In [2]:

```python
import numpy as np
import pandas as pd
from collections import OrderedDict
import torch
from torch import nn, optim
from torchvision import datasets, transforms, utils, models
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IPython.display import HTML
from PIL import Image
```

In [3]:

```python
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Using device: {device.type}")
```

In [4]:

```python
# Set up data
DATA_DIR = "../input/10-monkey-species/training/training"
IMAGE_SIZE = (128, 128)
BATCH_SIZE = 32

data_transforms = transforms.Compose([
    transforms.Resize(IMAGE_SIZE),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

dataset = datasets.ImageFolder(root=DATA_DIR, transform=data_transforms)

data_loader = torch.utils.data.DataLoader(
    dataset, batch_size=BATCH_SIZE, shuffle=True,
#     sampler=train_sampler
)

# Plot samples
sample_batch = next(iter(data_loader))
plt.figure(figsize=(10, 8)); plt.axis("off"); plt.title("Sample Training Images")
plt.imshow(np.transpose(utils.make_grid(sample_batch[0], padding=1, normalize=True), (1, 2

len(data_loader) * BATCH_SIZE
```

In [5]:

```python
#create generator for GAN
class Generator(nn.Module):

    def __init__(self, LATENT_SIZE):
        super(Generator, self).__init__()

        self.main = nn.Sequential(

            # input dim: [-1, LATENT_SIZE, 1, 1]

            nn.ConvTranspose2d(LATENT_SIZE, 1024, kernel_size=4, stride=1, padding=0, bias
            nn.BatchNorm2d(1024),
            nn.LeakyReLU(0.2, inplace=True),

            # output dim: [-1, 1024, 4, 4]

            nn.ConvTranspose2d(1024, 1024, kernel_size=4, stride=2, padding=1, bias=False)
            nn.BatchNorm2d(1024),
            nn.LeakyReLU(0.2, inplace=True),

            # output dim: [-1, 1024, 8, 8]
```

```python
            nn.ConvTranspose2d(1024, 512, kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(512),
            nn.LeakyReLU(0.2, inplace=True),

            # output dim: [-1, 512, 16, 16]

            nn.ConvTranspose2d(512, 128, kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2, inplace=True),

            # output dim: [-1, 128, 32, 32]

            nn.ConvTranspose2d(128, 64, kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.2, inplace=True),

            # output dim: [-1, 64, 64, 64]

            nn.ConvTranspose2d(64, 3, kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(3),

            # output dim: [-1, 3, 128, 128]

            nn.Tanh()

            # output dim: [-1, 3, 128, 128]
        )

    def forward(self, input):
        output = self.main(input)
        return output
```

In [6]:

```python
#create discriminator for GAN
class Discriminator(nn.Module):

    def __init__(self):
        super(Discriminator, self).__init__()

        self.main = nn.Sequential(

            # input dim: [-1, 3, 128, 128]

            nn.Conv2d(3, 64, kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.2, inplace=True),

            # output dim: [-1, 64, 64, 64]

            nn.Conv2d(64, 64, kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.2, inplace=True),

            # output dim: [-1, 64, 32, 32]

            nn.Conv2d(64, 64, kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.2, inplace=True),

            # output dim: [-1, 128, 16, 16]

            nn.Conv2d(64, 64, kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.2, inplace=True),
```

```python
            # output dim: [-1, 256, 8, 8]

            nn.Conv2d(64, 64, kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.2, inplace=True),

            # output dim: [-1, 512, 4, 4]

            nn.Conv2d(64, 1, kernel_size=4, stride=1, padding=0),

            # output dim: [-1, 1, 1, 1]

            nn.Flatten(),

            # output dim: [-1]

            nn.Sigmoid()

            # output dim: [-1]
        )

    def forward(self, input):
        output = self.main(input)
        return output
```

