Dataset downloading

```
from google.colab import drive
drive.mount('/content/gdrive/', force_remount=True)
!unzip -q /content/gdrive/My\ Drive/zuccarrot.zip
```

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=947318989803-6bn6qk8qdgf4n4g3pfee6491hc0brc4i.apps.googleusercontent.com&redirect_uri=urn%3ai

Enter your authorization code:
··········
Mounted at /content/gdrive/
```

```
dataset_path = 'zuccarrot/'
```

Library imports

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import models
import torch.optim
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
from torchvision.utils import save_image
import torchvision.transforms as transforms
from PIL import Image
from time import time
import datetime
import numpy as np
import itertools
import os
import glob
import random
import matplotlib.pyplot as plt
from matplotlib import rcParams
rcParams['figure.figsize'] = (20, 10)

torch.manual_seed(42)
np.random.seed(42)
torch.cuda.manual_seed(42)
torch.backends.cudnn.deterministic = True
```

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

cuda

Dataloader

```
class ImageDataset(Dataset):
    def __init__(self, files_path, data_transforms, mode='train'):
        self.transform = data_transforms

        self.files_A = sorted(glob.glob(os.path.join(files_path, '{}A/*.*'.format(mode))))
        self.files_B = sorted(glob.glob(os.path.join(files_path, '{}B/*.*'.format(mode))))

    def __getitem__(self, index):
        image_A = self.transform(Image.open(self.files_A[index % len(self.files_A)]))
        image_B = self.transform(Image.open(self.files_B[index % len(self.files_B)]))
        return {'A': image_A, 'B': image_B}

    def __len__(self):
        return max(len(self.files_A), len(self.files_B))
```

```
batch_size = 5
```

```
dataset = ImageDataset(files_path=dataset_path,
                       data_transforms=transforms.Compose([
                           transforms.Resize((286, 286), Image.BICUBIC),
                           transforms.RandomCrop((256, 256)),
                           transforms.RandomHorizontalFlip(),
                           transforms.ToTensor(),
                           transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
                          ]),
                       )

dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size, shuffle=True)
```

Generator

```
class ResidualBlock(nn.Module):
    def __init__(self, in_channels):
        super(ResidualBlock, self).__init__()

        self.res = nn.Sequential(nn.ReflectionPad2d(1),
                                 nn.Conv2d(in_channels, in_channels, kernel_size=(3, 3)),
                                 nn.InstanceNorm2d(in_channels),
                                 nn.ReLU(inplace=True),
                                 nn.ReflectionPad2d(1),
                                 nn.Conv2d(in_channels, in_channels, kernel_size=(3, 3)),
                                 nn.InstanceNorm2d(in_channels))
```

```
    def forward(self, x):
        return x + self.res(x)

class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()

        self.layers = nn.Sequential(
                  nn.ReflectionPad2d(3),
                  nn.Conv2d(3, 64, 7, stride=1, padding=0),
                  nn.InstanceNorm2d(64),
                  nn.ReLU(inplace=True),

                  nn.ReflectionPad2d(1),
                  nn.Conv2d(64, 128, 3, stride=2, padding=0),
                  nn.InstanceNorm2d(128),
                  nn.ReLU(inplace=True),
                  nn.ReflectionPad2d(1),
                  nn.Conv2d(128, 256, 3, stride=2, padding=0),
                  nn.InstanceNorm2d(256),
                  nn.ReLU(inplace=True),

                  ResidualBlock(256),
                  ResidualBlock(256),
                  ResidualBlock(256),
                  ResidualBlock(256),
                  ResidualBlock(256),
                  ResidualBlock(256),
                  ResidualBlock(256),
                  ResidualBlock(256),
                  ResidualBlock(256),

                  nn.ConvTranspose2d(256, 128, 4, stride=2, padding=1),
                  nn.InstanceNorm2d(128),
                  nn.ReLU(inplace=True),
                  nn.ConvTranspose2d(128, 64, 4, stride=2, padding=1),
                  nn.InstanceNorm2d(64),
                  nn.ReLU(inplace=True),

                  nn.ReflectionPad2d(3),
                  nn.Conv2d(64, 3, 7, stride=1, padding=0),
                  nn.Tanh()
        )

    def forward(self, x):
        x = self.layers(x)
        return x
```

Discriminator

```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()

        self.layers = nn.Sequential(
            nn.ReflectionPad2d(1),
            nn.Conv2d(3, 64, 4, stride=2, padding=0), # 256 -> 128
            nn.LeakyReLU(0.2, inplace=True),

            nn.ReflectionPad2d(1),
            nn.Conv2d(64, 128, 4, stride=2, padding=0),  # 128 -> 64
            nn.InstanceNorm2d(128),
            nn.LeakyReLU(0.2, inplace=True),

            nn.ReflectionPad2d(1),
            nn.Conv2d(128, 256, 4, stride=2, padding=0), # 64 -> 32
            nn.InstanceNorm2d(256),
            nn.LeakyReLU(0.2, inplace=True),

            nn.ReflectionPad2d(1),
            nn.Conv2d(256, 512, 4, padding=0), # 32 -> 31
            nn.InstanceNorm2d(512),
            nn.LeakyReLU(0.2, inplace=True),

            nn.ReflectionPad2d(1),
            nn.Conv2d(512, 1, 4, padding=0),  # 31 -> 30
        )

    def forward(self, x):
        x = self.layers(x)
        x = F.avg_pool2d(x, x.size()[2:])  # 30 -> 1
        x = torch.flatten(x, 1)
        return x
```

Auxiliary functions

```
def weights_init_normal(m):
    if isinstance(m, nn.Conv2d) or isinstance(m, nn.ConvTranspose2d):
      torch.nn.init.normal_(m.weight, 0.0, 0.02)
      if hasattr(m, 'bias') and m.bias is not None:
        torch.nn.init.constant_(m.bias.data, 0.0)
```

```
def visualize(loss_dict):
    i = 1
    for key in loss_dict:
      plt.subplot(5, 2, i)
      plt.plot(loss_dict[key], label = key+'_train')
      plt.title(key)
      plt.xlabel('Epochs')
```

```
    plt.xlabel('Epochs')
    plt.ylabel(key)
    plt.legend()
    i += 1
  plt.show()
```

```
class ReplayBuffer():
    def __init__(self, max_size=50):
        assert (max_size > 0), 'max_size should be > 0'
        self.max_size = max_size
        self.data = []

    def push_and_pop(self, data):
        to_return = []
        for element in data.data:
            element = torch.unsqueeze(element, 0)
            if len(self.data) < self.max_size:
                self.data.append(element)
                to_return.append(element)
            else:
                if random.uniform(0,1) > 0.5:
                    i = random.randint(0, self.max_size-1)
                    to_return.append(self.data[i].clone())
                    self.data[i] = element
                else:
                    to_return.append(element)
        return torch.cat(to_return)
```

Initialize model, loss, optimizer

```
Generator_A2B = Generator().to(device)
Generator_B2A = Generator().to(device)
Discriminator_A = Discriminator().to(device)
Discriminator_B = Discriminator().to(device)
```

```
Generator_A2B.apply(weights_init_normal)
Generator_B2A.apply(weights_init_normal)
Discriminator_A.apply(weights_init_normal)
Discriminator_B.apply(weights_init_normal)
```

```
criterion_GAN = torch.nn.MSELoss()
criterion_cycle = torch.nn.L1Loss()
criterion_identity = torch.nn.L1Loss()
```

```
optimizer_G = torch.optim.Adam(itertools.chain(Generator_A2B.parameters(), Generator_B2A.parameters()), lr
optimizer_D_A = torch.optim.Adam(Discriminator_A.parameters(), lr=2e-4)
optimizer_D_B = torch.optim.Adam(Discriminator_B.parameters(), lr=2e-4)
```

```
lr_scheduler_G = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer_G, mode='min', factor=0.1, patience=
lr_scheduler_D_A = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer_D_A, mode='min', factor=0.1, patie
lr_scheduler_D_B = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer_D_B, mode='min', factor=0.1, patie
```

```
Tensor = torch.cuda.FloatTensor
input_A = Tensor(batch_size, 3, 256, 256)
input_B = Tensor(batch_size, 3, 256, 256)

target_real = Tensor(batch_size, 1).fill_(1.0)
target_fake = Tensor(batch_size, 1).fill_(0.0)

fake_A_buffer = ReplayBuffer()
fake_B_buffer = ReplayBuffer()
```

Training

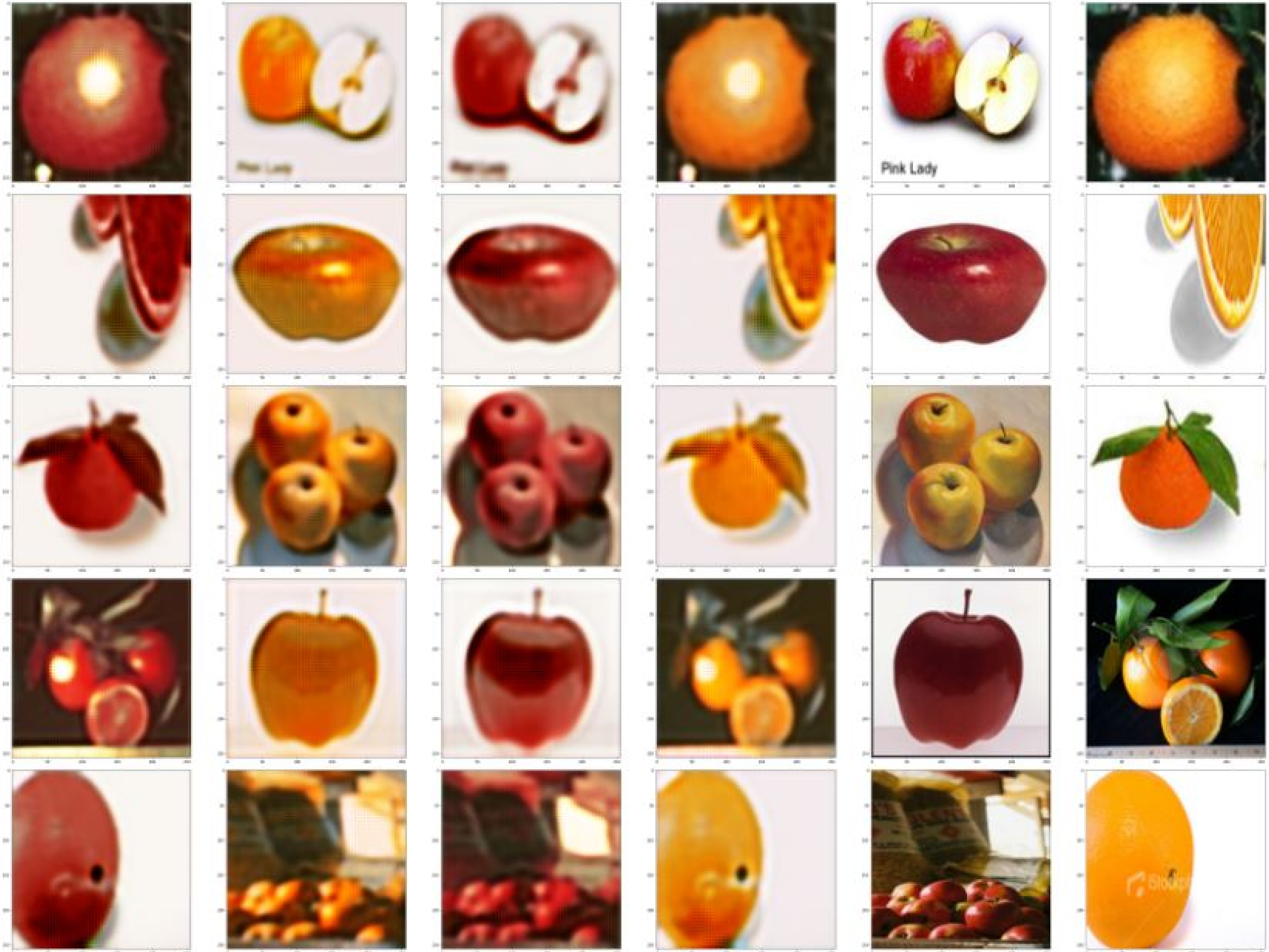
```
num_epochs = 100

loss_GAN_A2B_history = []
loss_GAN_B2A_history = []
loss_cycle_ABA_history = []
loss_cycle_BAB_history = []
loss_D_A_history = []
loss_D_B_history = []
identity_loss_A_history = []
identity_loss_B_history = []
G_loss_history = []
D_loss_history = []
```

```
Generator_A2B.train()
Generator_B2A.train()
Discriminator_A.train()
Discriminator_B.train()

prev_time = time()
for epoch in range(num_epochs):
    loss_GAN_A2B_sum = 0.
    loss_GAN_B2A_sum = 0.
    loss_cycle_ABA_sum = 0.
    loss_cycle_BAB_sum = 0.
    loss_D_A_sum = 0.
    loss_D_B_sum = 0.
    identity_loss_A_sum = 0.
    identity_loss_B_sum = 0.
    G_loss_sum = 0.
    D_loss_sum = 0.
    since = time()
    for i, batch in enumerate(dataloader):
```

```
real_A = input_A.copy_(batch['A'])
real_B = input_B.copy_(batch['B'])

optimizer_G.zero_grad()

# Identity loss
identity_A = Generator_B2A(real_A)
identity_loss_A = criterion_identity(identity_A, real_A)*0.5

identity_B = Generator_A2B(real_B)
identity_loss_B = criterion_identity(identity_B, real_B)*0.5

identity_loss_A_sum += identity_loss_A.item()
identity_loss_B_sum += identity_loss_B.item()

# Adversarial loss
fake_B = Generator_A2B(real_A)
pred_fake = Discriminator_B(fake_B)
loss_GAN_A2B = criterion_GAN(pred_fake, target_real)*0.5

fake_A = Generator_B2A(real_B)
pred_fake = Discriminator_A(fake_A)
loss_GAN_B2A = criterion_GAN(pred_fake, target_real)*0.5

loss_GAN_A2B_sum += loss_GAN_A2B.item()
loss_GAN_B2A_sum += loss_GAN_B2A.item()

# Cycle loss
recovered_A = Generator_B2A(fake_B)
loss_cycle_ABA = criterion_cycle(recovered_A, real_A)*5.

recovered_B = Generator_A2B(fake_A)
loss_cycle_BAB = criterion_cycle(recovered_B, real_B)*5.

loss_cycle_ABA_sum += loss_cycle_ABA.item()
loss_cycle_BAB_sum += loss_cycle_BAB.item()

# Generator loss =  adversarial loss + cycle loss + identity loss
loss_G = loss_GAN_A2B + loss_GAN_B2A + loss_cycle_ABA + loss_cycle_BAB + identity_loss_A + identit

G_loss_sum += loss_G.item()

loss_G.backward()
optimizer_G.step()

optimizer_D_A.zero_grad()

# DiscriminatorA loss
pred_real = Discriminator_A(real_A)
loss_D_real = criterion_GAN(pred_real, target_real)

fake_A = fake_A_buffer.push_and_pop(fake_A)
```

```
pred_fake = Discriminator_A(fake_A.detach())
loss_D_fake = criterion_GAN(pred_fake, target_fake)

loss_D_A = (loss_D_real + loss_D_fake)*0.5

loss_D_A_sum += loss_D_A.item()

loss_D_A.backward()
optimizer_D_A.step()

optimizer_D_B.zero_grad()

# DiscriminatorB loss
pred_real = Discriminator_B(real_B)
loss_D_real = criterion_GAN(pred_real, target_real)

fake_B = fake_B_buffer.push_and_pop(fake_B)
pred_fake = Discriminator_B(fake_B.detach())
loss_D_fake = criterion_GAN(pred_fake, target_fake)

loss_D_B = (loss_D_real + loss_D_fake)*0.5

loss_D_B_sum += loss_D_B.item()

loss_D_B.backward()
optimizer_D_B.step()

D_loss_sum += (loss_D_A + loss_D_B).item()


batches_done = epoch * len(dataloader) + i
batches_left = num_epochs * len(dataloader) - batches_done
time_left = datetime.timedelta(seconds=batches_left * (time() - prev_time))
prev_time = time()

print(
    "\r|Epoch {}/{}| |Batch {}/{}| |D_loss: {:.3f}| |G_loss: {:.3f}, adv: {:.3f}, cycle: {:.3f}, i
    format(epoch,
           num_epochs,
           i,
           len(dataloader),
           (loss_D_A + loss_D_B).item(),
           loss_G.item(),
           (loss_GAN_A2B + loss_GAN_B2A).item(),
           (loss_cycle_ABA + loss_cycle_BAB).item(),
           (identity_loss_A + identity_loss_B).item(),
           time_left
      )
)

lr_scheduler_G.step(loss_G)
lr_scheduler_D_A.step(loss_D_A)
lr_scheduler_D_B.step(loss_D_B)
```

```
        lr_scheduler_D_B.step(loss_D_B)

    loss_GAN_A2B_history.append(loss_GAN_A2B_sum/len(dataloader))
    loss_GAN_B2A_history.append(loss_GAN_B2A_sum/len(dataloader))
    loss_cycle_ABA_history.append(loss_cycle_ABA_sum/len(dataloader))
    loss_cycle_BAB_history.append(loss_cycle_BAB_sum/len(dataloader))
    loss_D_A_history.append(loss_D_A_sum/len(dataloader))
    loss_D_B_history.append(loss_D_B_sum/len(dataloader))
    identity_loss_A_history.append(identity_loss_A_sum/len(dataloader))
    identity_loss_B_history.append(identity_loss_B_sum/len(dataloader))
    G_loss_history.append(G_loss_sum/len(dataloader))
    D_loss_history.append(D_loss_sum/len(dataloader))
```


Pink Lady


Green Radish
Carrots
Pork Bones