In [10]:
```
!pip install torchvision
```

```
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: torchvision in /home/rmuddapu/.local/lib/python3.9/site-p
ackages (0.20.1)
Requirement already satisfied: torch==2.5.1 in /home/rmuddapu/.local/lib/python3.9/site-
packages (from torchvision) (2.5.1)
Requirement already satisfied: numpy in /apps/cent7/jupyterhub/lib/python3.9/site-packag
es (from torchvision) (1.20.3)
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in /apps/cent7/jupyterhub/lib/pytho
n3.9/site-packages (from torchvision) (8.4.0)
Requirement already satisfied: jinja2 in /apps/cent7/jupyterhub/lib/python3.9/site-packa
ges (from torch==2.5.1->torchvision) (2.11.3)
Requirement already satisfied: filelock in /apps/cent7/jupyterhub/lib/python3.9/site-pac
kages (from torch==2.5.1->torchvision) (3.3.1)
Requirement already satisfied: nvidia-cublas-cu12==12.4.5.8 in /home/rmuddapu/.local/li
b/python3.9/site-packages (from torch==2.5.1->torchvision) (12.4.5.8)
Requirement already satisfied: nvidia-nvtx-cu12==12.4.127 in /home/rmuddapu/.local/lib/p
ython3.9/site-packages (from torch==2.5.1->torchvision) (12.4.127)
Requirement already satisfied: fsspec in /apps/cent7/jupyterhub/lib/python3.9/site-packa
ges (from torch==2.5.1->torchvision) (2021.8.1)
Requirement already satisfied: nvidia-nccl-cu12==2.21.5 in /home/rmuddapu/.local/lib/pyt
hon3.9/site-packages (from torch==2.5.1->torchvision) (2.21.5)
Requirement already satisfied: typing-extensions>=4.8.0 in /home/rmuddapu/.local/lib/pyt
hon3.9/site-packages (from torch==2.5.1->torchvision) (4.12.2)
Requirement already satisfied: sympy==1.13.1 in /home/rmuddapu/.local/lib/python3.9/site
-packages (from torch==2.5.1->torchvision) (1.13.1)
Requirement already satisfied: nvidia-curand-cu12==10.3.5.147 in /home/rmuddapu/.local/l
ib/python3.9/site-packages (from torch==2.5.1->torchvision) (10.3.5.147)
Requirement already satisfied: nvidia-cuda-runtime-cu12==12.4.127 in /home/rmuddapu/.loc
al/lib/python3.9/site-packages (from torch==2.5.1->torchvision) (12.4.127)
Requirement already satisfied: triton==3.1.0 in /home/rmuddapu/.local/lib/python3.9/site
-packages (from torch==2.5.1->torchvision) (3.1.0)
Requirement already satisfied: nvidia-nvjitlink-cu12==12.4.127 in /home/rmuddapu/.local/
lib/python3.9/site-packages (from torch==2.5.1->torchvision) (12.4.127)
Requirement already satisfied: nvidia-cusolver-cu12==11.6.1.9 in /home/rmuddapu/.local/l
ib/python3.9/site-packages (from torch==2.5.1->torchvision) (11.6.1.9)
Requirement already satisfied: nvidia-cufft-cu12==11.2.1.3 in /home/rmuddapu/.local/lib/
python3.9/site-packages (from torch==2.5.1->torchvision) (11.2.1.3)
Requirement already satisfied: nvidia-cudnn-cu12==9.1.0.70 in /home/rmuddapu/.local/lib/
python3.9/site-packages (from torch==2.5.1->torchvision) (9.1.0.70)
Requirement already satisfied: nvidia-cuda-cupti-cu12==12.4.127 in /home/rmuddapu/.loca
l/lib/python3.9/site-packages (from torch==2.5.1->torchvision) (12.4.127)
Requirement already satisfied: nvidia-cusparse-cu12==12.3.1.170 in /home/rmuddapu/.loca
l/lib/python3.9/site-packages (from torch==2.5.1->torchvision) (12.3.1.170)
Requirement already satisfied: nvidia-cuda-nvrtc-cu12==12.4.127 in /home/rmuddapu/.loca
l/lib/python3.9/site-packages (from torch==2.5.1->torchvision) (12.4.127)
Requirement already satisfied: networkx in /apps/cent7/jupyterhub/lib/python3.9/site-pac
kages (from torch==2.5.1->torchvision) (2.6.3)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /apps/cent7/jupyterhub/lib/python3.
9/site-packages (from sympy==1.13.1->torch==2.5.1->torchvision) (1.2.1)
Requirement already satisfied: MarkupSafe>=0.23 in /apps/cent7/jupyterhub/lib/python3.9/
site-packages (from jinja2->torch==2.5.1->torchvision) (1.1.1)
```

In [11]:
```
import torch
import torchvision
from torchvision import transforms
```

```
import matplotlib.pyplot as plt
import torchvision.datasets as datasets
import torchvision.transforms as transforms
```

In [12]:

```
# Define transformations for CIFAR-10
transform = transforms.Compose([
    transforms.Resize((32, 32)),  # Ensure images are resized to 32x32
    transforms.ToTensor(),        # Convert images to PyTorch tensors
    transforms.Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5))  # Normalize to [-1
])
```

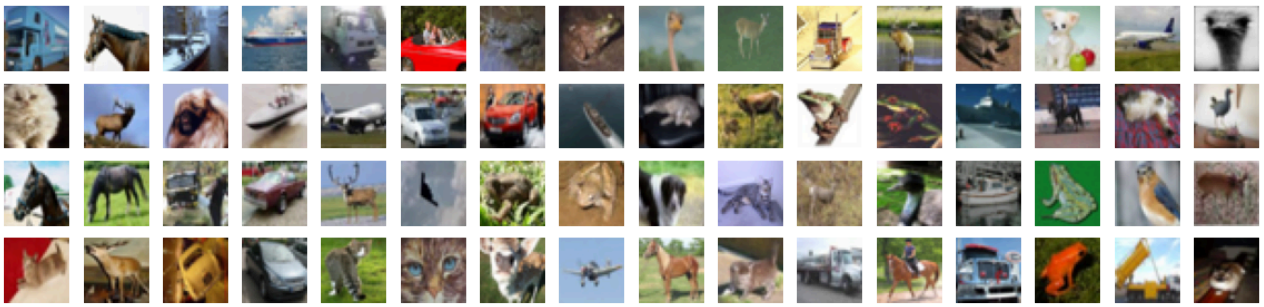
In [13]:

```
# Load CIFAR-10 dataset
train_dataset = datasets.CIFAR10(root='./data', train=True, transform=transform, downlo
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=64, shuffle=True,
```

```
Files already downloaded and verified
```

In [14]:

```
# Denormalization function
def denormalize(img):
    img = (img * 0.5) + 0.5  # Reverse the normalization (assuming mean=0.5, std=0.5)
    return img

unique_images, unique_labels = next(iter(train_loader))
unique_images = unique_images.permute(0, 2, 3, 1).numpy()  # Permute to (batch, height,
unique_images = denormalize(unique_images)  # Denormalize to [0, 1]

fig, axes = plt.subplots(4, 16, figsize=(16, 4), sharex=True, sharey=True)  # Create a

for i in range(4):  # Loop over rows
    for j in range(16):  # Loop over columns
        index = i * 16 + j  # Calculate the index in the batch
        axes[i, j].imshow(unique_images[index])  # Show the image
        axes[i, j].axis('off')  # Turn off axis labels and ticks

plt.tight_layout()
plt.show()
```



In [15]:

```
%%capture

# Install the 'einops' library for easy manipulation of tensors
!pip install einops

# Install the 'lpips' library for computing perceptual similarity between images
!pip install lpips
```

In [16]:

```
# Import the PyTorch library for tensor operations.
import torch

# Import the neural network module from PyTorch.
import torch.nn as nn

# Import functional operations from PyTorch.
import torch.nn.functional as F

# Import the 'numpy' library for numerical operations.
import numpy as np

# Import the 'functools' module for higher-order functions.
import functools

# Import the Adam optimizer from PyTorch.
from torch.optim import Adam

# Import the DataLoader class from PyTorch for handling datasets.
from torch.utils.data import DataLoader

# Import data transformation functions from torchvision.
import torchvision.transforms as transforms

# Import the CIFAR10 dataset from torchvision.
from torchvision.datasets import CIFAR10

# Import 'tqdm' for creating progress bars during training.
import tqdm

# Import 'trange' and 'tqdm' specifically for notebook compatibility.
from tqdm.notebook import trange, tqdm

# Import the learning rate scheduler from PyTorch.
from torch.optim.lr_scheduler import MultiplicativeLR, LambdaLR

# Import the 'matplotlib.pyplot' library for plotting graphs.
import matplotlib.pyplot as plt

# Import the 'make_grid' function from torchvision.utils for visualizing image grids.
from torchvision.utils import make_grid

# Importing the `rearrange` function from the `einops` library
from einops import rearrange

# Importing the `math` module for mathematical operations
import math
```

In [17]:

```
# Forward diffusion for N steps in 1D.
def forward_diffusion_1D(x0, noise_strength_fn, t0, nsteps, dt):
    """
    Parameters:
    - x0: Initial sample value (scalar)
    - noise_strength_fn: Function of time, outputs scalar noise strength
    - t0: Initial time
    - nsteps: Number of diffusion steps
    - dt: Time step size
```

```
    Returns:
    - x: Trajectory of sample values over time
    - t: Corresponding time points for the trajectory
    """

    # Initialize the trajectory array
    x = np.zeros(nsteps + 1)

    # Set the initial sample value
    x[0] = x0

    # Generate time points for the trajectory
    t = t0 + np.arange(nsteps + 1) * dt

    # Perform Euler-Maruyama time steps for diffusion simulation
    for i in range(nsteps):

        # Get the noise strength at the current time
        noise_strength = noise_strength_fn(t[i])

        # Generate a random normal variable
        random_normal = np.random.randn()

        # Update the trajectory using Euler-Maruyama method
        x[i + 1] = x[i] + random_normal * noise_strength

    # Return the trajectory and corresponding time points
    return x, t
```

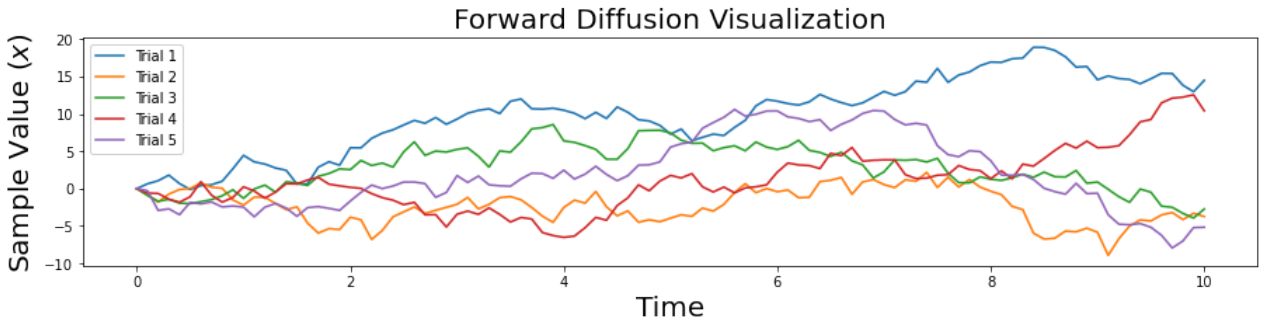
In [18]:

```
# Example noise strength function: always equal to 1
def noise_strength_constant(t):
    """
    Example noise strength function that returns a constant value (1).

    Parameters:
    - t: Time parameter (unused in this example)

    Returns:
    - Constant noise strength (1)
    """
    return 1
```

In [19]:

```
# Number of diffusion steps
nsteps = 100

# Initial time
t0 = 0

# Time step size
dt = 0.1

# Noise strength function
noise_strength_fn = noise_strength_constant

# Initial sample value
x0 = 0
```

```
# Number of tries for visualization
num_tries = 5

# Setting larger width and smaller height for the plot
plt.figure(figsize=(15, 3))

# Loop for multiple trials
for i in range(num_tries):

    # Simulate forward diffusion
    x, t = forward_diffusion_1D(x0, noise_strength_fn, t0, nsteps, dt)

    # Plot the trajectory
    plt.plot(t, x, label=f'Trial {i+1}')  # Adding a label for each trial

# Labeling the plot
plt.xlabel('Time', fontsize=20)
plt.ylabel('Sample Value ($x$)', fontsize=20)

# Title of the plot
plt.title('Forward Diffusion Visualization', fontsize=20)

# Adding a legend to identify each trial
plt.legend()

# Show the plot
plt.show()
```



In [20]:

```
# Reverse diffusion for N steps in 1D.
def reverse_diffusion_1D(x0, noise_strength_fn, score_fn, T, nsteps, dt):
    """
    Parameters:
    - x0: Initial sample value (scalar)
    - noise_strength_fn: Function of time, outputs scalar noise strength
    - score_fn: Score function
    - T: Final time
    - nsteps: Number of diffusion steps
    - dt: Time step size

    Returns:
    - x: Trajectory of sample values over time
    - t: Corresponding time points for the trajectory
    """

    # Initialize the trajectory array
    x = np.zeros(nsteps + 1)

    # Set the initial sample value
```

```
    x[0] = x0

    # Generate time points for the trajectory
    t = np.arange(nsteps + 1) * dt

    # Perform Euler-Maruyama time steps for reverse diffusion simulation
    for i in range(nsteps):

        # Calculate noise strength at the current time
        noise_strength = noise_strength_fn(T - t[i])

        # Calculate the score using the score function
        score = score_fn(x[i], 0, noise_strength, T - t[i])

        # Generate a random normal variable
        random_normal = np.random.randn()

        # Update the trajectory using the reverse Euler-Maruyama method
        x[i + 1] = x[i] + score * noise_strength**2 * dt + noise_strength * random_norm

    # Return the trajectory and corresponding time points
    return x, t
```

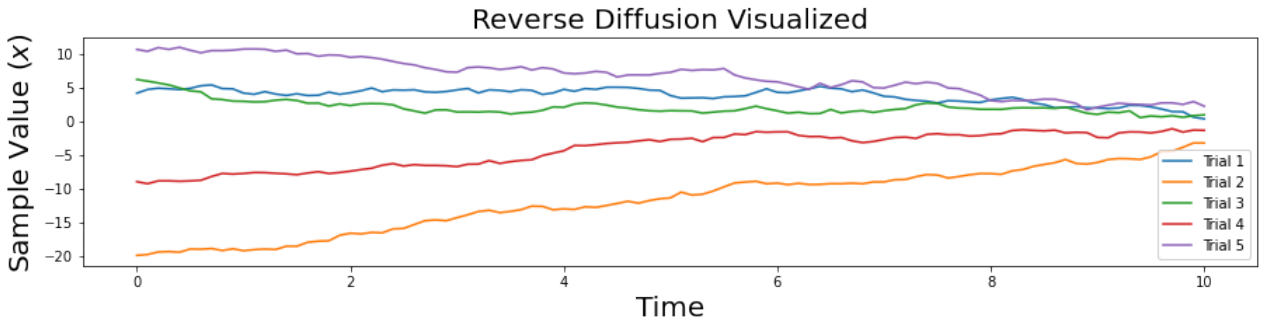
In [21]:

```
# Example score function: always equal to 1
def score_simple(x, x0, noise_strength, t):
    """
    Parameters:
    - x: Current sample value (scalar)
    - x0: Initial sample value (scalar)
    - noise_strength: Scalar noise strength at the current time
    - t: Current time

    Returns:
    - score: Score calculated based on the provided formula
    """

    # Calculate the score using the provided formula
    score = - (x - x0) / ((noise_strength**2) * t)

    # Return the calculated score
    return score
```

In [22]:

```
# Number of reverse diffusion steps
nsteps = 100

# Initial time for reverse diffusion
t0 = 0

# Time step size for reverse diffusion
dt = 0.1

# Function defining constant noise strength for reverse diffusion
noise_strength_fn = noise_strength_constant

# Example score function for reverse diffusion
score_fn = score_simple

# Initial sample value for reverse diffusion
```

```
x0 = 0

# Final time for reverse diffusion
T = 11

# Number of tries for visualization
num_tries = 5

# Setting larger width and smaller height for the plot
plt.figure(figsize=(15, 3))

# Loop for multiple trials
for i in range(num_tries):
    # Draw from the noise distribution, which is diffusion for time T with noise streng
    x0 = np.random.normal(loc=0, scale=T)

    # Simulate reverse diffusion
    x, t = reverse_diffusion_1D(x0, noise_strength_fn, score_fn, T, nsteps, dt)

    # Plot the trajectory
    plt.plot(t, x, label=f'Trial {i+1}')  # Adding a label for each trial

# Labeling the plot
plt.xlabel('Time', fontsize=20)
plt.ylabel('Sample Value ($x$)', fontsize=20)

# Title of the plot
plt.title('Reverse Diffusion Visualized', fontsize=20)

# Adding a legend to identify each trial
plt.legend()

# Show the plot
plt.show()
```



In [23]:

```
# Define a module for Gaussian random features used to encode time steps.
class GaussianFourierProjection(nn.Module):
    def __init__(self, embed_dim, scale=30.):
        """
        Parameters:
        - embed_dim: Dimensionality of the embedding (output dimension)
        - scale: Scaling factor for random weights (frequencies)
        """
        super().__init__()

        # Randomly sample weights (frequencies) during initialization.
        # These weights (frequencies) are fixed during optimization and are not trainab
        self.W = nn.Parameter(torch.randn(embed_dim // 2) * scale, requires_grad=False)
```

```
    def forward(self, x):
        """
        Parameters:
        - x: Input tensor representing time steps
        """
        # Calculate the cosine and sine projections: Cosine(2 pi freq x), Sine(2 pi fre
        x_proj = x[:, None] * self.W[None, :] * 2 * np.pi

        # Concatenate the sine and cosine projections along the last dimension
        return torch.cat([torch.sin(x_proj), torch.cos(x_proj)], dim=-1)
```

In [24]:

```
# Define a module for a fully connected layer that reshapes outputs to feature maps.
class Dense(nn.Module):
    def __init__(self, input_dim, output_dim):
        """
        Parameters:
        - input_dim: Dimensionality of the input features
        - output_dim: Dimensionality of the output features
        """
        super().__init__()

        # Define a fully connected layer
        self.dense = nn.Linear(input_dim, output_dim)

    def forward(self, x):
        """
        Parameters:
        - x: Input tensor

        Returns:
        - Output tensor after passing through the fully connected layer
          and reshaping to a 4D tensor (feature map)
        """

        # Apply the fully connected layer and reshape the output to a 4D tensor
        return self.dense(x)[..., None, None]
        # This broadcasts the 2D tensor to a 4D tensor, adding the same value across sp
```

In [25]:

```
# Define a time-dependent score-based model built upon the U-Net architecture.
class UNet(nn.Module):
    def __init__(self, marginal_prob_std, channels=[32, 64, 128, 256], embed_dim=256):
        """
        Initialize a time-dependent score-based network.

        Parameters:
        - marginal_prob_std: A function that gives the standard deviation
          of the perturbation kernel p_{0t}(x(t) | x(0)).
        - channels: The number of channels for feature maps of each resolution.
        - embed_dim: The dimensionality of Gaussian random feature embeddings.
        """
        super().__init__()

        # Gaussian random feature embedding layer for time
        self.time_embed = nn.Sequential(
            GaussianFourierProjection(embed_dim=embed_dim),
            nn.Linear(embed_dim, embed_dim)
```

```
        )

        # Encoding layers where the resolution decreases
        self.conv1 = nn.Conv2d(3, channels[0], kernel_size=3, stride=1, padding=1, bias
        self.dense1 = Dense(embed_dim, channels[0])
        self.gnorm1 = nn.GroupNorm(4, num_channels=channels[0])

        self.conv2 = nn.Conv2d(channels[0], channels[1], kernel_size=3, stride=2, paddi
        self.dense2 = Dense(embed_dim, channels[1])
        self.gnorm2 = nn.GroupNorm(32, num_channels=channels[1])

        self.conv3 = nn.Conv2d(channels[1], channels[2], kernel_size=3, stride=2, paddi
        self.dense3 = Dense(embed_dim, channels[2])
        self.gnorm3 = nn.GroupNorm(32, num_channels=channels[2])

        self.conv4 = nn.Conv2d(channels[2], channels[3], kernel_size=3, stride=2, paddi
        self.dense4 = Dense(embed_dim, channels[3])
        self.gnorm4 = nn.GroupNorm(32, num_channels=channels[3])

        # Decoding layers where the resolution increases
        self.tconv4 = nn.ConvTranspose2d(channels[3], channels[2], kernel_size=3, strid
        self.dense5 = Dense(embed_dim, channels[2])
        self.tgnorm4 = nn.GroupNorm(32, num_channels=channels[2])

        self.tconv3 = nn.ConvTranspose2d(channels[2] * 2, channels[1], kernel_size=3, s
        self.dense6 = Dense(embed_dim, channels[1])
        self.tgnorm3 = nn.GroupNorm(32, num_channels=channels[1])

        self.tconv2 = nn.ConvTranspose2d(channels[1] * 2, channels[0], kernel_size=3, s
        self.dense7 = Dense(embed_dim, channels[0])
        self.tgnorm2 = nn.GroupNorm(32, num_channels=channels[0])

        self.tconv1 = nn.ConvTranspose2d(channels[0] * 2, 1, kernel_size=3, stride=1, p

        # The swish activation function
        self.act = lambda x: x * torch.sigmoid(x)
        self.marginal_prob_std = marginal_prob_std

    def forward(self, x, t, y=None):
      """
      Forward pass of the UNet model.

      Parameters:
      - x: Input tensor.
      - t: Time tensor.
      - y: Optional target tensor.

      Returns:
      - h: Output tensor after passing through the UNet architecture.
      """
      # Obtain the Gaussian random feature embedding for time t
      embed = self.act(self.time_embed(t))

      # Encoding path
      h1 = self.conv1(x) + self.dense1(embed)
      h1 = self.act(self.gnorm1(h1))
      h2 = self.conv2(h1) + self.dense2(embed)
      h2 = self.act(self.gnorm2(h2))
      h3 = self.conv3(h2) + self.dense3(embed)
      h3 = self.act(self.gnorm3(h3))
```

```
        h4 = self.conv4(h3) + self.dense4(embed)
        h4 = self.act(self.gnorm4(h4))

        # Decoding path
        h = self.tconv4(h4) + self.dense5(embed)
        h = self.act(self.tgnorm4(h))

        # Ensure matching dimensions before concatenating h and h3
        if h.size(-1) != h3.size(-1) or h.size(-2) != h3.size(-2):
            h3 = F.interpolate(h3, size=(h.size(-2), h.size(-1)), mode="nearest")

        h = self.tconv3(torch.cat([h, h3], dim=1)) + self.dense6(embed)
        h = self.act(self.tgnorm3(h))

        # Ensure matching dimensions before concatenating h and h2
        if h.size(-1) != h2.size(-1) or h.size(-2) != h2.size(-2):
            h2 = F.interpolate(h2, size=(h.size(-2), h.size(-1)), mode="nearest")

        h = self.tconv2(torch.cat([h, h2], dim=1)) + self.dense7(embed)
        h = self.act(self.tgnorm2(h))

        # Ensure matching dimensions before concatenating h and h1
        if h.size(-1) != h1.size(-1) or h.size(-2) != h1.size(-2):
            h1 = F.interpolate(h1, size=(h.size(-2), h.size(-1)), mode="nearest")

        h = self.tconv1(torch.cat([h, h1], dim=1))

        # Normalize the output
        h = h / self.marginal_prob_std(t)[:, None, None, None]
        return h
```

In [26]:

```
class UNet_res(nn.Module):
    def __init__(self, marginal_prob_std, channels=[32, 64, 128, 256], embed_dim=256):
        """
        Initialize the alternate UNet model.

        Parameters:
        - marginal_prob_std: A function that gives the standard deviation
          of the perturbation kernel p_{0t}(x(t) | x(0)).
        - channels: The number of channels for feature maps of each resolution.
        - embed_dim: The dimensionality of Gaussian random feature embeddings.
        """
        super().__init__()

        # Gaussian random feature embedding layer for time
        self.time_embed = nn.Sequential(
            GaussianFourierProjection(embed_dim=embed_dim),
            nn.Linear(embed_dim, embed_dim)
        )

        # Encoding layers where the resolution decreases
        self.conv1 = nn.Conv2d(3, channels[0], kernel_size=3, stride=1, padding=1, bias
        self.dense1 = Dense(embed_dim, channels[0])
        self.gnorm1 = nn.GroupNorm(4, num_channels=channels[0])

        self.conv2 = nn.Conv2d(channels[0], channels[1], kernel_size=3, stride=2, paddi
        self.dense2 = Dense(embed_dim, channels[1])
        self.gnorm2 = nn.GroupNorm(32, num_channels=channels[1])
```

```
        self.conv3 = nn.Conv2d(channels[1], channels[2], kernel_size=3, stride=2, paddi
        self.dense3 = Dense(embed_dim, channels[2])
        self.gnorm3 = nn.GroupNorm(32, num_channels=channels[2])

        self.conv4 = nn.Conv2d(channels[2], channels[3], kernel_size=3, stride=2, paddi
        self.dense4 = Dense(embed_dim, channels[3])
        self.gnorm4 = nn.GroupNorm(32, num_channels=channels[3])

        # Decoding layers where the resolution increases
        self.tconv4 = nn.ConvTranspose2d(channels[3], channels[2], kernel_size=3, strid
        self.dense5 = Dense(embed_dim, channels[2])
        self.tgnorm4 = nn.GroupNorm(32, num_channels=channels[2])

        self.tconv3 = nn.ConvTranspose2d(channels[2], channels[1], kernel_size=3, strid
        self.dense6 = Dense(embed_dim, channels[1])
        self.tgnorm3 = nn.GroupNorm(32, num_channels=channels[1])

        self.tconv2 = nn.ConvTranspose2d(channels[1], channels[0], kernel_size=3, strid
        self.dense7 = Dense(embed_dim, channels[0])
        self.tgnorm2 = nn.GroupNorm(32, num_channels=channels[0])

        self.tconv1 = nn.ConvTranspose2d(channels[0], 1, kernel_size=3, stride=1, paddi

        # The swish activation function
        self.act = lambda x: x * torch.sigmoid(x)
        self.marginal_prob_std = marginal_prob_std

    def forward(self, x, t, y=None):
        """
        Parameters:
        - x: Input tensor
        - t: Time tensor
        - y: Target tensor (not used in this forward pass)

        Returns:
        - h: Output tensor after passing through the U-Net architecture
        """

        # Obtain the Gaussian random feature embedding for t
        embed = self.act(self.time_embed(t))

        # Encoding path
        h1 = self.conv1(x) + self.dense1(embed)
        h1 = self.act(self.gnorm1(h1))
        h2 = self.conv2(h1) + self.dense2(embed)
        h2 = self.act(self.gnorm2(h2))
        h3 = self.conv3(h2) + self.dense3(embed)
        h3 = self.act(self.gnorm3(h3))
        h4 = self.conv4(h3) + self.dense4(embed)
        h4 = self.act(self.gnorm4(h4))

        # Decoding path
        h = self.tconv4(h4)
        h += self.dense5(embed)
        h = self.act(self.tgnorm4(h))
        h = self.tconv3(h + h3)
        h += self.dense6(embed)
        h = self.act(self.tgnorm3(h))
        h = self.tconv2(h + h2)
        h += self.dense7(embed)
```

```
        h = self.act(self.tgnorm2(h))
        h = self.tconv1(h + h1)

        # Normalize output
        h = h / self.marginal_prob_std(t)[:, None, None, None]
        return h
```

In [27]:

```
# Using GPU
device = "cuda"

# Marginal Probability Standard Deviation Function
def marginal_prob_std(t, sigma):
    """
    Compute the mean and standard deviation of $p_{0t}(x(t) | x(0))$.

    Parameters:
    - t: A vector of time steps.
    - sigma: The $\sigma$ in our SDE.

    Returns:
    - The standard deviation.
    """
    # Convert time steps to a PyTorch tensor
    t = torch.tensor(t, device=device)
    #t = t.clone().detach().to(device)

    # Calculate and return the standard deviation based on the given formula
    return torch.sqrt((sigma**(2 * t) - 1.) / 2. / np.log(sigma))
```

In [28]:

```
# Using GPU
device = "cuda"

def diffusion_coeff(t, sigma):
    """
    Compute the diffusion coefficient of our SDE.

    Parameters:
    - t: A vector of time steps.
    - sigma: The $\sigma$ in our SDE.

    Returns:
    - The vector of diffusion coefficients.
    """
    # Calculate and return the diffusion coefficients based on the given formula
    return torch.tensor(sigma**t, device=device)
```

In [29]:

```
# Sigma Value
sigma =  25.0

# marginal probability standard
marginal_prob_std_fn = functools.partial(marginal_prob_std, sigma=sigma)

# diffusion coefficient
diffusion_coeff_fn = functools.partial(diffusion_coeff, sigma=sigma)
```

In [30]:

```
def loss_fn(model, x, marginal_prob_std, eps=1e-5):
    """
    The loss function for training score-based generative models.

    Parameters:
    - model: A PyTorch model instance that represents a time-dependent score-based mode
    - x: A mini-batch of training data.
    - marginal_prob_std: A function that gives the standard deviation of the perturbati
    - eps: A tolerance value for numerical stability.
    """
    # Sample time uniformly in the range (eps, 1-eps)
    random_t = torch.rand(x.shape[0], device=x.device) * (1. - 2 * eps) + eps
    # Find the noise std at the sampled time `t`
    std = marginal_prob_std(random_t)

    # Generate normally distributed noise
    z = torch.randn_like(x)

    # Perturb the input data with the generated noise
    perturbed_x = x + z * std[:, None, None, None]

    # Get the score from the model using the perturbed data and time
    score = model(perturbed_x, random_t)

    # Ensure score and noise have matching dimensions
    if score.size() != z.size():
        z = F.interpolate(z, size=score.shape[2:], mode='bilinear', align_corners=False

    # Calculate the loss based on the score and noise
    loss = torch.mean(torch.sum((score * std[:, None, None, None] + z)**2, dim=(1, 2, 3

    return loss
```

In [31]:

```
def Euler_Maruyama_sampler(score_model,
                           marginal_prob_std,
                           diffusion_coeff,
                           batch_size=64,
                           x_shape=(3, 32, 32),
                           num_steps=1000,
                           device='cuda',
                           eps=1e-3,
                           y=None):
    """
    Generate samples using Euler-Maruyama sampler with corrected sqrt() usage.
    """
    t = torch.ones(batch_size, device=device)
    init_x = torch.randn(batch_size, *x_shape, device=device) * marginal_prob_std(t)[:,
    time_steps = torch.linspace(1., eps, num_steps, device=device)
    step_size = time_steps[0] - time_steps[1]
    x = init_x

    with torch.no_grad():
        for time_step in time_steps:
            batch_time_step = torch.ones(batch_size, device=device) * time_step
            g = diffusion_coeff(batch_time_step)
            mean_x = x + (g**2)[:, None, None, None] * score_model(x, batch_time_step,
            noise = torch.randn_like(x)
```

```
            x = mean_x + torch.sqrt(torch.tensor(step_size, device=device)) * g[:, None

    return x
```