In [8]:

```python
#initialize the weights
def weights_init(m):
    if isinstance(m, (nn.Conv2d, nn.ConvTranspose2d)):
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif isinstance(m, nn.BatchNorm2d):
        nn.init.normal_(m.weight.data, 1.0, 0.02)
        nn.init.constant_(m.bias.data, 0)

LATENT_SIZE = 50
LR = 0.001

generator = Generator(LATENT_SIZE)
generator.apply(weights_init)
generator.to(device)
discriminator = Discriminator()
discriminator.apply(weights_init)
discriminator.to(device);

criterion = nn.BCELoss()
optimizerG = optim.Adam(generator.parameters(), lr=LR, betas=(0.5, 0.999))
optimizerD = optim.Adam(discriminator.parameters(), lr=LR, betas=(0.5, 0.999))
fixed_noise = torch.randn(BATCH_SIZE, LATENT_SIZE, 1, 1, device=device)
```

In [15]:

```python
from statistics import mean
#train the GAN model
img_list = []
D_real_epoch, D_fake_epoch, loss_dis_epoch, loss_gen_epoch = [], [], [], []

NUM_EPOCHS = 100

print('Training started:\n')

for epoch in range(NUM_EPOCHS):

    D_real_iter, D_fake_iter, loss_dis_iter, loss_gen_iter = [], [], [], []

    for real_batch, _ in data_loader:
```

```python
        # STEP 1: train discriminator
        # ==================================
        # Train with real data
        discriminator.zero_grad()

        real_batch = real_batch.to(device)
        real_labels = torch.ones((real_batch.shape[0],), dtype=torch.float).to(device)

        output = discriminator(real_batch).view(-1)
        loss_real = criterion(output, real_labels)

        # Iteration book-keeping
        D_real_iter.append(output.mean().item())

        # Train with fake data
        noise = torch.randn(real_batch.shape[0], LATENT_SIZE, 1, 1).to(device)

        fake_batch = generator(noise)
        fake_labels = torch.zeros_like(real_labels)

        output = discriminator(fake_batch.detach()).view(-1)
        loss_fake = criterion(output, fake_labels)

        # Update discriminator weights
        loss_dis = loss_real + loss_fake
        loss_dis.backward()
        optimizerD.step()

        # Iteration book-keeping
        loss_dis_iter.append(loss_dis.mean().item())
        D_fake_iter.append(output.mean().item())

        # STEP 2: train generator
        # ==================================
        generator.zero_grad()
        output = discriminator(fake_batch).view(-1)
        loss_gen = criterion(output, real_labels)
        loss_gen.backward()

        # Book-keeping
        loss_gen_iter.append(loss_gen.mean().item())

        # Update generator weights and store loss
        optimizerG.step()

    print(f"Epoch ({epoch + 1}/{NUM_EPOCHS})\t",
          f"Loss_G: {mean(loss_gen_iter):.4f}",
          f"Loss_D: {mean(loss_dis_iter):.4f}\t",
          f"D_real: {mean(D_real_iter):.4f}",
          f"D_fake: {mean(D_fake_iter):.4f}")

    # Epoch book-keeping
    loss_gen_epoch.append(mean(loss_gen_iter))
    loss_dis_epoch.append(mean(loss_dis_iter))
    D_real_epoch.append(mean(D_real_iter))
    D_fake_epoch.append(mean(D_fake_iter))

    # Keeping track of the evolution of a fixed noise latent vector
    with torch.no_grad():
        fake_images = generator(fixed_noise).detach().cpu()
        img_list.append(utils.make_grid(fake_images, normalize=True, nrows=10))

print("\nTraining ended.")
```

In [16]:

```
#training loss
plt.plot(np.array(loss_gen_epoch), label='loss_gen')
plt.plot(np.array(loss_dis_epoch), label='loss_dis')
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend();
```

In [17]:

```
plt.plot(np.array(D_real_epoch), label='D_real')
plt.plot(np.array(D_fake_epoch), label='D_fake')
plt.xlabel("Epoch")
plt.ylabel("Probability")
plt.legend();
```

In [18]:

```
%%capture

fig = plt.figure(figsize=(10, 10))
ims = [[plt.imshow(np.transpose(i,(1, 2, 0)), animated=True)] for i in img_list[::10]]
ani = animation.ArtistAnimation(fig, ims, interval=500, repeat_delay=2000, blit=True)
ani.save('GAN.gif', writer='imagemagick', fps=2)
```

In [19]:

```
HTML(ani.to_jshtml())
```

In [ ]: