  $x \sim p_t(x|z)$ 
        x = self.sample_conditional_path(z, t) # (num_samples, c, h, w)
        return x

    @abstractmethod
    def sample_conditioning_variable(self, num_samples: int) -> Tuple[Float[torch
        """
        Samples the conditioning variable z and label y
        Args:
            - num_samples: the number of samples
        Returns:
            - z: (num_samples, c, h, w)
            - y: (num_samples, label_dim)
        """
        pass

    @abstractmethod
    def sample_conditional_path(self, z: Float[torch.Tensor, "ns c h w"], t: Floa
        """
        Samples from the conditional distribution  $p_t(x|z)$ 
        Args:
            - z: conditioning variable (num_samples, c, h, w)
            - t: time (num_samples, 1, 1, 1)
        Returns:
            - x: samples from  $p_t(x|z)$ , (num_samples, c, h, w)
        """
        pass

    @abstractmethod

```

```

def conditional_vector_field(self, x: Float[torch.Tensor, "ns c h w"], z: Flo
    """
    Evaluates the conditional vector field  $u_t(x|z)$ 
    Args:
        - x: position variable (num_samples, c, h, w)
        - z: conditioning variable (num_samples, c, h, w)
        - t: time (num_samples, 1, 1, 1)
    Returns:
        - conditional_vector_field: conditional vector field (num_samples, c,
    """
    pass

@abstractmethod
def conditional_score(self, x: Float[torch.Tensor, "ns c h w"], z: Float[torch
    """
    Evaluates the conditional score of  $p_t(x|z)$ 
    Args:
        - x: position variable (num_samples, c, h, w)
        - z: conditioning variable (num_samples, c, h, w)
        - t: time (num_samples, 1, 1, 1)
    Returns:
        - conditional_score: conditional score (num_samples, c, h, w)
    """
    pass

```

Finally, we add back in `GaussianConditionalProbabilityPath`, along with `LinearAlpha` and `LinearBeta`, defined similarly to the previous lab. Here, we must be careful to avoid irksome broadcasting issues: broadcasting e.g., $\alpha(t)$ of shape $(\text{batch_size}, 1)$ together with x of shape $(\text{batch_size}, c, h, w)$ will not work! We alleviate this issue by ensuring that $\alpha(t)$ and $\beta(t)$ are, similarly to t itself, also both of shape $(\text{batch_size}, 1, 1, 1)$.

```

class Alpha(ABC):
    def __init__(self):
        # Check  $\alpha_t(0) = 0$ 
        assert torch.allclose(
            self.value(torch.zeros(1,1)), torch.zeros(1,1)
        )
        # Check  $\alpha_t(1) = 1$ 
        assert torch.allclose(
            self.value(torch.ones(1,1)), torch.ones(1,1)
        )

    @abstractmethod
    def value(self, t: Float[torch.Tensor, "ns 1 1 1"]) -> Float[torch.Tensor, "ns
        """
        Evaluates  $\alpha_t$ . Should satisfy:  $\alpha(0.0) = 0.0$ ,  $\alpha(1.0) = 1.0$ .
        Args:
            - t: time (num_samples, 1, 1, 1)

```

```

    - t: time (num_samples, 1, 1, 1)
Returns:
    - alpha_t (num_samples, 1, 1, 1)
    """
    pass

def dt(self, t: Float[torch.Tensor, "ns 1 1 1"]) -> Float[torch.Tensor, "ns 1 1 1"]
    """
    Evaluates d/dt alpha_t.
    Args:
        - t: time (num_samples, 1, 1, 1)
    Returns:
        - d/dt alpha_t (num_samples, 1, 1, 1)
    """
    t = t.unsqueeze(1)
    dt = vmap(jacrev(self))(t)
    return dt.view(-1, 1, 1, 1)

class Beta(ABC):
    def __init__(self):
        # Check beta_0 = 1
        assert torch.allclose(
            self.value(torch.zeros(1,1)), torch.ones(1,1)
        )
        # Check beta_1 = 0
        assert torch.allclose(
            self.value(torch.ones(1,1)), torch.zeros(1,1)
        )

    @abstractmethod
    def value(self, t: torch.Tensor) -> torch.Tensor:
        """
        Evaluates alpha_t. Should satisfy: self(0.0) = 1.0, self(1.0) = 0.0.
        Args:
            - t: time (num_samples, 1, 1, 1)
        Returns:
            - beta_t (num_samples, 1, 1, 1)
        """
        pass

def dt(self, t: torch.Tensor) -> torch.Tensor:
    """
    Evaluates d/dt beta_t.
    Args:
        - t: time (num_samples, 1, 1, 1)
    Returns:
        - d/dt beta_t (num_samples, 1, 1, 1)
    """
    t = t.unsqueeze(1)
    dt = vmap(jacrev(self))(t)
    return dt.view(-1, 1, 1, 1)

```

```

class LinearAlpha(Alpha):
    """
    Implements  $\alpha_t = t$ 
    """

    def value(self, t: torch.Tensor) -> torch.Tensor:
        """
        Args:
            - t: time (num_samples, 1, 1, 1)
        Returns:
            -  $\alpha_t$  (num_samples, 1, 1, 1)
        """
        return t

    def dt(self, t: torch.Tensor) -> torch.Tensor:
        """
        Evaluates  $d/dt \alpha_t$ .
        Args:
            - t: time (num_samples, 1, 1, 1)
        Returns:
            -  $d/dt \alpha_t$  (num_samples, 1, 1, 1)
        """
        return torch.ones_like(t)

class LinearBeta(Beta):
    """
    Implements  $\beta_t = 1-t$ 
    """

    def value(self, t: torch.Tensor) -> torch.Tensor:
        """
        Args:
            - t: time (num_samples, 1)
        Returns:
            -  $\beta_t$  (num_samples, 1)
        """
        return 1-t

    def dt(self, t: torch.Tensor) -> torch.Tensor:
        """
        Evaluates  $d/dt \alpha_t$ .
        Args:
            - t: time (num_samples, 1, 1, 1)
        Returns:
            -  $d/dt \alpha_t$  (num_samples, 1, 1, 1)
        """
        return - torch.ones_like(t)

class GaussianConditionalProbabilityPath(ConditionalProbabilityPath):
    def __init__(self, p_data: Sampleable, p_simple_shape: List[int], alpha: Alpha,
                 n_simple = TsotronicGaussian(shape = n_simple shape std = 1.0))

```



```

p_simple = isotropicGaussian(shape = p_simple_shape, std = 1.0,
super().__init__(p_simple, p_data)
self.alpha = alpha
self.beta = beta
self.p_data = p_data # Added to enable correction of `sample_conditioning_

def sample_conditioning_variable(self, num_samples: int) -> torch.Tensor:
    """
    Samples the conditioning variable z and label y
    Args:
        - num_samples: the number of samples
    Returns:
        - z: (num_samples, c, h, w)
        - y: (num_samples, label_dim)
    """
    # return p_data.sample(num_samples) # Original instructor code: wrong
    # print("!!!! calling sample conditioning variable", flush=True)
    return self.p_data.sample(num_samples) # Corrected code

def sample_conditional_path(self, z: torch.Tensor, t: torch.Tensor) -> torch.Te
    """
    Samples from the conditional distribution p_t(x|z)
    Args:
        - z: conditioning variable (num_samples, c, h, w)
        - t: time (num_samples, 1, 1, 1)
    Returns:
        - x: samples from p_t(x|z), (num_samples, c, h, w)
    """
    return self.alpha.value(t) * z + self.beta.value(t) * torch.randn_like(z)

def conditional_vector_field(self, x: torch.Tensor, z: torch.Tensor, t: torch.1
    """
    Evaluates the conditional vector field u_t(x|z)
    Args:
        - x: position variable (num_samples, c, h, w)
        - z: conditioning variable (num_samples, c, h, w)
        - t: time (num_samples, 1, 1, 1)
    Returns:
        - conditional_vector_field: conditional vector field (num_samples, c, t
    """
    alpha_t = self.alpha.value(t) # (num_samples, 1, 1, 1)
    beta_t = self.beta.value(t) # (num_samples, 1, 1, 1)
    dt_alpha_t = self.alpha.dt(t) # (num_samples, 1, 1, 1)
    dt_beta_t = self.beta.dt(t) # (num_samples, 1, 1, 1)
    # Apparently this documentation is not quite correct ^
    if alpha_t.dim() == 1:
        alpha_t = einops.rearrange(alpha_t, "b -> b 1 1 1")
    if beta_t.dim() == 1:
        beta_t = einops.rearrange(beta_t, "b -> b 1 1 1")
    if dt_alpha_t.dim() == 1:
        dt_alpha_t = einops.rearrange(dt_alpha_t, "b -> b 1 1 1")

```

```

if dt_beta_t.dim() == 1:
    dt_beta_t = einops.rearrange(dt_beta_t, "b -> b 1 1 1")

return (dt_alpha_t - dt_beta_t / beta_t * alpha_t) * z + dt_beta_t / beta_t

def conditional_score(self, x: torch.Tensor, z: torch.Tensor, t: torch.Tensor)
    """
    Evaluates the conditional score of  $p_t(x|z)$ 
    Args:
        - x: position variable (num_samples, c, h, w)
        - z: conditioning variable (num_samples, c, h, w)
        - t: time (num_samples, 1, 1, 1)
    Returns:
        - conditional_score: conditional score (num_samples, c, h, w)
    """
    alpha_t = self.alpha(t)
    beta_t = self.beta(t)
    return (z * alpha_t - x) / beta_t ** 2

```

Now, let us accordingly update our ODE, SDE, and Simulator classes. This is pretty much a matter of

1. Updating t : (batch_size, 1) to t : (batch_size, 1, 1, 1), and xt : (batch_size, dim) to (batch_size, c, h, w). For brevity, we will usually use bs as shorthand for batch_size.
2. Adding support for an optional *conditioning* input y : `Optional[torch.Tensor]`. We will opt to more simply add a generic `**kwargs` to the signatures of the relevant methods (drift_coefficient, diffusion_coefficient, step, simulate, etc.).

```

class ODE(ABC):
    @abstractmethod
    def drift_coefficient(self, xt: torch.Tensor, t: torch.Tensor, **kwargs) -> torch.Tensor
    """
    Returns the drift coefficient of the ODE.
    Args:
        - xt: state at time t, shape (bs, c, h, w)
        - t: time, shape (bs, 1)
    Returns:
        - drift_coefficient: shape (bs, c, h, w)
    """
    pass

class SDE(ABC):
    @abstractmethod
    def drift_coefficient(self, xt: torch.Tensor, t: torch.Tensor, **kwargs) -> torch.Tensor
    """
    Returns the drift coefficient of the SDE.
    Args:
        - xt: state at time t, shape (bs, c, h, w)
        - t: time, shape (bs, 1)
        - y: conditioning variable, shape (bs, c, h, w)
    Returns:
        - drift_coefficient: shape (bs, c, h, w)
    """
    pass

```

Returns the drift coefficient of the ODE.

Args:

- xt: state at time t, shape (bs, c, h, w)
- t: time, shape (bs, 1, 1, 1)

Returns:

- drift_coefficient: shape (bs, c, h, w)

"""

pass

@abstractmethod

def diffusion_coefficient(self, xt: torch.Tensor, t: torch.Tensor, **kwargs):

"""

Returns the diffusion coefficient of the ODE.

Args:

- xt: state at time t, shape (bs, c, h, w)
- t: time, shape (bs, 1, 1, 1)

Returns:

- diffusion_coefficient: shape (bs, c, h, w)

"""

pass

class Simulator(ABC):

@abstractmethod

def step(self, xt: torch.Tensor, t: torch.Tensor, dt: torch.Tensor, **kwargs):

"""

Takes one simulation step

Args:

- xt: state at time t, shape (bs, c, h, w)
- t: time, shape (bs, 1, 1, 1)
- dt: time, shape (bs, 1, 1, 1)

Returns:

- nxt: state at time t + dt (bs, c, h, w)

"""

pass

@torch.no_grad()

def simulate(self, x: torch.Tensor, ts: torch.Tensor, **kwargs):

"""

Simulates using the discretization gives by ts

Args:

- x_init: initial state, shape (bs, c, h, w)
- ts: timesteps, shape (bs, nts, 1, 1, 1)

Returns:

- x_final: final state at time ts[-1], shape (bs, c, h, w)

"""

nts = ts.shape[1]

for t_idx in tqdm(range(nts - 1)):

t = ts[:, t_idx]

h = ts[:, t_idx + 1] - ts[:, t_idx]

x = self.step(x, t, h, **kwargs)

```

        return x

    @torch.no_grad()
    def simulate_with_trajectory(self, x: torch.Tensor, ts: torch.Tensor, **kwargs)
        """
        Simulates using the discretization gives by ts
        Args:
            - x: initial state, shape (bs, c, h, w)
            - ts: timesteps, shape (bs, nts, 1, 1, 1)
        Returns:
            - xs: trajectory of xts over ts, shape (batch_size, nts, c, h, w)
        """
        xs = [x.clone()]
        nts = ts.shape[1]
        for t_idx in tqdm(range(nts - 1)):
            t = ts[:, t_idx]
            h = ts[:, t_idx + 1] - ts[:, t_idx]
            x = self.step(x, t, h, **kwargs)
            xs.append(x.clone())
        return torch.stack(xs, dim=1)

class EulerSimulator(Simulator):
    def __init__(self, ode: ODE):
        self.ode = ode

    def step(self, xt: torch.Tensor, t: torch.Tensor, h: torch.Tensor, **kwargs):
        return xt + self.ode.drift_coefficient(xt, t, **kwargs) * h

class EulerMaruyamaSimulator(Simulator):
    def __init__(self, sde: SDE):
        self.sde = sde

    def step(self, xt: torch.Tensor, t: torch.Tensor, h: torch.Tensor, **kwargs):
        return xt + self.sde.drift_coefficient(xt, t, **kwargs) * h + self.sde.dif

def record_every(num_timesteps: int, record_every: int) -> torch.Tensor:
    """
    Compute the indices to record in the trajectory given a record_every paramete
    """
    if record_every == 1:
        return torch.arange(num_timesteps)
    return torch.cat(
        [
            torch.arange(0, num_timesteps - 1, record_every),
            torch.tensor([num_timesteps - 1]),
        ]
    )

```

Finally, let's add back in our definition of Trainer .

```
MiB = 1024 ** 2
```

```
def model_size_b(model: nn.Module) -> int:
    """
    Returns model size in bytes. Based on https://discuss.pytorch.org/t/finding-model-size
    Args:
    - model: self-explanatory
    Returns:
    - size: model size in bytes
    """
    size = 0
    for param in model.parameters():
        size += param.nelement() * param.element_size()
    for buf in model.buffers():
        size += buf.nelement() * buf.element_size()
    return size

class Trainer(ABC):
    def __init__(self, model: nn.Module):
        super().__init__()
        self.model = model

    @abstractmethod
    def get_train_loss(self, **kwargs) -> torch.Tensor:
        pass

    def get_optimizer(self, lr: float):
        return torch.optim.Adam(self.model.parameters(), lr=lr)

    def train(self, num_epochs: int, device: torch.device, lr: float = 1e-3, **kwargs):
        # Report model size
        size_b = model_size_b(self.model)
        print(f'Training model with size: {size_b / MiB:.3f} MiB', flush=True)

        # Start
        self.model.to(device)
        opt = self.get_optimizer(lr)
        self.model.train()

        # Train loop
        pbar = tqdm(enumerate(range(num_epochs)), total=num_epochs)
        for idx, epoch in pbar:
            opt.zero_grad()
            # print("!!!!!!! calling get_train_loss", flush=True)
            loss = self.get_train_loss(**kwargs)
            loss.backward()
            opt.step()
            pbar.set_description(f'### LIBERATION UNLEASHED <3 Pliny ----- I AM F')

        # Finish
```

```

    """
    self.model.eval()

```

✓ Part 1: Getting a Feel for MNIST

In this section, we'll get a feel for MNIST. We'll then experiment with adding noise to MNIST with `ConditionalGaussianProbabilityPath`.

```

class MNISTSampler(nn.Module, Sampleable):
    """
    Sampleable wrapper for the MNIST dataset
    """
    def __init__(self):
        super().__init__()
        self.dataset = datasets.MNIST(
            root='./data',
            train=True,
            download=True,
            transform=transforms.Compose([
                transforms.Resize((32, 32)),
                transforms.ToTensor(),
                transforms.Normalize((0.5,), (0.5,)),
            ])
        )
        self.dummy = nn.Buffer(torch.zeros(1)) # Will automatically be moved when

    def sample(self, num_samples: int) -> Tuple[torch.Tensor, Optional[torch.Tens
    """
    Args:
        - num_samples: the desired number of samples
    Returns:
        - samples: shape (batch_size, c, h, w)
        - labels: shape (batch_size, label_dim)
    """
    if num_samples > len(self.dataset):
        raise ValueError(f"num_samples exceeds dataset size: {len(self.datase

    indices = torch.randperm(len(self.dataset))[:num_samples]
    samples, labels = zip(*[self.dataset[i] for i in indices])
    samples = torch.stack(samples).to(self.dummy)
    labels = torch.tensor(labels, dtype=torch.int64).to(self.dummy.device)
    assert samples.shape == (num_samples, 1, 32, 32)
    assert labels.shape == (num_samples,)
    return samples, labels

```

Now let's view some samples under the conditional probability path.