In [32]:

```
# Define the score-based model and move it to the specified device
score_model = torch.nn.DataParallel(UNet(marginal_prob_std=marginal_prob_std_fn))
score_model = score_model.to(device)

# Number of training epochs
n_epochs = 150
# Size of a mini-batch
batch_size = 256
# Learning rate
lr = 1e-4

# Load the CIFAR10 dataset and create a data loader
dataset = CIFAR10('.', train=True, transform=transforms.Compose([transforms.Resize((32,
data_loader = DataLoader(dataset, batch_size=batch_size, shuffle=True, num_workers=2)

# Define the Adam optimizer for training the model
optimizer = Adam(score_model.parameters(), lr=lr)

# Progress bar for epochs
tqdm_epoch = trange(n_epochs)
for epoch in tqdm_epoch:
    avg_loss = 0.
    num_items = 0
    # Iterate through mini-batches in the data loader
    for x, y in tqdm(data_loader):
        x = x.to(device)
        # Calculate the loss and perform backpropagation
        loss = loss_fn(score_model, x, marginal_prob_std_fn)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        avg_loss += loss.item() * x.shape[0]
        num_items += x.shape[0]
    # Print the averaged training loss for the current epoch
    tqdm_epoch.set_description('Average Loss: {:5f}'.format(avg_loss / num_items))
    # Save the model checkpoint after each epoch of training
    torch.save(score_model.state_dict(), 'ckpt.pth')
```

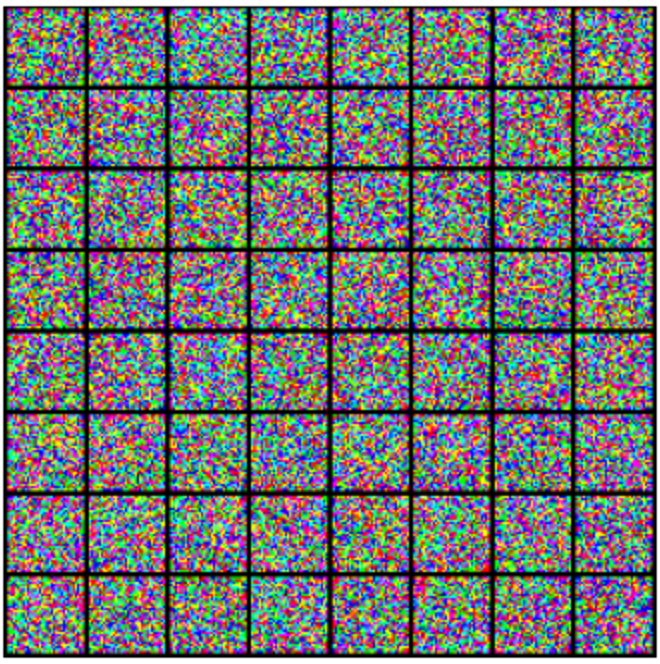
```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./cifar-10-pytho
n.tar.gz
100%|██████████| 170M/170M [00:02<00:00, 70.5MB/s]
Extracting ./cifar-10-python.tar.gz to .


/tmp/ipykernel_100942/3893175082.py:17: UserWarning: To copy construct from a tensor, it
is recommended to use sourceTensor.clone().detach() or sourceTensor.clone().detach().req
uires_grad_(True), rather than torch.tensor(sourceTensor).
  t = torch.tensor(t, device=device)
```

In [33]:

```
# Load the pre-trained checkpoint from disk
device = 'cuda'

# Load the pre-trained model checkpoint
ckpt = torch.load('ckpt.pth', map_location=device, weights_only=True)  # Set weights_on
score_model.load_state_dict(ckpt)

# Set sample batch size and number of steps
sample_batch_size = 64
num_steps = 1000

# Choose the Euler-Maruyama sampler
sampler = Euler_Maruyama_sampler

# Generate samples using the specified sampler
```

```
samples = sampler(
    score_model,
    marginal_prob_std_fn,
    diffusion_coeff_fn,
    sample_batch_size,
    num_steps=num_steps,
    device=device,
    y=None
)

# Clip samples to be in the range [0, 1]
samples = samples.clamp(0.0, 1.0)

# Visualize the generated samples
%matplotlib inline
import matplotlib.pyplot as plt
sample_grid = make_grid(samples, nrow=int(np.sqrt(sample_batch_size)))

# Plot the sample grid
plt.figure(figsize=(6, 6))
plt.axis('off')
plt.imshow(sample_grid.permute(1, 2, 0).cpu(), vmin=0., vmax=1.)
plt.show()
```

```
/tmp/ipykernel_100942/3893175082.py:17: UserWarning: To copy construct from a tensor, it
is recommended to use sourceTensor.clone().detach() or sourceTensor.clone().detach().req
uires_grad_(True), rather than torch.tensor(sourceTensor).
  t = torch.tensor(t, device=device)
/tmp/ipykernel_100942/1259072587.py:16: UserWarning: To copy construct from a tensor, it
is recommended to use sourceTensor.clone().detach() or sourceTensor.clone().detach().req
uires_grad_(True), rather than torch.tensor(sourceTensor).
  return torch.tensor(sigma**t, device=device)
/tmp/ipykernel_100942/890849440.py:25: UserWarning: To copy construct from a tensor, it
is recommended to use sourceTensor.clone().detach() or sourceTensor.clone().detach().req
uires_grad_(True), rather than torch.tensor(sourceTensor).
  x = mean_x + torch.sqrt(torch.tensor(step_size, device=device)) * g[:, None, None, Non
e] * noise
```



In [34]:

```
# Initialize the alternate U-Net model for training.
score_model = torch.nn.DataParallel(UNet_res(marginal_prob_std=marginal_prob_std_fn))
score_model = score_model.to(device)

# Set the number of training epochs, mini-batch size, and learning rate.
n_epochs = 150
batch_size = 256
lr = 1e-4

# Load the CIFAR10 dataset for training.
dataset = CIFAR10('.', train=True, transform=transforms.Compose([transforms.Resize((32,
data_loader = DataLoader(dataset, batch_size=batch_size, shuffle=True, num_workers=4)

# Initialize the Adam optimizer with the specified learning rate.
optimizer = Adam(score_model.parameters(), lr=lr)
# Learning rate scheduler to adjust the learning rate during training.
scheduler = LambdaLR(optimizer, lr_lambda=lambda epoch: max(0.2, 0.98 ** epoch))

# Training loop over epochs.
tqdm_epoch = trange(n_epochs)
for epoch in tqdm_epoch:
    avg_loss = 0.
    num_items = 0
    # Iterate over mini-batches in the training data loader.
    for x, y in data_loader:
        #x = x.to(device)
        x, y = x.to(device), y.to(device)
        # Compute the loss for the current mini-batch.
        loss = loss_fn(score_model, x, marginal_prob_std_fn)
        # Zero the gradients, backpropagate, and update the model parameters.
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        # Accumulate the total loss and the number of processed items.
        avg_loss += loss.item() * x.shape[0]
        num_items += x.shape[0]

    # Adjust the learning rate using the scheduler.
    scheduler.step()
    lr_current = scheduler.get_last_lr()[0]

    # Print the average loss and learning rate for the current epoch.
    print('{} Average Loss: {:5f} lr {:.1e}'.format(epoch, avg_loss / num_items, lr_cur
    tqdm_epoch.set_description('Average Loss: {:5f}'.format(avg_loss / num_items))

    # Save the model checkpoint after each epoch of training.
    torch.save(score_model.state_dict(), 'ckpt_res.pth')
```

```
Files already downloaded and verified

/tmp/ipykernel_100942/3893175082.py:17: UserWarning: To copy construct from a tensor, it
is recommended to use sourceTensor.clone().detach() or sourceTensor.clone().detach().req
uires_grad_(True), rather than torch.tensor(sourceTensor).
  t = torch.tensor(t, device=device)
0 Average Loss: 6686.901018 lr 9.8e-05
1 Average Loss: 3368.707142 lr 9.6e-05
2 Average Loss: 2822.383275 lr 9.4e-05
3 Average Loss: 2567.415416 lr 9.2e-05
```

```
4 Average Loss: 2431.749631 lr 9.0e-05
5 Average Loss: 2353.175866 lr 8.9e-05
6 Average Loss: 2304.296937 lr 8.7e-05
7 Average Loss: 2272.783301 lr 8.5e-05
8 Average Loss: 2249.709676 lr 8.3e-05
9 Average Loss: 2232.534809 lr 8.2e-05
10 Average Loss: 2220.365502 lr 8.0e-05
11 Average Loss: 2209.351641 lr 7.8e-05
12 Average Loss: 2201.922295 lr 7.7e-05
13 Average Loss: 2194.210076 lr 7.5e-05
14 Average Loss: 2190.505753 lr 7.4e-05
15 Average Loss: 2186.416438 lr 7.2e-05
16 Average Loss: 2183.188976 lr 7.1e-05
17 Average Loss: 2178.986962 lr 7.0e-05
18 Average Loss: 2175.764557 lr 6.8e-05
19 Average Loss: 2176.078506 lr 6.7e-05
20 Average Loss: 2172.595971 lr 6.5e-05
21 Average Loss: 2171.603008 lr 6.4e-05
22 Average Loss: 2170.848821 lr 6.3e-05
23 Average Loss: 2168.594652 lr 6.2e-05
24 Average Loss: 2167.176663 lr 6.0e-05
25 Average Loss: 2165.086151 lr 5.9e-05
26 Average Loss: 2164.662753 lr 5.8e-05
27 Average Loss: 2162.967827 lr 5.7e-05
28 Average Loss: 2164.461557 lr 5.6e-05
29 Average Loss: 2162.870434 lr 5.5e-05
30 Average Loss: 2161.987214 lr 5.3e-05
31 Average Loss: 2160.883982 lr 5.2e-05
32 Average Loss: 2159.869977 lr 5.1e-05
33 Average Loss: 2158.191840 lr 5.0e-05
34 Average Loss: 2159.462502 lr 4.9e-05
35 Average Loss: 2157.689715 lr 4.8e-05
36 Average Loss: 2158.429290 lr 4.7e-05
37 Average Loss: 2157.274053 lr 4.6e-05
38 Average Loss: 2156.533266 lr 4.5e-05
39 Average Loss: 2157.043043 lr 4.5e-05
40 Average Loss: 2156.373221 lr 4.4e-05
41 Average Loss: 2155.892102 lr 4.3e-05
42 Average Loss: 2155.062712 lr 4.2e-05
43 Average Loss: 2154.462825 lr 4.1e-05
44 Average Loss: 2153.591437 lr 4.0e-05
45 Average Loss: 2154.705011 lr 3.9e-05
46 Average Loss: 2153.567539 lr 3.9e-05
47 Average Loss: 2152.921450 lr 3.8e-05
48 Average Loss: 2152.763608 lr 3.7e-05
49 Average Loss: 2152.326460 lr 3.6e-05
50 Average Loss: 2152.980462 lr 3.6e-05
51 Average Loss: 2151.015435 lr 3.5e-05
52 Average Loss: 2151.960279 lr 3.4e-05
53 Average Loss: 2150.031146 lr 3.4e-05
54 Average Loss: 2151.467470 lr 3.3e-05
55 Average Loss: 2150.047150 lr 3.2e-05
56 Average Loss: 2151.157701 lr 3.2e-05
57 Average Loss: 2148.651539 lr 3.1e-05
58 Average Loss: 2150.167031 lr 3.0e-05
59 Average Loss: 2149.078455 lr 3.0e-05
60 Average Loss: 2148.202515 lr 2.9e-05
61 Average Loss: 2147.587889 lr 2.9e-05
62 Average Loss: 2148.330391 lr 2.8e-05
63 Average Loss: 2147.655966 lr 2.7e-05
```

```
64 Average Loss: 2148.102908 lr 2.7e-05
65 Average Loss: 2147.349392 lr 2.6e-05
66 Average Loss: 2147.653178 lr 2.6e-05
67 Average Loss: 2146.939824 lr 2.5e-05
68 Average Loss: 2147.361926 lr 2.5e-05
69 Average Loss: 2146.151646 lr 2.4e-05
70 Average Loss: 2146.377884 lr 2.4e-05
71 Average Loss: 2146.429291 lr 2.3e-05
72 Average Loss: 2146.483938 lr 2.3e-05
73 Average Loss: 2146.402553 lr 2.2e-05
74 Average Loss: 2145.577492 lr 2.2e-05
75 Average Loss: 2145.404566 lr 2.2e-05
76 Average Loss: 2144.415421 lr 2.1e-05
77 Average Loss: 2144.625085 lr 2.1e-05
78 Average Loss: 2144.286437 lr 2.0e-05
79 Average Loss: 2145.178149 lr 2.0e-05
80 Average Loss: 2146.211880 lr 2.0e-05
81 Average Loss: 2144.740868 lr 2.0e-05
82 Average Loss: 2143.449378 lr 2.0e-05
83 Average Loss: 2145.120883 lr 2.0e-05
84 Average Loss: 2145.134621 lr 2.0e-05
85 Average Loss: 2143.722671 lr 2.0e-05
86 Average Loss: 2142.950258 lr 2.0e-05
87 Average Loss: 2143.126643 lr 2.0e-05
88 Average Loss: 2143.619596 lr 2.0e-05
89 Average Loss: 2142.723582 lr 2.0e-05
90 Average Loss: 2142.944223 lr 2.0e-05
91 Average Loss: 2145.270620 lr 2.0e-05
92 Average Loss: 2143.849191 lr 2.0e-05
93 Average Loss: 2142.442079 lr 2.0e-05
94 Average Loss: 2142.579509 lr 2.0e-05
95 Average Loss: 2143.153895 lr 2.0e-05
96 Average Loss: 2141.945950 lr 2.0e-05
97 Average Loss: 2142.753033 lr 2.0e-05
98 Average Loss: 2143.530562 lr 2.0e-05
99 Average Loss: 2141.998980 lr 2.0e-05
100 Average Loss: 2142.886004 lr 2.0e-05
101 Average Loss: 2142.986163 lr 2.0e-05
102 Average Loss: 2142.012143 lr 2.0e-05
103 Average Loss: 2141.552692 lr 2.0e-05
104 Average Loss: 2141.797712 lr 2.0e-05
105 Average Loss: 2141.802507 lr 2.0e-05
106 Average Loss: 2141.877255 lr 2.0e-05
107 Average Loss: 2140.662192 lr 2.0e-05
108 Average Loss: 2140.569687 lr 2.0e-05
109 Average Loss: 2141.078631 lr 2.0e-05
110 Average Loss: 2141.460208 lr 2.0e-05
111 Average Loss: 2141.731946 lr 2.0e-05
112 Average Loss: 2141.170379 lr 2.0e-05
113 Average Loss: 2140.383734 lr 2.0e-05
114 Average Loss: 2141.257002 lr 2.0e-05
115 Average Loss: 2139.860460 lr 2.0e-05
116 Average Loss: 2140.853704 lr 2.0e-05
117 Average Loss: 2139.748139 lr 2.0e-05
118 Average Loss: 2140.346822 lr 2.0e-05
119 Average Loss: 2140.474028 lr 2.0e-05
120 Average Loss: 2140.223845 lr 2.0e-05
121 Average Loss: 2140.922610 lr 2.0e-05
122 Average Loss: 2140.593307 lr 2.0e-05
123 Average Loss: 2140.153832 lr 2.0e-05
```

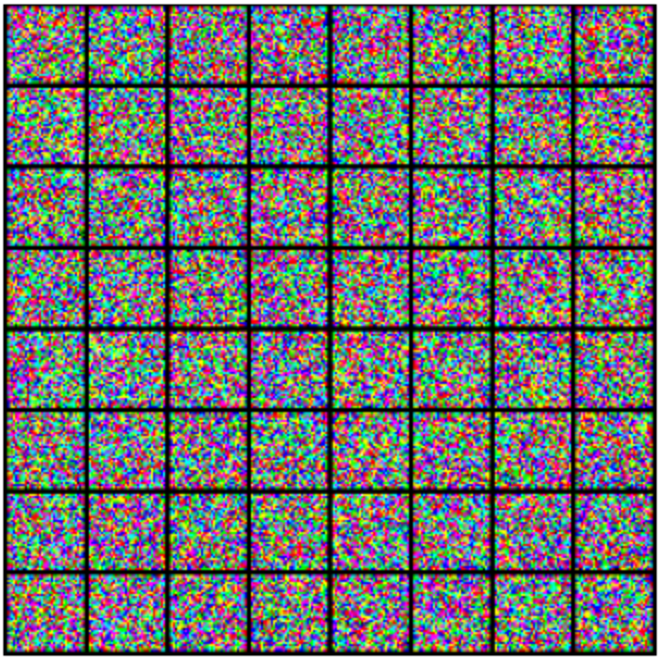
```
124 Average Loss: 2139.767509 lr 2.0e-05
125 Average Loss: 2140.934473 lr 2.0e-05
126 Average Loss: 2139.741909 lr 2.0e-05
127 Average Loss: 2139.852036 lr 2.0e-05
128 Average Loss: 2140.290848 lr 2.0e-05
129 Average Loss: 2139.564766 lr 2.0e-05
130 Average Loss: 2138.432785 lr 2.0e-05
131 Average Loss: 2140.181837 lr 2.0e-05
132 Average Loss: 2140.109730 lr 2.0e-05
133 Average Loss: 2138.761187 lr 2.0e-05
134 Average Loss: 2138.896576 lr 2.0e-05
135 Average Loss: 2139.338224 lr 2.0e-05
136 Average Loss: 2139.239420 lr 2.0e-05
137 Average Loss: 2139.041625 lr 2.0e-05
138 Average Loss: 2138.322051 lr 2.0e-05
139 Average Loss: 2138.237709 lr 2.0e-05
140 Average Loss: 2139.293892 lr 2.0e-05
141 Average Loss: 2139.039397 lr 2.0e-05
142 Average Loss: 2137.252364 lr 2.0e-05
143 Average Loss: 2139.027528 lr 2.0e-05
144 Average Loss: 2138.371124 lr 2.0e-05
145 Average Loss: 2138.199668 lr 2.0e-05
146 Average Loss: 2138.413989 lr 2.0e-05
147 Average Loss: 2138.584425 lr 2.0e-05
148 Average Loss: 2137.113562 lr 2.0e-05
149 Average Loss: 2138.495873 lr 2.0e-05
```

In [35]:

```
# Load the pre-trained checkpoint from disk.
device = 'cuda'

# Load the pre-trained model checkpoint
ckpt = torch.load('ckpt_res.pth', map_location=device)
score_model.load_state_dict(ckpt)

# Set sample batch size and number of steps
sample_batch_size = 64
num_steps = 1000

# Choose the Euler-Maruyama sampler
sampler = Euler_Maruyama_sampler

# Generate samples using the specified sampler
samples = sampler(score_model,
                  marginal_prob_std_fn,
                  diffusion_coeff_fn,
                  sample_batch_size,
                  num_steps=num_steps,
                  device=device,
                  y=None)

# Clip samples to be in the range [0, 1]
samples = samples.clamp(0.0, 1.0)

# Visualize the generated samples
%matplotlib inline
import matplotlib.pyplot as plt
sample_grid = make_grid(samples, nrow=int(np.sqrt(sample_batch_size)))

# Plot the sample grid
```

```
plt.figure(figsize=(6, 6))
plt.axis('off')
plt.imshow(sample_grid.permute(1, 2, 0).cpu(), vmin=0., vmax=1.)
plt.show()
```

```
/tmp/ipykernel_100942/3196970592.py:5: FutureWarning: You are using `torch.load` with `w
eights_only=False` (the current default value), which uses the default pickle module imp
licitly. It is possible to construct malicious pickle data which will execute arbitrary
code during unpickling (See https://github.com/pytorch/pytorch/blob/main/SECURITY.md#unt
rusted-models for more details). In a future release, the default value for `weights_onl
y` will be flipped to `True`. This limits the functions that could be executed during un
pickling. Arbitrary objects will no longer be allowed to be loaded via this mode unless
they are explicitly allowlisted by the user via `torch.serialization.add_safe_globals`.
We recommend you start setting `weights_only=True` for any use case where you don't have
full control of the loaded file. Please open an issue on GitHub for any issues related t
o this experimental feature.
  ckpt = torch.load('ckpt_res.pth', map_location=device)
/tmp/ipykernel_100942/3893175082.py:17: UserWarning: To copy construct from a tensor, it
is recommended to use sourceTensor.clone().detach() or sourceTensor.clone().detach().req
uires_grad_(True), rather than torch.tensor(sourceTensor).
  t = torch.tensor(t, device=device)
/tmp/ipykernel_100942/1259072587.py:16: UserWarning: To copy construct from a tensor, it
is recommended to use sourceTensor.clone().detach() or sourceTensor.clone().detach().req
uires_grad_(True), rather than torch.tensor(sourceTensor).
  return torch.tensor(sigma**t, device=device)
/tmp/ipykernel_100942/890849440.py:25: UserWarning: To copy construct from a tensor, it
is recommended to use sourceTensor.clone().detach() or sourceTensor.clone().detach().req
uires_grad_(True), rather than torch.tensor(sourceTensor).
  x = mean_x + torch.sqrt(torch.tensor(step_size, device=device)) * g[:, None, None, Non
e] * noise
```



In [36]:

```
class CrossAttention(nn.Module):
    def __init__(self, embed_dim, hidden_dim, context_dim=None, num_heads=1):
        """
        Initialize the CrossAttention module.

        Parameters:
        - embed_dim: The dimensionality of the output embeddings.
```

```
        - hidden_dim: The dimensionality of the hidden representations.
        - context_dim: The dimensionality of the context representations (if not self a
        - num_heads: Number of attention heads (currently supports 1 head).

        Note: For simplicity reasons, the implementation assumes 1-head attention.
        Feel free to implement multi-head attention using fancy tensor manipulations.
        """
        super(CrossAttention, self).__init__()

        self.hidden_dim = hidden_dim
        self.context_dim = context_dim
        self.embed_dim = embed_dim


        # Linear layer for query projection
        self.query = nn.Linear(hidden_dim, embed_dim, bias=False)

        # Check if self-attention or cross-attention
        if context_dim is None:
            self.self_attn = True
            self.key = nn.Linear(hidden_dim, embed_dim, bias=False)
            self.value = nn.Linear(hidden_dim, hidden_dim, bias=False)
        else:
            self.self_attn = False
            self.key = nn.Linear(context_dim, embed_dim, bias=False)
            self.value = nn.Linear(context_dim, hidden_dim, bias=False)

    def forward(self, tokens, context=None):
        """
        Forward pass of the CrossAttention module.

        Parameters:
        - tokens: Input tokens with shape [batch, sequence_len, hidden_dim].
        - context: Context information with shape [batch, context_seq_len, context_dim]
                    If self_attn is True, context is ignored.

        Returns:
        - ctx_vecs: Context vectors after attention with shape [batch, sequence_len, em
        """

        if self.self_attn:
            # Self-attention case
            Q = self.query(tokens)
            K = self.key(tokens)
            V = self.value(tokens)
        else:
            # Cross-attention case
            Q = self.query(tokens)
            K = self.key(context)
            V = self.value(context)

        # Compute score matrices, attention matrices, and context vectors
        scoremats = torch.einsum("BTH,BSH->BTS", Q, K)  # Inner product of Q and K, a t
        attnmats = F.softmax(scoremats / math.sqrt(self.embed_dim), dim=-1)  # Softmax
        ctx_vecs = torch.einsum("BTS,BSH->BTH", attnmats, V)  # Weighted average value

        return ctx_vecs
```

In [37]:

```python
class TransformerBlock(nn.Module):
    """The transformer block that combines self-attn, cross-attn, and feed forward neur
    def __init__(self, hidden_dim, context_dim):
        """
        Initialize the TransformerBlock.

        Parameters:
        - hidden_dim: The dimensionality of the hidden state.
        - context_dim: The dimensionality of the context tensor.

        Note: For simplicity, the self-attn and cross-attn use the same hidden_dim.
        """

        super(TransformerBlock, self).__init__()

        # Self-attention module
        self.attn_self = CrossAttention(hidden_dim, hidden_dim)

        # Cross-attention module
        self.attn_cross = CrossAttention(hidden_dim, hidden_dim, context_dim)

        # Layer normalization modules
        self.norm1 = nn.LayerNorm(hidden_dim)
        self.norm2 = nn.LayerNorm(hidden_dim)
        self.norm3 = nn.LayerNorm(hidden_dim)

        # Implement a 2-layer MLP with K * hidden_dim hidden units, and nn.GELU nonline
        self.ffn = nn.Sequential(
            nn.Linear(hidden_dim, 3 * hidden_dim),
            nn.GELU(),
            nn.Linear(3 * hidden_dim, hidden_dim)
        )

    def forward(self, x, context=None):
        """
        Forward pass of the TransformerBlock.

        Parameters:
        - x: Input tensor with shape [batch, sequence_len, hidden_dim].
        - context: Context tensor with shape [batch, context_seq_len, context_dim].

        Returns:
        - x: Output tensor after passing through the TransformerBlock.
        """

        # Apply self-attention with layer normalization and residual connection
        x = self.attn_self(self.norm1(x)) + x

        # Apply cross-attention with layer normalization and residual connection
        x = self.attn_cross(self.norm2(x), context=context) + x

        # Apply feed forward neural network with layer normalization and residual conne
        x = self.ffn(self.norm3(x)) + x

        return x
```

In [38]:
```
class SpatialTransformer(nn.Module):
    def __init__(self, hidden_dim, context_dim):
        """
        Initialize the SpatialTransformer.

        Parameters:
        - hidden_dim: The dimensionality of the hidden state.
        - context_dim: The dimensionality of the context tensor.
        """
        super(SpatialTransformer, self).__init__()

        # TransformerBlock for spatial transformation
        self.transformer = TransformerBlock(hidden_dim, context_dim)

    def forward(self, x, context=None):
        """
        Forward pass of the SpatialTransformer.

        Parameters:
        - x: Input tensor with shape [batch, channels, height, width].
        - context: Context tensor with shape [batch, context_seq_len, context_dim].

        Returns:
        - x: Output tensor after applying spatial transformation.
        """
        b, c, h, w = x.shape
        x_in = x

        # Combine the spatial dimensions and move the channel dimension to the end
        x = rearrange(x, "b c h w -> b (h w) c")

        # Apply the sequence transformer
        x = self.transformer(x, context)

        # Reverse the process
        x = rearrange(x, 'b (h w) c -> b c h w', h=h, w=w)

        # Residue connection
        return x + x_in
```

In [39]:
```
class UNet_Tranformer(nn.Module):
    """A time-dependent score-based model built upon U-Net architecture."""
    def __init__(self, marginal_prob_std, channels=[32, 64, 128, 256], embed_dim=256,
                 text_dim=256, nClass=10):
        """
        Initialize a time-dependent score-based network.

        Parameters:
        - marginal_prob_std: A function that gives the standard deviation
          of the perturbation kernel p_{0t}(x(t) | x(0)).
        - channels: The number of channels for feature maps of each resolution.
        - embed_dim: The dimensionality of Gaussian random feature embeddings of time.
        - text_dim: The embedding dimension of text/digits.
        - nClass: Number of classes to model.
        """
        super().__init__()
```

```
        # Gaussian random feature embedding layer for time
        self.time_embed = nn.Sequential(
            GaussianFourierProjection(embed_dim=embed_dim),
            nn.Linear(embed_dim, embed_dim)
        )

        # Encoding layers where the resolution decreases
        self.conv1 = nn.Conv2d(3, channels[0], kernel_size=3, stride=1, padding=1, bias
        self.dense1 = Dense(embed_dim, channels[0])
        self.gnorm1 = nn.GroupNorm(4, num_channels=channels[0])

        self.conv2 = nn.Conv2d(channels[0], channels[1], kernel_size=3, stride=2, paddi
        self.dense2 = Dense(embed_dim, channels[1])
        self.gnorm2 = nn.GroupNorm(32, num_channels=channels[1])

        self.conv3 = nn.Conv2d(channels[1], channels[2], kernel_size=3, stride=2, paddi
        self.dense3 = Dense(embed_dim, channels[2])
        self.gnorm3 = nn.GroupNorm(32, num_channels=channels[2])
        self.attn3 = SpatialTransformer(channels[2], text_dim)

        self.conv4 = nn.Conv2d(channels[2], channels[3], kernel_size=3, stride=2, paddi
        self.dense4 = Dense(embed_dim, channels[3])
        self.gnorm4 = nn.GroupNorm(32, num_channels=channels[3])
        self.attn4 = SpatialTransformer(channels[3], text_dim)

        # Decoding layers where the resolution increases
        self.tconv4 = nn.ConvTranspose2d(channels[3], channels[2], kernel_size=3, strid
        self.dense5 = Dense(embed_dim, channels[2])
        self.tgnorm4 = nn.GroupNorm(32, num_channels=channels[2])

        self.tconv3 = nn.ConvTranspose2d(channels[2], channels[1], kernel_size=3, strid
        self.dense6 = Dense(embed_dim, channels[1])
        self.tgnorm3 = nn.GroupNorm(32, num_channels=channels[1])

        self.tconv2 = nn.ConvTranspose2d(channels[1], channels[0], kernel_size=3, strid
        self.dense7 = Dense(embed_dim, channels[0])
        self.tgnorm2 = nn.GroupNorm(32, num_channels=channels[0])

        self.tconv1 = nn.ConvTranspose2d(channels[0], 3, kernel_size=3, stride=1, paddi

        # The swish activation function
        self.act = nn.SiLU()
        self.marginal_prob_std = marginal_prob_std
        self.cond_embed = nn.Embedding(nClass, text_dim)

    def forward(self, x, t, y=None):
        """
        Forward pass of the UNet_Transformer model.

        Parameters:
        - x: Input tensor.
        - t: Time tensor.
        - y: Target tensor.

        Returns:
        - h: Output tensor after passing through the UNet_Transformer architecture.
        """
        # Obtain the Gaussian random feature embedding for t
        embed = self.act(self.time_embed(t))
        y_embed = self.cond_embed(y).unsqueeze(1)
```

```
        # Encoding path
        h1 = self.conv1(x) + self.dense1(embed)
        h1 = self.act(self.gnorm1(h1))
        h2 = self.conv2(h1) + self.dense2(embed)
        h2 = self.act(self.gnorm2(h2))
        h3 = self.conv3(h2) + self.dense3(embed)
        h3 = self.act(self.gnorm3(h3))
        h3 = self.attn3(h3, y_embed)
        h4 = self.conv4(h3) + self.dense4(embed)
        h4 = self.act(self.gnorm4(h4))
        h4 = self.attn4(h4, y_embed)

        # Decoding path
        h = self.tconv4(h4) + self.dense5(embed)
        h = self.act(self.tgnorm4(h))
        h = self.tconv3(h + h3) + self.dense6(embed)
        h = self.act(self.tgnorm3(h))
        h = self.tconv2(h + h2) + self.dense7(embed)
        h = self.act(self.tgnorm2(h))
        h = self.tconv1(h + h1)

        # Normalize output
        h = h / self.marginal_prob_std(t)[:, None, None, None]
        return h
```

In [40]:

```
def loss_fn_cond(model, x, y, marginal_prob_std, eps=1e-5):
    """
    The loss function for training score-based generative models with conditional infor

    Parameters:
    - model: A PyTorch model instance that represents a time-dependent score-based mode
    - x: A mini-batch of training data.
    - y: Conditional information (target tensor).
    - marginal_prob_std: A function that gives the standard deviation of the perturbati
    - eps: A tolerance value for numerical stability.

    Returns:
    - loss: The calculated loss.
    """
    # Sample time uniformly in the range [eps, 1-eps]
    random_t = torch.rand(x.shape[0], device=x.device) * (1. - eps) + eps
    # Generate random noise with the same shape as the input
    z = torch.randn_like(x)
    # Compute the standard deviation of the perturbation kernel at the sampled time
    std = marginal_prob_std(random_t)
    # Perturb the input data with the generated noise and scaled by the standard deviat
    perturbed_x = x + z * std[:, None, None, None]
    # Get the model's score for the perturbed input, considering conditional informatio
    score = model(perturbed_x, random_t, y=y)
    # Calculate the loss using the score and perturbation
    loss = torch.mean(torch.sum((score * std[:, None, None, None] + z)**2, dim=(1, 2, 3
    return loss
```

In [41]:

```
# Specify whether to continue training or initialize a new model

continue_training = False # Either True or False
```

```
if not continue_training:

    # Initialize a new UNet with Transformer model
    score_model = torch.nn.DataParallel(UNet_Tranformer(marginal_prob_std=marginal_prob
    score_model = score_model.to(device)

# Set training hyperparameters
n_epochs =   200   #{'type':'integer'}
batch_size =  256 #{'type':'integer'}
lr = 1e-4         #{'type':'number'}

# Load the CIFAR10 dataset and create a data loader
dataset = CIFAR10('.', train=True, transform=transforms.Compose([transforms.Resize((32,
data_loader = DataLoader(dataset, batch_size=batch_size, shuffle=True, num_workers=4)

# Define the optimizer and learning rate scheduler
optimizer = Adam(score_model.parameters(), lr=lr)
scheduler = LambdaLR(optimizer, lr_lambda=lambda epoch: max(0.2, 0.98 ** epoch))

# Use tqdm to display a progress bar over epochs
tqdm_epoch = trange(n_epochs)
for epoch in tqdm_epoch:
    avg_loss = 0.
    num_items = 0

    # Iterate over batches in the data loader
    for x, y in tqdm(data_loader):
        #x = x.to(device)
        x, y = x.to(device), y.to(device)

        # Compute the loss using the conditional score-based model
        loss = loss_fn_cond(score_model, x, y, marginal_prob_std_fn)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        avg_loss += loss.item() * x.shape[0]
        num_items += x.shape[0]

    # Adjust learning rate using the scheduler
    scheduler.step()
    lr_current = scheduler.get_last_lr()[0]

    # Print epoch information including average loss and current learning rate
    print('{} Average Loss: {:5f} lr {:.1e}'.format(epoch, avg_loss / num_items, lr_cur
    tqdm_epoch.set_description('Average Loss: {:5f}'.format(avg_loss / num_items))

    # Save the model checkpoint after each epoch of training
    torch.save(score_model.state_dict(), 'ckpt_transformer.pth')
```

```
Files already downloaded and verified


/tmp/ipykernel_100942/3893175082.py:17: UserWarning: To copy construct from a tensor, it
is recommended to use sourceTensor.clone().detach() or sourceTensor.clone().detach().req
uires_grad_(True), rather than torch.tensor(sourceTensor).
  t = torch.tensor(t, device=device)
0 Average Loss: 3306.254765 lr 9.8e-05

1 Average Loss: 1123.250744 lr 9.6e-05

2 Average Loss: 775.076693 lr 9.4e-05
```

```
3 Average Loss: 605.974069 lr 9.2e-05
4 Average Loss: 503.167865 lr 9.0e-05
5 Average Loss: 426.989081 lr 8.9e-05
6 Average Loss: 374.637452 lr 8.7e-05
7 Average Loss: 333.472011 lr 8.5e-05
8 Average Loss: 302.695736 lr 8.3e-05
9 Average Loss: 277.525446 lr 8.2e-05
10 Average Loss: 254.193762 lr 8.0e-05
11 Average Loss: 238.033515 lr 7.8e-05
12 Average Loss: 222.530208 lr 7.7e-05
13 Average Loss: 210.842423 lr 7.5e-05
14 Average Loss: 198.491607 lr 7.4e-05
15 Average Loss: 189.396689 lr 7.2e-05
16 Average Loss: 182.912221 lr 7.1e-05
17 Average Loss: 176.501906 lr 7.0e-05
18 Average Loss: 171.135163 lr 6.8e-05
19 Average Loss: 166.636694 lr 6.7e-05
20 Average Loss: 162.168355 lr 6.5e-05
21 Average Loss: 159.193941 lr 6.4e-05
22 Average Loss: 157.263784 lr 6.3e-05
23 Average Loss: 154.428925 lr 6.2e-05
24 Average Loss: 152.917524 lr 6.0e-05
25 Average Loss: 150.678608 lr 5.9e-05
26 Average Loss: 150.356289 lr 5.8e-05
27 Average Loss: 147.684114 lr 5.7e-05
28 Average Loss: 146.925079 lr 5.6e-05
29 Average Loss: 146.704910 lr 5.5e-05
30 Average Loss: 143.309815 lr 5.3e-05
31 Average Loss: 144.464329 lr 5.2e-05
32 Average Loss: 142.022520 lr 5.1e-05
33 Average Loss: 142.892888 lr 5.0e-05
34 Average Loss: 142.306742 lr 4.9e-05
35 Average Loss: 139.968249 lr 4.8e-05
36 Average Loss: 137.531110 lr 4.7e-05
```

```
37 Average Loss: 138.357244 lr 4.6e-05
38 Average Loss: 139.235764 lr 4.5e-05
39 Average Loss: 136.093122 lr 4.5e-05
40 Average Loss: 135.606605 lr 4.4e-05
41 Average Loss: 136.010208 lr 4.3e-05
42 Average Loss: 134.232050 lr 4.2e-05
43 Average Loss: 135.876279 lr 4.1e-05
44 Average Loss: 135.089783 lr 4.0e-05
45 Average Loss: 134.187901 lr 3.9e-05
46 Average Loss: 132.844327 lr 3.9e-05
47 Average Loss: 131.406348 lr 3.8e-05
48 Average Loss: 130.815617 lr 3.7e-05
49 Average Loss: 131.928891 lr 3.6e-05
50 Average Loss: 131.175908 lr 3.6e-05
51 Average Loss: 132.373528 lr 3.5e-05
52 Average Loss: 131.901112 lr 3.4e-05
53 Average Loss: 130.444340 lr 3.4e-05
54 Average Loss: 131.455150 lr 3.3e-05
55 Average Loss: 129.681240 lr 3.2e-05
56 Average Loss: 128.974952 lr 3.2e-05
57 Average Loss: 128.492457 lr 3.1e-05
58 Average Loss: 129.383904 lr 3.0e-05
59 Average Loss: 128.091236 lr 3.0e-05
60 Average Loss: 127.640297 lr 2.9e-05
61 Average Loss: 127.741445 lr 2.9e-05
62 Average Loss: 126.232855 lr 2.8e-05
63 Average Loss: 126.511522 lr 2.7e-05
64 Average Loss: 128.075099 lr 2.7e-05
65 Average Loss: 125.853330 lr 2.6e-05
66 Average Loss: 128.219130 lr 2.6e-05
67 Average Loss: 127.148509 lr 2.5e-05
68 Average Loss: 125.554953 lr 2.5e-05
69 Average Loss: 126.169059 lr 2.4e-05
70 Average Loss: 126.681661 lr 2.4e-05
71 Average Loss: 125.687875 lr 2.3e-05
```

```
72 Average Loss: 125.561705 lr 2.3e-05
73 Average Loss: 126.758951 lr 2.2e-05
74 Average Loss: 127.242826 lr 2.2e-05
75 Average Loss: 124.862177 lr 2.2e-05
76 Average Loss: 125.227223 lr 2.1e-05
77 Average Loss: 125.326246 lr 2.1e-05
78 Average Loss: 125.811814 lr 2.0e-05
79 Average Loss: 125.430572 lr 2.0e-05
80 Average Loss: 123.787792 lr 2.0e-05
81 Average Loss: 125.000441 lr 2.0e-05
82 Average Loss: 123.694654 lr 2.0e-05
83 Average Loss: 123.445283 lr 2.0e-05
84 Average Loss: 124.960931 lr 2.0e-05
85 Average Loss: 123.415156 lr 2.0e-05
86 Average Loss: 123.184263 lr 2.0e-05
87 Average Loss: 123.654332 lr 2.0e-05
88 Average Loss: 122.801168 lr 2.0e-05
89 Average Loss: 122.830142 lr 2.0e-05
90 Average Loss: 122.604704 lr 2.0e-05
91 Average Loss: 121.785714 lr 2.0e-05
92 Average Loss: 124.157227 lr 2.0e-05
93 Average Loss: 122.138739 lr 2.0e-05
94 Average Loss: 122.431965 lr 2.0e-05
95 Average Loss: 121.952033 lr 2.0e-05
96 Average Loss: 122.174926 lr 2.0e-05
97 Average Loss: 122.696816 lr 2.0e-05
98 Average Loss: 122.236433 lr 2.0e-05
99 Average Loss: 123.535536 lr 2.0e-05
100 Average Loss: 122.303800 lr 2.0e-05
101 Average Loss: 121.036578 lr 2.0e-05
102 Average Loss: 121.731702 lr 2.0e-05
103 Average Loss: 122.400558 lr 2.0e-05
104 Average Loss: 121.173569 lr 2.0e-05
105 Average Loss: 121.022060 lr 2.0e-05
```

```
106 Average Loss: 121.482933 lr 2.0e-05
107 Average Loss: 122.086435 lr 2.0e-05
108 Average Loss: 120.961228 lr 2.0e-05
109 Average Loss: 121.902642 lr 2.0e-05
110 Average Loss: 120.587761 lr 2.0e-05
111 Average Loss: 121.486073 lr 2.0e-05
112 Average Loss: 120.106644 lr 2.0e-05
113 Average Loss: 120.874357 lr 2.0e-05
114 Average Loss: 118.027891 lr 2.0e-05
115 Average Loss: 120.373360 lr 2.0e-05
116 Average Loss: 120.602858 lr 2.0e-05
117 Average Loss: 119.934267 lr 2.0e-05
118 Average Loss: 119.951023 lr 2.0e-05
119 Average Loss: 121.055866 lr 2.0e-05
120 Average Loss: 118.960286 lr 2.0e-05
121 Average Loss: 120.818119 lr 2.0e-05
122 Average Loss: 121.495364 lr 2.0e-05
123 Average Loss: 120.747039 lr 2.0e-05
124 Average Loss: 120.071077 lr 2.0e-05
125 Average Loss: 120.101903 lr 2.0e-05
126 Average Loss: 119.574408 lr 2.0e-05
127 Average Loss: 120.973085 lr 2.0e-05
128 Average Loss: 120.259735 lr 2.0e-05
129 Average Loss: 119.769961 lr 2.0e-05
130 Average Loss: 119.557918 lr 2.0e-05
131 Average Loss: 118.727809 lr 2.0e-05
132 Average Loss: 118.341303 lr 2.0e-05
133 Average Loss: 119.630873 lr 2.0e-05
134 Average Loss: 119.001967 lr 2.0e-05
135 Average Loss: 120.103848 lr 2.0e-05
136 Average Loss: 118.161987 lr 2.0e-05
137 Average Loss: 118.400670 lr 2.0e-05
138 Average Loss: 119.530594 lr 2.0e-05
139 Average Loss: 118.552226 lr 2.0e-05
140 Average Loss: 119.880400 lr 2.0e-05
```

```
141 Average Loss: 119.857876 lr 2.0e-05
142 Average Loss: 118.563280 lr 2.0e-05
143 Average Loss: 118.481091 lr 2.0e-05
144 Average Loss: 118.563764 lr 2.0e-05
145 Average Loss: 118.347543 lr 2.0e-05
146 Average Loss: 119.284935 lr 2.0e-05
147 Average Loss: 117.352046 lr 2.0e-05
148 Average Loss: 117.916845 lr 2.0e-05
149 Average Loss: 118.030389 lr 2.0e-05
150 Average Loss: 118.309390 lr 2.0e-05
151 Average Loss: 118.905597 lr 2.0e-05
152 Average Loss: 117.874200 lr 2.0e-05
153 Average Loss: 117.323180 lr 2.0e-05
154 Average Loss: 117.270847 lr 2.0e-05
155 Average Loss: 117.962676 lr 2.0e-05
156 Average Loss: 118.421931 lr 2.0e-05
157 Average Loss: 119.876932 lr 2.0e-05
158 Average Loss: 118.908172 lr 2.0e-05
159 Average Loss: 116.431527 lr 2.0e-05
160 Average Loss: 117.927555 lr 2.0e-05
161 Average Loss: 118.428078 lr 2.0e-05
162 Average Loss: 116.859965 lr 2.0e-05
163 Average Loss: 118.011658 lr 2.0e-05
164 Average Loss: 117.126529 lr 2.0e-05
165 Average Loss: 116.756927 lr 2.0e-05
166 Average Loss: 116.638563 lr 2.0e-05
167 Average Loss: 116.054738 lr 2.0e-05
168 Average Loss: 118.550724 lr 2.0e-05
169 Average Loss: 116.006649 lr 2.0e-05
170 Average Loss: 118.339910 lr 2.0e-05
171 Average Loss: 115.609484 lr 2.0e-05
172 Average Loss: 116.248252 lr 2.0e-05
173 Average Loss: 117.335856 lr 2.0e-05
174 Average Loss: 116.002338 lr 2.0e-05
```

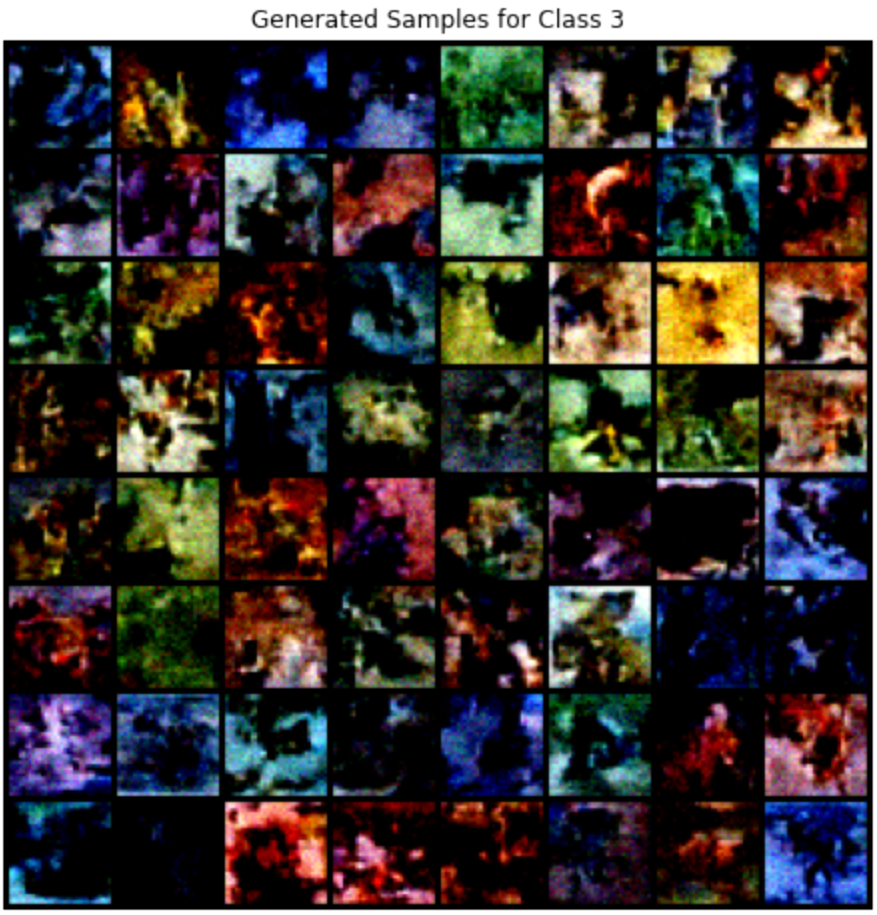
```
175 Average Loss: 116.973482 lr 2.0e-05

176 Average Loss: 116.071740 lr 2.0e-05

177 Average Loss: 117.073110 lr 2.0e-05

178 Average Loss: 116.247582 lr 2.0e-05

179 Average Loss: 117.801115 lr 2.0e-05

180 Average Loss: 115.474765 lr 2.0e-05

181 Average Loss: 115.542122 lr 2.0e-05

182 Average Loss: 115.965238 lr 2.0e-05

183 Average Loss: 116.721344 lr 2.0e-05

184 Average Loss: 116.423777 lr 2.0e-05

185 Average Loss: 116.865593 lr 2.0e-05

186 Average Loss: 115.027593 lr 2.0e-05

187 Average Loss: 115.653050 lr 2.0e-05

188 Average Loss: 117.126095 lr 2.0e-05

189 Average Loss: 115.374482 lr 2.0e-05

190 Average Loss: 116.864594 lr 2.0e-05

191 Average Loss: 115.610969 lr 2.0e-05

192 Average Loss: 116.671183 lr 2.0e-05

193 Average Loss: 115.329164 lr 2.0e-05

194 Average Loss: 115.637890 lr 2.0e-05

195 Average Loss: 117.170060 lr 2.0e-05

196 Average Loss: 116.409412 lr 2.0e-05

197 Average Loss: 114.593106 lr 2.0e-05

198 Average Loss: 114.789068 lr 2.0e-05

199 Average Loss: 115.624652 lr 2.0e-05
```

In [42]:

```
# Load model and generate samples
score_model = torch.nn.DataParallel(UNet_Tranformer(marginal_prob_std=marginal_prob_std
ckpt = torch.load('ckpt_transformer.pth', map_location=device)
score_model.load_state_dict(ckpt)
score_model.eval()

target_class = 3
samples = Euler_Maruyama_sampler(
    score_model,
    marginal_prob_std_fn,
    diffusion_coeff_fn,
    batch_size=64,
    x_shape=(3, 32, 32),
    num_steps=250,
    device=device,
    y=target_class * torch.ones(64, dtype=torch.long, device=device)
```

```
)

# Visualize samples
samples = samples.clamp(0.0, 1.0)
sample_grid = make_grid(samples, nrow=8)
plt.figure(figsize=(8, 8))
plt.axis('off')
plt.imshow(sample_grid.permute(1, 2, 0).cpu(), vmin=0., vmax=1.)
plt.title(f"Generated Samples for Class {target_class}")
plt.show()
```

```
/tmp/ipykernel_100942/143424198.py:3: FutureWarning: You are using `torch.load` with `we
ights_only=False` (the current default value), which uses the default pickle module impl
icitly. It is possible to construct malicious pickle data which will execute arbitrary c
ode during unpickling (See https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untr
usted-models for more details). In a future release, the default value for `weights_only
` will be flipped to `True`. This limits the functions that could be executed during unp
ickling. Arbitrary objects will no longer be allowed to be loaded via this mode unless t
hey are explicitly allowlisted by the user via `torch.serialization.add_safe_globals`. W
e recommend you start setting `weights_only=True` for any use case where you don't have
full control of the loaded file. Please open an issue on GitHub for any issues related t
o this experimental feature.
  ckpt = torch.load('ckpt_transformer.pth', map_location=device)
/tmp/ipykernel_100942/3893175082.py:17: UserWarning: To copy construct from a tensor, it
is recommended to use sourceTensor.clone().detach() or sourceTensor.clone().detach().req
uires_grad_(True), rather than torch.tensor(sourceTensor).
  t = torch.tensor(t, device=device)
/tmp/ipykernel_100942/1259072587.py:16: UserWarning: To copy construct from a tensor, it
is recommended to use sourceTensor.clone().detach() or sourceTensor.clone().detach().req
uires_grad_(True), rather than torch.tensor(sourceTensor).
  return torch.tensor(sigma**t, device=device)
/tmp/ipykernel_100942/890849440.py:25: UserWarning: To copy construct from a tensor, it
is recommended to use sourceTensor.clone().detach() or sourceTensor.clone().detach().req
uires_grad_(True), rather than torch.tensor(sourceTensor).
  x = mean_x + torch.sqrt(torch.tensor(step_size, device=device)) * g[:, None, None, Non
e] * noise
```


Generated Samples for Class 3