Lab 2, Training

In [1]:
```
from typing import Dict, Tuple
from tqdm import tqdm
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader
from torchvision import models, transforms
from torchvision.utils import save_image, make_grid
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation, PillowWriter
import numpy as np
from IPython.display import HTML
from diffusion_utilities import *
```

Setting Things Up

In [2]:
```
class ContextUnet(nn.Module):
    def __init__(self, in_channels, n_feat=256, n_cfeat=10, heigh
        super(ContextUnet, self).__init__()

        # number of input channels, number of intermediate featu
        self.in_channels = in_channels
        self.n_feat = n_feat
        self.n_cfeat = n_cfeat
        self.h = height  #assume h == w. must be divisible by 4,

        # Initialize the initial convolutional layer
        self.init_conv = ResidualConvBlock(in_channels, n_feat, i

        # Initialize the down-sampling path of the U-Net with tw
        self.down1 = UnetDown(n_feat, n_feat)        # down1 #[1
        self.down2 = UnetDown(n_feat, 2 * n_feat)    # down2 #[1

         # original: self.to_vec = nn.Sequential(nn.AvgPool2d(7),
        self.to_vec = nn.Sequential(nn.AvgPool2d((4)), nn.GELU())

        # Embed the timestep and context labels with a one-layer
        self.timeembed1 = EmbedFC(1, 2*n_feat)
        self.timeembed2 = EmbedFC(1, 1*n_feat)
        self.contextembed1 = EmbedFC(n_cfeat, 2*n_feat)
        self.contextembed2 = EmbedFC(n_cfeat, 1*n_feat)

        # Initialize the up-sampling path of the U-Net with three
        self.up0 = nn.Sequential(
            nn.ConvTranspose2d(2 * n_feat, 2 * n_feat, self.h//4,
            nn.GroupNorm(8, 2 * n_feat), # normalize
            nn.ReLU(),
        )
        self.up1 = UnetUp(4 * n_feat, n_feat)
        self.up2 = UnetUp(2 * n_feat, n_feat)

        # Initialize the final convolutional layers to map to the
        self.out = nn.Sequential(
```

```
            nn.Conv2d(2 * n_feat, n_feat, 3, 1, 1), # reduce numb
            nn.GroupNorm(8, n_feat), # normalize
            nn.ReLU(),
            nn.Conv2d(n_feat, self.in_channels, 3, 1, 1), # map t
        )

    def forward(self, x, t, c=None):
        """
        x : (batch, n_feat, h, w) : input image
        t : (batch, n_cfeat)      : time step
        c : (batch, n_classes)    : context label
        """
        # x is the input image, c is the context label, t is the

        # pass the input image through the initial convolutional
        x = self.init_conv(x)
        # pass the result through the down-sampling path
        down1 = self.down1(x)       #[10, 256, 8, 8]
        down2 = self.down2(down1)   #[10, 256, 4, 4]

        # convert the feature maps to a vector and apply an activ
        hiddenvec = self.to_vec(down2)

        # mask out context if context_mask == 1
        if c is None:
            c = torch.zeros(x.shape[0], self.n_cfeat).to(x)

        # embed context and timestep
        cemb1 = self.contextembed1(c).view(-1, self.n_feat * 2, 1
        temb1 = self.timeembed1(t).view(-1, self.n_feat * 2, 1, 1
        cemb2 = self.contextembed2(c).view(-1, self.n_feat, 1, 1)
        temb2 = self.timeembed2(t).view(-1, self.n_feat, 1, 1)
        #print(f"uunet forward: cemb1 {cemb1.shape}. temb1 {temb1


        up1 = self.up0(hiddenvec)
        up2 = self.up1(cemb1*up1 + temb1, down2)  # add and multi
        up3 = self.up2(cemb2*up2 + temb2, down1)
        out = self.out(torch.cat((up3, x), 1))
        return out
```

In [3]:
```
# hyperparameters

# diffusion hyperparameters
timesteps = 500
beta1 = 1e-4
beta2 = 0.02

# network hyperparameters
device = torch.device("cuda:0" if torch.cuda.is_available() else
n_feat = 64 # 64 hidden dimension feature
n_cfeat = 5 # context vector is of size 5
height = 16 # 16x16 image
save_dir = './weights/'

# training hyperparameters
batch_size = 100
n_epoch = 32
lrate=1e-3
```

In [4]:
```
# construct DDPM noise schedule
b_t = (beta2 - beta1) * torch.linspace(0, 1, timesteps + 1, devic
a_t = 1 - b_t
ab_t = torch.cumsum(a_t.log(), dim=0).exp()
ab_t[0] = 1
```

In [5]:
```
# construct model
nn_model = ContextUnet(in_channels=3, n_feat=n_feat, n_cfeat=n_cf
```

Training

In [6]:
```
# load dataset and construct optimizer
dataset = CustomDataset("./sprites_1788_16x16.npy", "./sprite_lab
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=T
optim = torch.optim.Adam(nn_model.parameters(), lr=lrate)
```

```
sprite shape: (89400, 16, 16, 3)
labels shape: (89400, 5)
```

In [7]:
```
# helper function: perturbs an image to a specified noise level
def perturb_input(x, t, noise):
    return ab_t.sqrt()[t, None, None, None] * x + (1 - ab_t[t, No
```

This code will take hours to run on a CPU. We recommend you skip this step here and check the intermediate results below.