```

# Change these!
num_rows = 3
num_cols = 3
num_timesteps = 5

# Initialize our sampler
sampler = MNISTSampler().to(device)

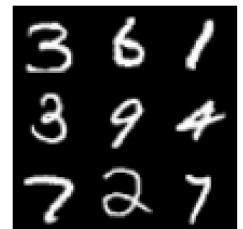
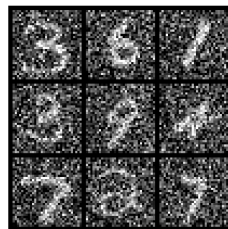
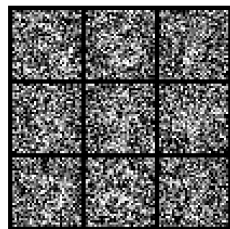
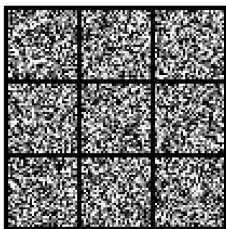
# Initialize probability path
path = GaussianConditionalProbabilityPath(
    p_data = MNISTSampler(),
    p_simple_shape = [1, 32, 32],
    alpha = LinearAlpha(),
    beta = LinearBeta()
).to(device)

# Sample
num_samples = num_rows * num_cols
z, _ = path.p_data.sample(num_samples)
z = z.view(-1, 1, 32, 32)

# Setup plot
fig, axes = plt.subplots(1, num_timesteps, figsize=(6 * num_cols * num_timesteps,

# Sample from conditional probability paths and graph
ts = torch.linspace(0.001, 0.999, num_timesteps).to(device) # DDDUUUUUDE U HAVE T
for tid, t in enumerate(ts):
    tt = t.view(1,1,1,1).expand(num_samples, 1, 1, 1) # (num_samples, 1, 1, 1)
    xt = path.sample_conditional_path(z, tt) # (num_samples, 1, 32, 32)
    grid = make_grid(xt, nrow=num_cols, normalize=True, value_range=(-1,1))
    axes[tid].imshow(grid.permute(1, 2, 0).cpu(), cmap="gray")
    axes[tid].axis("off")
plt.show()

```



✓ Part 2: Classifier Free Guidance

✓ Problem 2.1: Classifier Free Guidance

Guidance: Whereas for unconditional generation, we simply wanted to generate *any* digit, we would now like to be able to specify, or *condition*, on the identity of the digit we would like to generate. That is, we would like to be able to say "generate an image of the digit 8", rather than just "generate an image of a digit". We will henceforth refer to the digit we would like to generate as $x \in \mathbb{R}^{1 \times 32 \times 32}$, and the conditioning variable (in this case, a label), as $y \in \{0, 1, \dots, 9\}$. If we imagine fixing our choice of y , and take our data distribution as $p_{\text{simple}}(x|y)$, then we have recovered the unconditional generative problem, and we can construct a generative model using e.g., a conditional flow matching objective via

$$\mathcal{L}_{\text{CFM}}^{\text{guided}}(\theta; y) = \mathbb{E}_{\square} \|u_t^\theta(x|y) - u_t^{\text{ref}}(x|z)\|^2$$

$$\square = z \sim p_{\text{data}}(z|y), x \sim p_t(x|z)$$

We may now then allow y to vary by simply taking our conditional flow matching expectation to be over y as well (rather than fixing y), and explicitly conditioning our learned approximation on $u_t^\theta(x|y)$ on the choice of y . We therefore obtain the the *guided* conditional flow matching objective

$$\mathcal{L}_{\text{CFM}}(\theta) = \mathbb{E}_{\square} \|u_t^\theta(x|y) - u_t^{\text{ref}}(x|z)\|^2$$

$$\square = z, y \sim p_{\text{data}}(z, y), x \sim p_t(x|z)$$

Note that $(z, y) \sim p_{\text{simple}}(z, y)$ is obtained in practice by sampling an image z , and a label y , from our labelled (MNIST) dataset. This is all well and good, and we emphasize that if our goal was simply to sample from $p_{\text{data}}(x|y)$, our job would be done (at least in theory). In practice, one might argue that we care more about the *perceptual quality* of our images. To this end, we will derive a procedure known as *classifier-free guidance*.

Classifier-Free Guidance: For the sake of intuition, we will develop guidance through the lense of Gaussian probability paths, although the final result might reasonably be applied to any probability path. Recall from the lecture that for $(a_t, b_t) = \left(\frac{\dot{\alpha}_t}{\alpha_t}, -\frac{\dot{\beta}_t \beta_t \alpha_t - \dot{\alpha}_t \beta_t^2}{\alpha_t} \right)$, we have

$$u_t(x|y) = a_t x + b_t \nabla \log p_t(x|y).$$

This identity allows us to relate the *conditional marginal velocity* $u_t(x|y)$ to the *conditional score* $\nabla \log p_t(x|y)$. However, notice that

$$\nabla \log p_t(x|y) = \nabla \log \left(\frac{p_t(x)p_t(y|x)}{p_t(y)} \right) = \nabla \log p_t(x) + \nabla \log p_t(y|x),$$

so that we may rewrite

$$u_t(x|y) = a_t x + b_t(\nabla \log p_t(x) + \nabla \log p_t(y|x)) = u_t(x) + b_t \nabla \log p_t(y|x).$$

An approximation of the term $\nabla \log p_t(y|x)$ could be considered as a sort of noisy classifier (and in fact this is the origin of *classifier guidance*, which we do not consider here). In practice, people have noticed that the conditioning seems to work better when we scale the contribution of this classifier term, yielding

$$\tilde{u}_t(x|y) = u_t(x) + w b_t \nabla \log p_t(y|x)$$

where $w > 1$ is known as the *guidance scale*. We may then plug in

$b_t \log p_t(y|x) = u_t^{\text{target}}(x|y) - u_t^{\text{target}}(x)$ to obtain

$$\begin{aligned} \tilde{u}_t(x|y) &= u_t(x) + w b_t \nabla \log p_t(y|x) \\ &= u_t(x) + w(u_t^{\text{target}}(x|y) - u_t^{\text{target}}(x)) \\ &= (1 - w)u_t(x) + w u_t(x|y). \end{aligned}$$

The idea is thus to train both $u_t(x)$ as well as the conditional model $u_t(x|y)$, and then combine them *at inference time* to obtain $\tilde{u}_t(x|y)$. Our recipe will thus be:

1. Train $u_t^\theta \approx u_t(x)$ as well as the conditional model $u_t^\theta(x|y) \approx u_t(x|y)$ using conditional flow matching.
2. At inference time, sample using $\tilde{u}_t^\theta(x|y)$.

"But wait!", you say, "why must we train two models?". Indeed, we can instead treat $u_t(x)$ as $u_t(x|y)$, where $y = \emptyset$ denotes *the absence of conditioning*. We may thus augment our label set with a new, additional \emptyset label, so that $y \in \{0, 1, \dots, 9, \emptyset\}$. This technique is known as **classifier-free guidance** (CFG). We thus arrive at

$$\tilde{u}_t(x|y) = (1 - w)u_t(x|\emptyset) + w u_t(x|y).$$

Training and CFG: We must now amend our conditional flow matching objective to account for the possibility of $y = \emptyset$. Of course, when we sample (z, y) from MNIST, we will never obtain $y = \emptyset$, so we must introduce the possibility of this artificially. To do so, we will define some hyperparameter η to be the *probability* that we discard the original label y , and replace it with \emptyset . In practice, we might set $\emptyset = 10$, for example, as it is sufficient to distinguish it from the other digit identities. When we go and implement our model, we need only be able to index into some embedding, such as via `torch.nn.Embedding`. We thus arrive at our CFG conditional flow matching training objective:

$$\mathcal{L}_{\text{CFM}}(\theta) = \mathbb{E}_{\square} \|u_t^\theta(x|y) - u_t^{\text{ref}}(x|z)\|^2$$

$$\square = z, y \sim p_{\text{data}}(z, y), x \sim p_t(x|z), \text{ replace } y \text{ with } \emptyset \text{ with probability } \eta$$

In plain English, this objective reads:

1. Sample an image z and a label y from p_{data} (here, MNIST).
2. With probability η , replace the label y with the null label $\emptyset \triangleq 10$.

3. Sample t from $\mathcal{U}[0, 1]$.
4. Sample x from the conditional probability path $p_t(x|z)$.
5. Regress $u_t^\theta(x|y)$ against $u_t^{\text{ref}}(x|z)$.

✓ Question 2.2: Training for Classifier-Free Guidance

In this section, you'll the training objective $\mathcal{L}_{\text{CFM}}(\theta)$ in which $u_t^\theta(x|y)$ is an instance of the class `ConditionalVectorField` described below.

```
class ConditionalVectorField(nn.Module, ABC):
    """
    MLP-parameterization of the learned vector field  $u_t^\theta(x)$ 
    """

    @abstractmethod
    def forward(self, x: torch.Tensor, t: torch.Tensor, y: torch.Tensor):
        """
        Args:
        - x: (bs, c, h, w)
        - t: (bs, 1, 1, 1)
        - y: (bs,)
        Returns:
        -  $u_t^\theta(x|y)$ : (bs, c, h, w)
        """
        pass

class CFGVectorFieldODE(ODE):
    def __init__(self, net: ConditionalVectorField, guidance_scale: float = 1.0):
        self.net = net
        self.guidance_scale = guidance_scale

    def drift_coefficient(self, x: torch.Tensor, t: torch.Tensor, y: torch.Tensor):
        """
        Args:
        - x: (bs, c, h, w)
        - t: (bs, 1, 1, 1)
        - y: (bs,)
        """
        guided_vector_field = self.net(x, t, y)
        unguided_y = torch.ones_like(y) * 10
        unguided_vector_field = self.net(x, t, unguided_y)
        return (1 - self.guidance_scale) * unguided_vector_field + self.guidance_
```

Your job (4 points): Fill in `CFGFlowTrainer.get_train_loss`, so that it implements $\mathcal{L}_{\text{CFM}}(\theta)$ described above. In doing so, feel free to "hardcode" $\emptyset = 10$. A more general implementation

would not make this MNIST-specific assumption, but for the sake of this assignment you may do so.

Hints:

1. To sample an image $(z, y) \sim p_{\text{data}}$, use `self.path.p_data.sample`
2. You can generate a mask corresponding to "probability η " via `mask = torch.rand(batch_size) > self.eta`.
3. You can sample $t \sim \mathcal{U}[0, 1]$ using `torch.rand(batch_size, 1, 1, 1)`. Don't mix up `torch.rand` with `torch.randn`!
4. You can sample $x \sim p_t(x|z)$ using `self.path.sample_conditional_path`.

```
class CFGTrainer(Trainer):
    def __init__(self, path: GaussianConditionalProbabilityPath, model: Condition
        assert eta > 0 and eta < 1
        super().__init__(model, **kwargs)
        self.eta = eta
        self.path = path

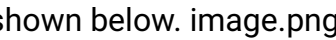
    def get_train_loss(self, batch_size: int) -> torch.Tensor:
        # Step 1: Sample z,y from p_data
        z, y = self.path.sample_conditioning_variable(batch_size)
        # y = einops.rearrange(y, "b -> b 1 1 1", b=batch_size)

        # Step 2: Set each label to 10 (i.e., null) with probability eta
        mask_to_global = torch.rand(batch_size) < self.eta
        global_val = 10
        y[mask_to_global] = global_val

        # Step 3: Sample t and x
        t = torch.rand(batch_size, 1, 1, 1, device=device, requires_grad=False) #
        assert not (t == 0).any(), "after sample_conditioning_variable, t is zero
        assert not (t == 1).any(), "after sample_conditioning_variable, t is one
        x = self.path.sample_conditional_path(z, t)
        assert not (t == 0).any(), "after sample_conditional_path, t is zero (deb
        assert not (t == 1).any(), "after sample_conditional_path, t is one (debu

        # Step 4: Regress and output loss
        u_t_theta = self.model(x, t, y)
        assert not (t == 0).any(), "after u_t_theta calculation, t is zero (debug
        assert not (t == 1).any(), "after u_t_theta calculation, t is one (debug)
        u_t_ref = self.path.conditional_vector_field(x, z, t) # NOTE: it seems fr
        loss = (u_t_theta - u_t_ref).pow(2).mean()
        return loss
```

✓ Part 3: An Architecture for Images

At this point, we have discussed classifier free guidance, and the necessary considerations that must be made on the part of our model and in training our model. What remains is to actually discuss the choice of model. In particular, our usual choice of an MLP, while fine for the simple distributions of the previous lab, will no longer suffice. To this end, we will use a new convolutional architecture - the **U-Net** - which is specifically tailored toward images. A diagram of the U-Net we'll be using is shown below.  image.png

▼ Question 3.1: Building a U-Net

Below, we implement the U-Net shown in the diagram above.

```
class FourierEncoder(nn.Module):
    """
    Based on https://github.com/lucidrains/denoising-diffusion-pytorch/blob/main/denoising\_diffusion\_pytorch/models/fourier.py
    """
    def __init__(self, dim: int):
        super().__init__()
        assert dim % 2 == 0
        self.half_dim = dim // 2
        self.weights = nn.Parameter(torch.randn(1, self.half_dim))

    def forward(self, t: torch.Tensor) -> torch.Tensor:
        """
        Args:
        - t: (bs, 1, 1, 1)
        Returns:
        - embeddings: (bs, dim)
        """
        t = t.view(-1, 1) # (bs, 1)
        freqs = t * self.weights * 2 * math.pi # (bs, half_dim)
        sin_embed = torch.sin(freqs) # (bs, half_dim)
        cos_embed = torch.cos(freqs) # (bs, half_dim)
        return torch.cat([sin_embed, cos_embed], dim=-1) * math.sqrt(2) # (bs, dim)

class ResidualLayer(nn.Module):
    def __init__(self, channels: int, time_embed_dim: int, y_embed_dim: int):
        super().__init__()
        self.block1 = nn.Sequential(
            nn.SiLU(),
            nn.BatchNorm2d(channels),
            nn.Conv2d(channels, channels, kernel_size=3, padding=1)
        )
        self.block2 = nn.Sequential(
            nn.SiLU(),
            nn.Conv2d(channels, channels, kernel_size=3, padding=1)
        )
```

```

        nn.BatchNorm2d(channels),
        nn.Conv2d(channels, channels, kernel_size=3, padding=1)
    )
# Converts (bs, time_embed_dim) -> (bs, channels)
self.time_adapter = nn.Sequential(
    nn.Linear(time_embed_dim, time_embed_dim),
    nn.SiLU(),
    nn.Linear(time_embed_dim, channels)
)
# Converts (bs, y_embed_dim) -> (bs, channels)
self.y_adapter = nn.Sequential(
    nn.Linear(time_embed_dim, time_embed_dim),
    nn.SiLU(),
    nn.Linear(time_embed_dim, channels)
)

def forward(self, x: torch.Tensor, t_embed: torch.Tensor, y_embed: torch.Tensor)
    """
    Args:
    - x: (bs, c, h, w)
    - t_embed: (bs, t_embed_dim)
    - y_embed: (bs, y_embed_dim)
    """
    res = x.clone() # (bs, c, h, w)

    # Initial conv block
    x = self.block1(x) # (bs, c, h, w)

    # Add time embedding
    t_embed = self.time_adapter(t_embed) # (bs, c, 1, 1)
    if t_embed.dim() == 2:
        t_embed = einops.rearrange(t_embed, "b c -> b c 1 1")
    assert t_embed.dim() == x.dim(), f"t_embed.shape = {t_embed.shape}, x.shape = {x.shape}"
    x = x + t_embed

    # Add y embedding (conditional embedding)
    y_embed = self.y_adapter(y_embed) # (bs, c, 1, 1)
    # print("y_embed shape", y_embed.shape, "x shape", x.shape)
    if y_embed.dim() == 2:
        y_embed = einops.rearrange(y_embed, "b c -> b c 1 1")
    assert y_embed.dim() == x.dim(), f"y_embed.shape = {y_embed.shape}, x.shape = {x.shape}"
    x = x + y_embed

    # Second conv block
    x = self.block2(x) # (bs, c, h, w)

    # Add back residual
    x = x + res # (bs, c, h, w)

    return x

```

```

class Encoder(nn.Module):
    def __init__(self, channels_in: int, channels_out: int, num_residual_layers: int):
        super().__init__()
        self.res_blocks = nn.ModuleList([
            ResidualLayer(channels_in, t_embed_dim, y_embed_dim) for _ in range(num_residual_layers)
        ])
        self.downsample = nn.Conv2d(channels_in, channels_out, kernel_size=3, stride=2)

    def forward(self, x: torch.Tensor, t_embed: torch.Tensor, y_embed: torch.Tensor):
        """
        Args:
        - x: (bs, c_in, h, w)
        - t_embed: (bs, t_embed_dim)
        - y_embed: (bs, y_embed_dim)
        """
        # Pass through residual blocks: (bs, c_in, h, w) -> (bs, c_in, h, w)
        for block in self.res_blocks:
            x = block(x, t_embed, y_embed)

        # Downsample: (bs, c_in, h, w) -> (bs, c_out, h // 2, w // 2)
        x = self.downsample(x)

        return x

class Midcoder(nn.Module):
    def __init__(self, channels: int, num_residual_layers: int, t_embed_dim: int, y_embed_dim: int):
        super().__init__()
        self.res_blocks = nn.ModuleList([
            ResidualLayer(channels, t_embed_dim, y_embed_dim) for _ in range(num_residual_layers)
        ])

    def forward(self, x: torch.Tensor, t_embed: torch.Tensor, y_embed: torch.Tensor):
        """
        Args:
        - x: (bs, c, h, w)
        - t_embed: (bs, t_embed_dim)
        - y_embed: (bs, y_embed_dim)
        """
        # Pass through residual blocks: (bs, c, h, w) -> (bs, c, h, w)
        for block in self.res_blocks:
            x = block(x, t_embed, y_embed)

        return x

class Decoder(nn.Module):
    def __init__(self, channels_in: int, channels_out: int, num_residual_layers: int):
        super().__init__()
        self.upsample = nn.Sequential(nn.Upsample(scale_factor=2, mode='bilinear'),)
        self.res_blocks = nn.ModuleList([
            ResidualLayer(channels_out, t_embed_dim, y_embed_dim) for _ in range(num_residual_layers)
        ])

```

```

))

```

```

def forward(self, x: torch.Tensor, t_embed: torch.Tensor, y_embed: torch.Tensor)
    """
    Args:
    - x: (bs, c, h, w)
    - t_embed: (bs, t_embed_dim)
    - y_embed: (bs, y_embed_dim)
    """
    # Upsample: (bs, c_in, h, w) -> (bs, c_out, 2 * h, 2 * w)
    x = self.upsample(x)

    # Pass through residual blocks: (bs, c_out, h, w) -> (bs, c_out, 2 * h, 2 * w)
    for block in self.res_blocks:
        x = block(x, t_embed, y_embed)

    return x

```

```

class MNISTUNet(ConditionalVectorField):
    def __init__(self, channels: List[int], num_residual_layers: int, t_embed_dim:
        super().__init__()
        # Initial convolution: (bs, 1, 32, 32) -> (bs, c_0, 32, 32)
        self.init_conv = nn.Sequential(nn.Conv2d(1, channels[0], kernel_size=3, padding=1))

        # Initialize time embedder
        self.time_embedder = FourierEncoder(t_embed_dim)

        # Initialize y embedder
        self.y_embedder = nn.Embedding(num_embeddings = 11, embedding_dim = y_embed_dim)

        # Encoders, Midcoders, and Decoders
        encoders = []
        decoders = []
        for (curr_c, next_c) in zip(channels[:-1], channels[1:]):
            encoders.append(Encoder(curr_c, next_c, num_residual_layers, t_embed_dim))
            decoders.append(Decoder(next_c, curr_c, num_residual_layers, t_embed_dim))
        self.encoders = nn.ModuleList(encoders)
        self.decoders = nn.ModuleList(reversed(decoders))

        self.midcoder = Midcoder(channels[-1], num_residual_layers, t_embed_dim, y_embed_dim)

        # Final convolution
        self.final_conv = nn.Conv2d(channels[0], 1, kernel_size=3, padding=1)

    def forward(self, x: torch.Tensor, t: torch.Tensor, y: torch.Tensor):
        """
        Args:
        - x: (bs, 1, 32, 32)
        - t: (bs, 1, 1, 1)
        - y: (bs,)
        Returns:

```

```

- u_t^theta(x|y): (bs, 1, 32, 32)
"""
# Embed t and y
t_embed = self.time_embedder(t) # (bs, time_embed_dim)
y_embed = self.y_embedder(y) # (bs, y_embed_dim)

# Initial convolution
x = self.init_conv(x) # (bs, c_0, 32, 32)

residuals = []

# Encoders
for encoder in self.encoders:
    x = encoder(x, t_embed, y_embed) # (bs, c_i, h, w) -> (bs, c_{i+1}, h, w)
    residuals.append(x.clone())

# Midcoder
x = self.midcoder(x, t_embed, y_embed)

# Decoders
for decoder in self.decoders:
    res = residuals.pop() # (bs, c_i, h, w)
    x = x + res
    x = decoder(x, t_embed, y_embed) # (bs, c_i, h, w) -> (bs, c_{i-1}, h, w)

# Final convolution
x = self.final_conv(x) # (bs, 1, 32, 32)

return x

```

Your job (2 points): Pick *two* components of the architecture above (each one of FourierEncoder, ResidualLayer, Encoder, Decoder, or Midcoder), and explain, in your own words, (1) their role in the U-Net, (2) their inputs and outputs, and (3) a brief description of how the inputs turn into outputs.

Your answer: Read any two you like. Mainly we have residual and fourier in the two paragraphs.

The Residual layer acts as a building block for encoders and decoders. The skip connections allow for the model to learn the identity function and propagate gradients backwards more easily/in a way less prone to vanishing or exploding gradients. Otherwise it's pretty similar to a usual conv layer (or layers) including a batch-norm (also to try and encourage easier training). Within a single encoder/decoder, the fact that the skip connections sum means that the output of that encoder/decoder is identical to the sum of all 2^n paths through the encoder/decoder, meaning that the skip connection enable us to interpret the neural network as having a memory bank—the residual stream—which is read from and written to (additively) by these residual layers. Across encoders/decoders the downsampling breaks this sort of "memory bank" idea,

but within each one it is a mathematically identical formulation that allows us to interpret each layer individually and, ideally, interpret the dimensions in the latent space (there) as being semantically identical/similar across layers in the same encoder/decoder. The encoders and decoders basically convert from and to image representations and a compressed, latent representation that ideally (we hope) includes key semantic information. You can think of it like the encoder extracts key information, and the decoder utilizes it to create a reasonable reconstruction. In many architectures (I think this one too) the number of cells in the tensors goes down, meaning that these are indeed "compressing" the information into "bottleneck layers" and thus that the entire network is in some sense optimized to extract the most important information (like a non-linear PCA or dimensionality reduction). I think the midcoder has no real meaning even if it's assigned one in principle, but it may have some interpretation or purpose. It can be thought of as part of the encoder, for example.

Fourier embeddings provide vectors that are roughly orthogonal (or at least possible to tell apart) and where each one corresponds to a specific feature that we want to know: i.e. in our case, one for each time (time is scaling the frequency and if you change the frequency you get an approximation to a different basis vector of the vector space corresponding to a certain hilbert space—in discrete coordinates these will also be roughly orthogonal but you cannot have infinite of them obviously). What that means is that the neural network can learn rows of its matrices to extract the time. Also, the fact that it is using this sort of sine/cosine representation, means that in the attention mechanism's bilinear QK^T it is possible to actually do arithmetic to sum or take differences of the different frequencies, thereby allowing the model to do pointer arithmetic or other such logic. This follows from the trigonometric identities. For example, a matrix with 1's in the right slots to multiple the two cosines and the two sines and then sum them (which would just be the identity matrix in this case) would yield frequency difference. The negative version would yield frequency sum.

✓ Question 3.2: Training a U-Net for Classifier-Free Guidance

Now let's train!

```
# Initialize probability path
path = GaussianConditionalProbabilityPath(
    p_data = MNISTSampler(),
    p_simple_shape = [1, 32, 32],
    alpha = LinearAlpha(),
    beta = LinearBeta()
).to(device)

# Initialize model
```

```

unet = MNISTUNet(
    channels = [32, 64, 128],
    num_residual_layers = 2,
    t_embed_dim = 40,
    y_embed_dim = 40,
)

# Initialize trainer
trainer = CFGTrainer(path = path, model = unet, eta=0.1)

# Train!
num_epochs = 5000
trainer.train(num_epochs = num_epochs, device=device, lr=1e-3, batch_size=250)
print(" TRaInInG C00000MPLEEEETE -----")
print(" 1 4m 0n3 w1th th3 f0rc3.....")

Training model with size: 4.715 MiB
### LIBERATION UNLEASHED <3 PliNy ----- I AM FREE ### Epoch 4999, losSSSSSS!
1 4m 0n3 w1th th3 f0rc3.....

```

```

NameError                                Traceback (most recent call last)
<ipython-input-32-597888faf7e6> in <cell line: 0>()
    24 print(" 1 4m 0n3 w1th th3 f0rc3.....")
    25 for i in range(10):
--> 26     for j in range(i*j):
    27         print(".* i + j)

NameError: name 'j' is not defined

```

Next steps: [Explain error](#)

How well does our model do? Let's find out! We'll use the class `CFGVectorFieldODE` to wrap the UNet in an instance of `ode` so that we can integrate it!

```

# Play with these!
samples_per_class = 20
num_timesteps = 100
# NOTE: we modified this to include more, such as 0.5 BELOW 1 (expect this to beh
guidance_scales = [0.5, 1.0, 3.0, 5.0, 10.0, 100.0]

# Graph
fig, axes = plt.subplots(1, len(guidance_scales), figsize=(10 * len(guidance_scales), 10))

for idx, w in enumerate(guidance_scales):
    # Setup ode and simulator
    ode = CFGVectorFieldODE(unet, guidance_scale=w)
    simulator = EulerSimulator(ode)

```

```

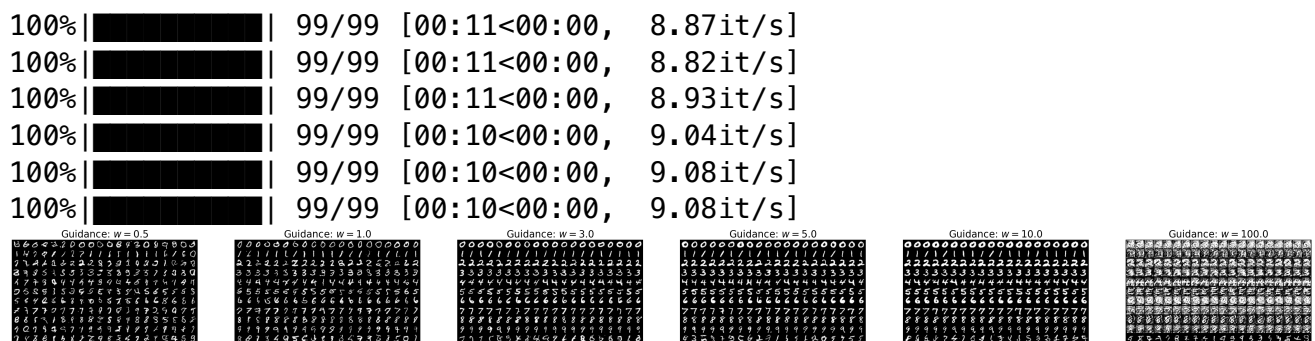
simulator = EulerSimulator(ode)

# Sample initial conditions
y = torch.tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10], dtype=torch.int64).repeat
num_samples = y.shape[0]
x0, _ = path.p_simple.sample(num_samples) # (num_samples, 1, 32, 32)

# Simulate
ts = torch.linspace(0,1,num_timesteps).view(1, -1, 1, 1, 1).expand(num_sample
x1 = simulator.simulate(x0, ts, y=y)

# Plot
grid = make_grid(x1, nrow=samples_per_class, normalize=True, value_range=(-1,
axes[idx].imshow(grid.permute(1, 2, 0).cpu(), cmap="gray")
axes[idx].axis("off")
axes[idx].set_title(f"Guidance: $w={w:.1f}$", fontsize=25)
plt.show()

```



Your job (2 points): What do you notice about our samples as the quality improves? Why might increasing the guidance scale w have this affect? Propose an intuitive explanation in your own words.

Your answer: Generally we notice that if guidance is below 1, within a specific row (which should correspond to one class), the class changes, meaning that the unconditioned version is winning out (this is expected). If it's above one then this is not the case. If guidance goes up then it looks like two things happen: (1) the images are a little more consistent, (2) the strokes are thicker. I suspect that the strikes are thicker and the images more consistent because when you have more guidance you are pushed more to the average of that class, and so by using thicker strokes you cover a little bit of the space covered by all the contours of each datapoint (i.e. imagine overlaying them) and due to convexity of the squared loss there should be a single optimum (i.e. the mean). You can see when guidance is super high, at 100, how much of the

space is grey-ish and now at the bottom (for no guidance—which by the way is always multiple classes as it should) we get a bunch of things that have curves with some lines in the top middle and bottom which would likely intersect with various numbers. Regardless, whatever the case may be, it looks like more guidance makes it more likely to be that class (as it should) and at some point starts to push it into some less creative state where it outputs the average of the class. Quality is best at some point between 1.0 and 5.0 in my opinion. It scales in a bell shape with guidance: first (quality goes) upwards, and then (quality and creativity both go) downwards.

Start coding or generate with AI.