If you decide to try it, you could download to your own machine. Be sure to change the cell type. Note, the CPU run time in the course is limited so you will not be able to fully train the network using the class platforr

```
# training without context code

# set into train mode
nn_model.train()

for ep in range(n_epoch):
    print(f'epoch {ep}')

    # linearly decay learning rate
```

```
    optim.param_groups[0]['lr'] = lrate*(1-ep/n_epoch)

    pbar = tqdm(dataloader, mininterval=2 )
    for x, _ in pbar:   # x: images
        optim.zero_grad()
        x = x.to(device)

        # perturb data
        noise = torch.randn_like(x)
        t = torch.randint(1, timesteps + 1, (x.shape[0],)).to(device)
        x_pert = perturb_input(x, t, noise)

        # use network to recover noise
        pred_noise = nn_model(x_pert, t / timesteps)

        # loss is mean squared error between the predicted and true noise
        loss = F.mse_loss(pred_noise, noise)
        loss.backward()

        optim.step()

    # save model periodically
    if ep%4==0 or ep == int(n_epoch-1):
        if not os.path.exists(save_dir):
            os.mkdir(save_dir)
        torch.save(nn_model.state_dict(), save_dir + f"model_{ep}.pth")
        print('saved model at ' + save_dir + f"model_{ep}.pth")
```

Sampling

In [8]:

```
# helper function; removes the predicted noise (but adds some noi
def denoise_add_noise(x, t, pred_noise, z=None):
    if z is None:
        z = torch.randn_like(x)
    noise = b_t.sqrt()[t] * z
    mean = (x - pred_noise * ((1 - a_t[t]) / (1 - ab_t[t]).sqrt()
    return mean + noise
```

In [9]:

```
# sample using standard algorithm
@torch.no_grad()
def sample_ddpm(n_sample, save_rate=20):
    # x_T ~ N(0, 1), sample initial noise
    samples = torch.randn(n_sample, 3, height, height).to(device)

    # array to keep track of generated steps for plotting
    intermediate = []
    for i in range(timesteps, 0, -1):
        print(f'sampling timestep {i:3d}', end='\r')

        # reshape time tensor
        t = torch.tensor([i / timesteps])[:, None, None, None].to

        # sample some random noise to inject back in. For i = 1,
        z = torch.randn_like(samples) if i > 1 else 0

        eps = nn_model(samples, t)    # predict noise e_(x_t,t)
        samples = denoise_add_noise(samples, i, eps, z)
        if i % save_rate ==0 or i==timesteps or i<8:
            intermediate.append(samples.detach().cpu().numpy())
```

```
    intermediate = np.stack(intermediate)
    return samples, intermediate
```

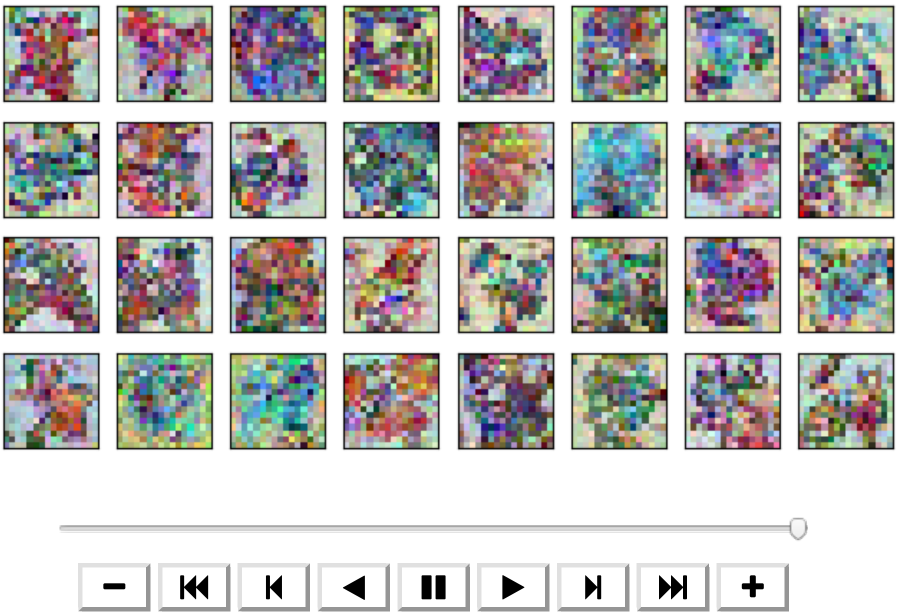
View Epoch 0

In [10]:
```
# load in model weights and set to eval mode
nn_model.load_state_dict(torch.load(f"{save_dir}/model_0.pth", ma
nn_model.eval()
print("Loaded in Model")
```

```
Loaded in Model
```

In [*]:
```
# visualize samples
plt.clf()
samples, intermediate_ddpm = sample_ddpm(32)
animation_ddpm = plot_sample(intermediate_ddpm,32,4,save_dir, "ar
HTML(animation_ddpm.to_jshtml())
```

```
gif animating frame 31 of 32
```



○ Once ● Loop ○ Reflect

View Epoch 4

In [*]:
```
# load in model weights and set to eval mode
nn_model.load_state_dict(torch.load(f"{save_dir}/model_4.pth", ma
nn_model.eval()
print("Loaded in Model")
```

In [13]:
```
# visualize samples
plt.clf()
samples, intermediate_ddpm = sample_ddpm(32)
animation_ddpm = plot_sample(intermediate_ddpm,32,4,save_dir, "ar
HTML(animation_ddpm.to_jshtml())
```

```
gif animating frame 31 of 32
```

Once Loop Reflect

```
<Figure size 640x480 with 0 Axes>
```

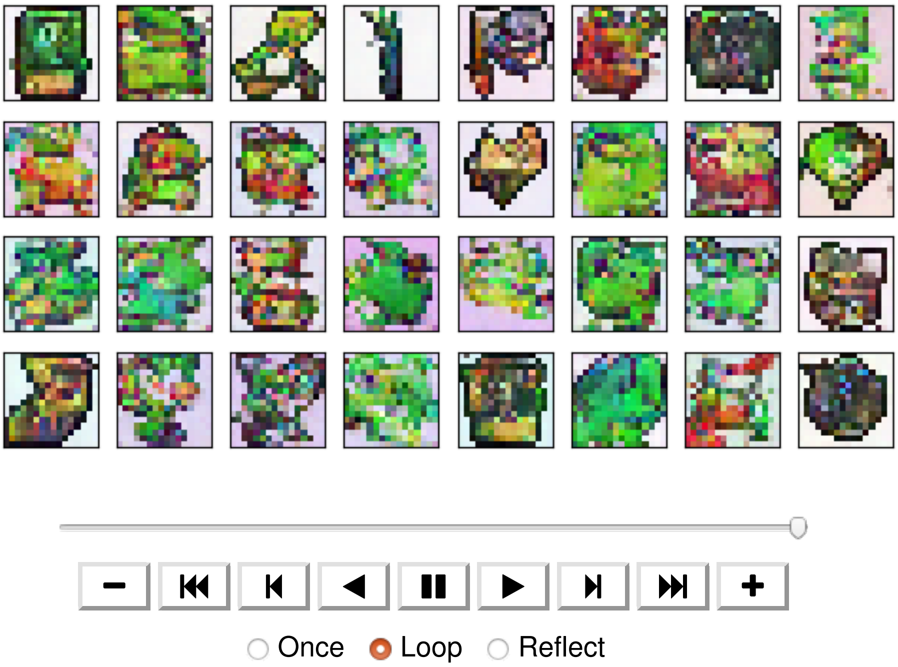
View Epoch 8

In [14]:
```
# load in model weights and set to eval mode
nn_model.load_state_dict(torch.load(f"{save_dir}/model_8.pth", ma
nn_model.eval()
print("Loaded in Model")
```

```
Loaded in Model
```

In [*]:
```
# visualize samples
plt.clf()
samples, intermediate_ddpm = sample_ddpm(32)
animation_ddpm = plot_sample(intermediate_ddpm,32,4,save_dir, "ar
HTML(animation_ddpm.to_jshtml())
```

gif animating frame 31 of 32



<Figure size 640x480 with 0 Axes>

View Epoch 31

In [*]:
```
# load in model weights and set to eval mode
nn_model.load_state_dict(torch.load(f"{save_dir}/model_31.pth", m
nn_model.eval()
print("Loaded in Model")
```

Loaded in Model

In [*]:
```
# visualize samples
plt.clf()
samples, intermediate_ddpm = sample_ddpm(32)
animation_ddpm = plot_sample(intermediate_ddpm,32,4,save_dir, "ar
HTML(animation_ddpm.to_jshtml())
```

```
gif animating frame 31 of 32
```

Once Loop Reflect

```
&lt;Figure size 640x480 with 0 Axes&gt;
```


Acknowledgments

Sprites by ElvGames, FrootsnVeggies (https://zrghr.itch.io/froots-and-veggies-culinary-pixels) and kyrise (https://kyrise.itch.io/)
This code is modified from, https://github.com/cloneofsimo/minDiffusion (https://github.com/cloneofsimo/minDiffusion)
Diffusion model is based on Denoising Diffusion Probabilistic Models (https://arxiv.org/abs/2006.11239) and Denoising Diffusion Implicit Models (https://arxiv.org/abs/2010.02502)

In [ ]: