Configuration Manual

MSc Research Project
Data Analytics

Sahil Bhujbal
Student ID: x18183395

School of Computing
National College of Ireland

Supervisor: Prof. Vladimir Milosavljevic

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Sahil Rajesh Bhujbal		
Student ID:	x18183395		
Programme:	MSc in Data Analytics	**Year:**	2019-2020
Module:	MSc Research Project		
Supervisor:	Prof. Vladimir Milosavljevic		
Submission Due Date:	28th September, 2020		
Project Title:	Configuration Manual		
Word Count:	1180 **Page Count:** 13		

Signature: ………

Date: ………

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Sahil Bhujbal
Student ID: x18183395

1 Introduction

This configuration manual provides a detailed report on the system configurations used and coding information regarding the three modelling phases implemented for the below mentioned research project:

"Enhancing Image Reconstruction with Prediction model using Deep Convolutional GANs"

2 System Configurations

This section covers both the Hardware as well as Software specifications required to undertake this research.

2.1 Hardware Requirements

- **OS type:** Windows 10 Home Environment
- **Processor:** Intel® Core™ i5-9300H CPU @ 2.40GHz
- **Installed RAM:** 8.00 GB
- **GPU:** NVIDIA GeForce GTX 1650 4.00 GB
- **Internal Storage:** 1 TB HDD

2.2 Software Requirements

- **Anaconda Environment-built Jupyter Notebook:** It is an open-source platform provided by the Anaconda company which allows to install and implement various software like the Jupyter Notebook and Spyder for Python programming, R-Studio for R programming, etc.[1] This research project uses the Jupyter Notebook to perform Python programming and Machine Learning related tasks.
- **Python Programming Language:** Python is installed in the system and a Global Environment is created to perform Deep Learning tasks on the GPU using the Jupyter Notebook. Python version in use is version 3.7.7.
- **CUDA Environment and PyTorch Library:** The Jupyter Notebook is integrated with CUDA using the created Global Environment variable using the NVIDIA cuDNN software, making the code capable to perform on the GPU. PyTorch Library is a Facebook-built open source Machine Learning library for performing Deep Learning tasks such as Computer Vision, Natural Language Processing, etc.[2] This project uses the library to implement Convolutional Neural Networks. PyTorch version in use is version 1.5.1.

[1] https://www.anaconda.com/ [Accessed on 12th August 2020]
[2] https://pytorch.org/ [Accessed on 12th August 2020]

3 Research Project Advancement

The proposed research project is implemented with the help of above mentioned Design Requirements. This research is divided into four phases performing experiments on three datasets respectively, covering all the aspects of Data Analytics and Machine Learning such as Data Pre-processing, Model implementation and Evaluation.

3.1 Data Pre-Processing

Since all three datasets are in form of images, the data pre-processing phase covers resizing images and splitting the data into train and validation folders and saving using the OS library. Note that, this phase is implemented on Celeb-A Faces Dataset[3] and the Flowers Dataset[4] as images in these datasets have redundant information at edges and needed to be removed. The Pokémon Dataset[5] have satisfied image requirements for modelling, and Just Image Conversion is performed in the Model Implementation phase.
Below is the code implementation of pre-processing for the Celeb-A Faces Dataset.

```
# Importing required libraries.
import os
from PIL import Image
import splitfolders as sf
```

```
# Providing path for the Celeb-A Faces dataset and saving the processed data.
data_dir = './data/celeba/main/'
save_train = './data/celeba/resized_celeba/train/data/'
save_test = './data/celeba/resized_celeba/val/data/'

image_size = 64
crop_size = 108
```

```
# Split with a ratio of 70-30.
sf.ratio(data_dir, output="./data/celeba/output", seed=1234, ratio=(.7, .3), group_prefix=None) # default values
```

```
Copying files: 202599 files [05:18, 636.72 files/s]
```

```
celeba_train = './data/celeba/output/train/data/'
celeba_test = './data/celeba/output/val/data/'
```

```
if not os.path.isdir(save_train):
    os.mkdir(save_train)

img_list = os.listdir(celeba_train)
```

```
# Code for resizing the Celeb-A dataset training images
for i in range(100000):
    img = Image.open(celeba_train + img_list[i])
    c_x = (img.size[0] - crop_size) // 2
    c_y = (img.size[1] - crop_size) // 2
    img = img.crop([c_x, c_y, c_x + crop_size, c_y + crop_size])
    img = img.resize((image_size, image_size), Image.BILINEAR)
    img.save(save_train + img_list[i], 'JPEG')

    if i % 1000 == 0:
        print('Resizing %d images...' % i)
```

[3] http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html [Accessed on 10th August 2020]
[4] http://www.robots.ox.ac.uk/~vgg/data/flowers/102/index.html [Accessed on 10th August 2020]
[5] https://www.kaggle.com/kvpratama/pokemon-images-dataset [Accessed on 10th August 2020]

```
if not os.path.isdir(save_test):
    os.mkdir(save_test)

img_list = os.listdir(celeba_test)
```

```
# Code for resizing the Celeb-A dataset testing images
for i in range(60000):
    img = Image.open(celeba_test + img_list[i])
    c_x = (img.size[0] - crop_size) // 2
    c_y = (img.size[1] - crop_size) // 2
    img = img.crop([c_x, c_y, c_x + crop_size, c_y + crop_size])
    img = img.resize((image_size, image_size), Image.BILINEAR)
    img.save(save_test + img_list[i], 'JPEG')

    if i % 1000 == 0:
        print('Resizing %d images...' % i)
```

Figure 1: Code for Data Pre-processing of Celeb-A Dataset

Below is the code implementation of pre-processing for the Flowers Dataset.

```
if not os.path.isdir(flowers_save_test):
    os.mkdir(flowers_save_test)

img_list = os.listdir(test_dir)
```

```
# Code for resizing the Flowers dataset testing images
for i in range(2900):
    img = Image.open(test_dir + img_list[i])
    c_x = (img.size[0] - crop_size) // 2
    c_y = (img.size[1] - crop_size) // 2
    img = img.crop([c_x, c_y, c_x + crop_size, c_y + crop_size])
    img = img.resize((image_size, image_size), Image.BILINEAR)
    img.save(flowers_save_test + img_list[i], 'JPEG')

    if i % 1000 == 0:
        print('Resizing %d images...' % i)
```

```
# Providing path for the Flowers dataset and saving the processed data.
flowers_data_dir = './data/flowers/main/'
flowers_save_train = './data/flowers/resized_flowers/train/data/'
flowers_save_test = './data/flowers/resized_flowers/val/data/'

image_size = 64
crop_size = 108
```

```
# Split with a ratio of 80-20.
sf.ratio(flowers_data_dir, output="./data/flowers/output", seed=1234, ratio=(.8, .2), group_prefix=None) # default values
```

```
Copying files: 8189 files [00:17, 478.96 files/s]
```

```
train_dir = './data/flowers/output/train/jpg/'
test_dir = './data/flowers/output/val/jpg/'
```

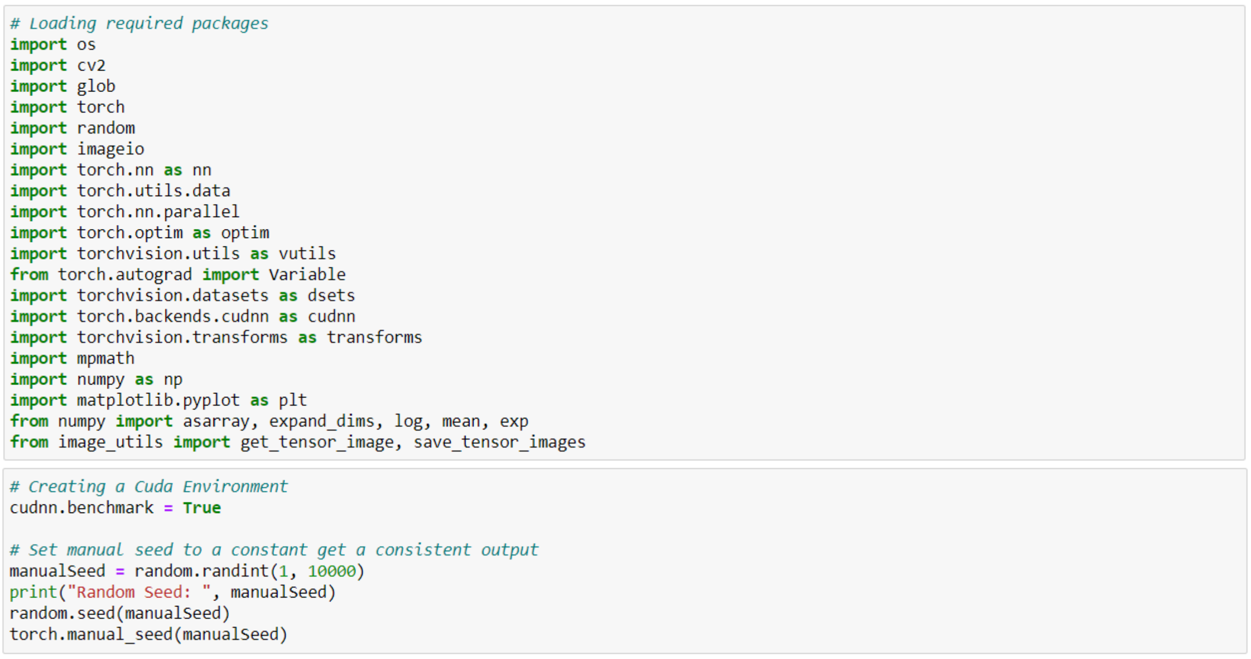
```
if not os.path.isdir(flowers_save_train):
    os.mkdir(flowers_save_train)

img_list = os.listdir(train_dir)
```

```
# Code for resizing the Flowers dataset training images
for i in range(7500):
    img = Image.open(train_dir + img_list[i])
    c_x = (img.size[0] - crop_size) // 2
    c_y = (img.size[1] - crop_size) // 2
    img = img.crop([c_x, c_y, c_x + crop_size, c_y + crop_size])
    img = img.resize((image_size, image_size), Image.BILINEAR)
    img.save(flowers_save_train + img_list[i], 'JPEG')

    if i % 1000 == 0:
        print('Resizing %d images...' % i)
```

Figure 2: Code for Data Pre-processing of Flowers Dataset

3.2 Model Implementation

This research makes use of the Deep Convolutional Generative Adversarial Network (DCGAN) model (Radford, Metz and Chintala, 2016) for implementing Image Generation, Prediction and Inpainting methodologies. Various Hyperparameters are introduced, modified and applied from the original DCGAN model to enhance this model's performance.
Below is the code implementation of the DCGAN model with required libraries and initializing hyperparameters.



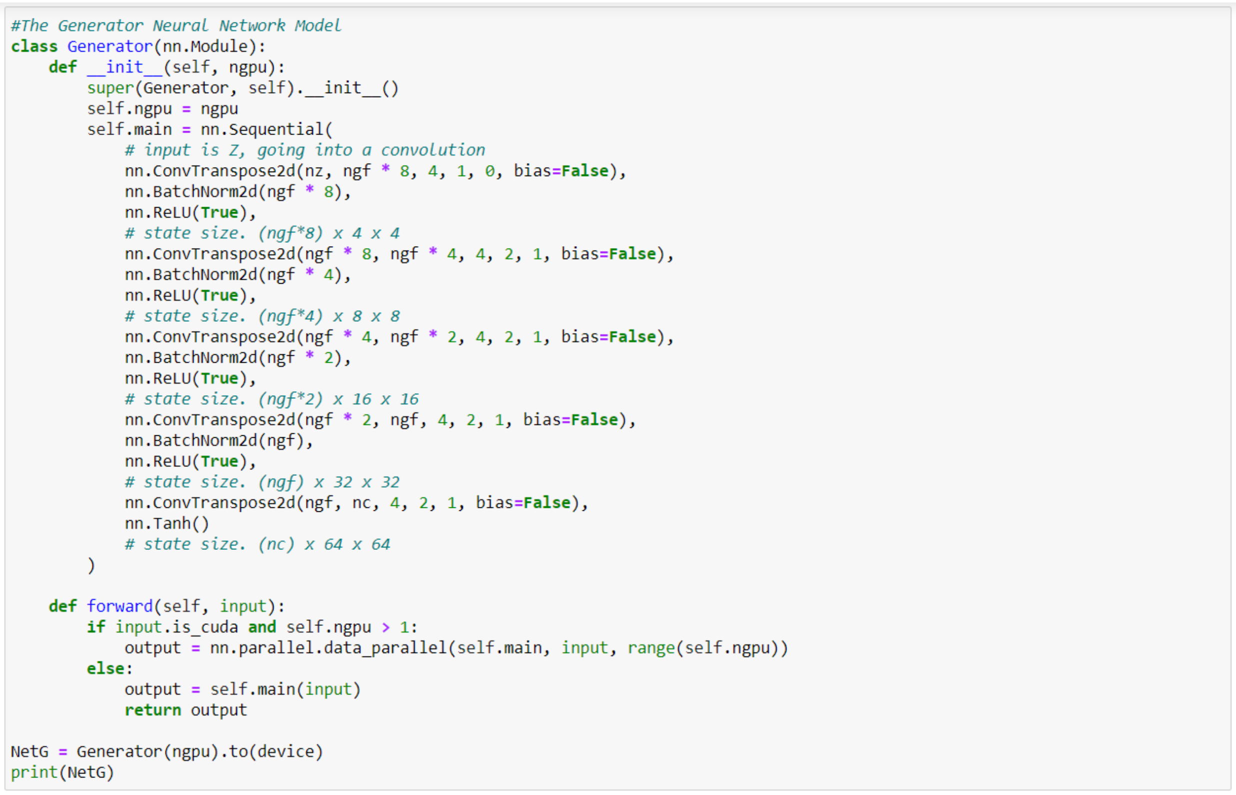
```
# Loading required packages
import os
import cv2
import glob
import torch
import random
import imageio
import torch.nn as nn
import torch.utils.data
import torch.nn.parallel
import torch.optim as optim
import torchvision.utils as vutils
from torch.autograd import Variable
import torchvision.datasets as dsets
import torch.backends.cudnn as cudnn
import torchvision.transforms as transforms
import mpmath
import numpy as np
import matplotlib.pyplot as plt
from numpy import asarray, expand_dims, log, mean, exp
from image_utils import get_tensor_image, save_tensor_images
```

```
# Creating a Cuda Environment
cudnn.benchmark = True

# Set manual seed to a constant get a consistent output
manualSeed = random.randint(1, 10000)
print("Random Seed: ", manualSeed)
random.seed(manualSeed)
torch.manual_seed(manualSeed)
```

Figure 3: Loading Libraries and Creating CUDA Environment

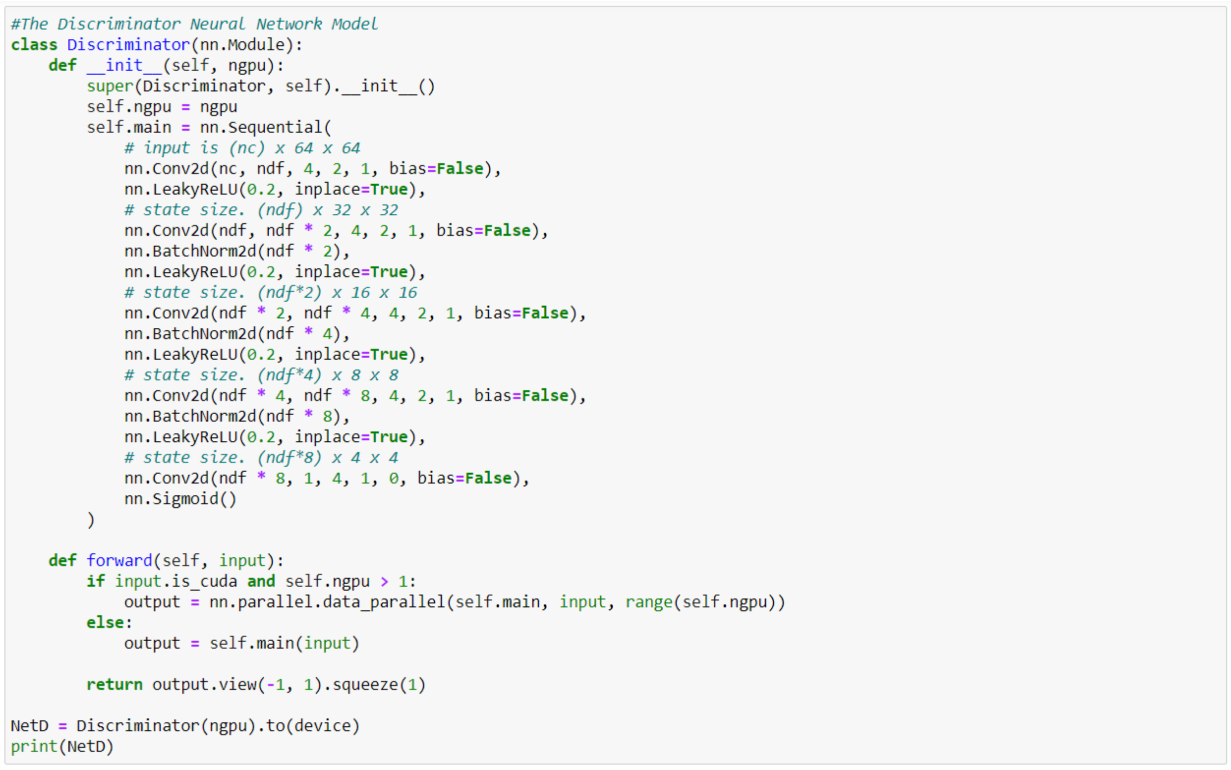
As per above snippet, all the mentioned libraries are called and implemented in all experiments along with its implementation in the DCGAN model. Few notable includes, the cv2-Computer Vision library, torch-PyTorch library, numpy and mpmath for perfroming mathematical calculations, matplotlib for plotting graphs, etc. Note that, Cuda Enviroment is alos created and seed has been auto-generated for reproducible purpose.

The code snippet for the Generator Network and the Dsicriminator Network is provided below.



```python
#The Generator Neural Network Model
class Generator(nn.Module):
    def __init__(self, ngpu):
        super(Generator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is Z, going into a convolution
            nn.ConvTranspose2d(nz, ngf * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),
            # state size. (ngf*8) x 4 x 4
            nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            # state size. (ngf*4) x 8 x 8
            nn.ConvTranspose2d(ngf * 4, ngf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            # state size. (ngf*2) x 16 x 16
            nn.ConvTranspose2d(ngf * 2, ngf, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),
            # state size. (ngf) x 32 x 32
            nn.ConvTranspose2d(ngf, nc, 4, 2, 1, bias=False),
            nn.Tanh()
            # state size. (nc) x 64 x 64
        )

    def forward(self, input):
        if input.is_cuda and self.ngpu > 1:
            output = nn.parallel.data_parallel(self.main, input, range(self.ngpu))
        else:
            output = self.main(input)
            return output

NetG = Generator(ngpu).to(device)
print(NetG)
```

Figure 4: Code of Generator Network



```python
#The Discriminator Neural Network Model
class Discriminator(nn.Module):
    def __init__(self, ngpu):
        super(Discriminator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is (nc) x 64 x 64
            nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf) x 32 x 32
            nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 2),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*2) x 16 x 16
            nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*4) x 8 x 8
            nn.Conv2d(ndf * 4, ndf * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 8),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*8) x 4 x 4
            nn.Conv2d(ndf * 8, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )

    def forward(self, input):
        if input.is_cuda and self.ngpu > 1:
            output = nn.parallel.data_parallel(self.main, input, range(self.ngpu))
        else:
            output = self.main(input)

        return output.view(-1, 1).squeeze(1)

NetD = Discriminator(ngpu).to(device)
print(NetD)
```

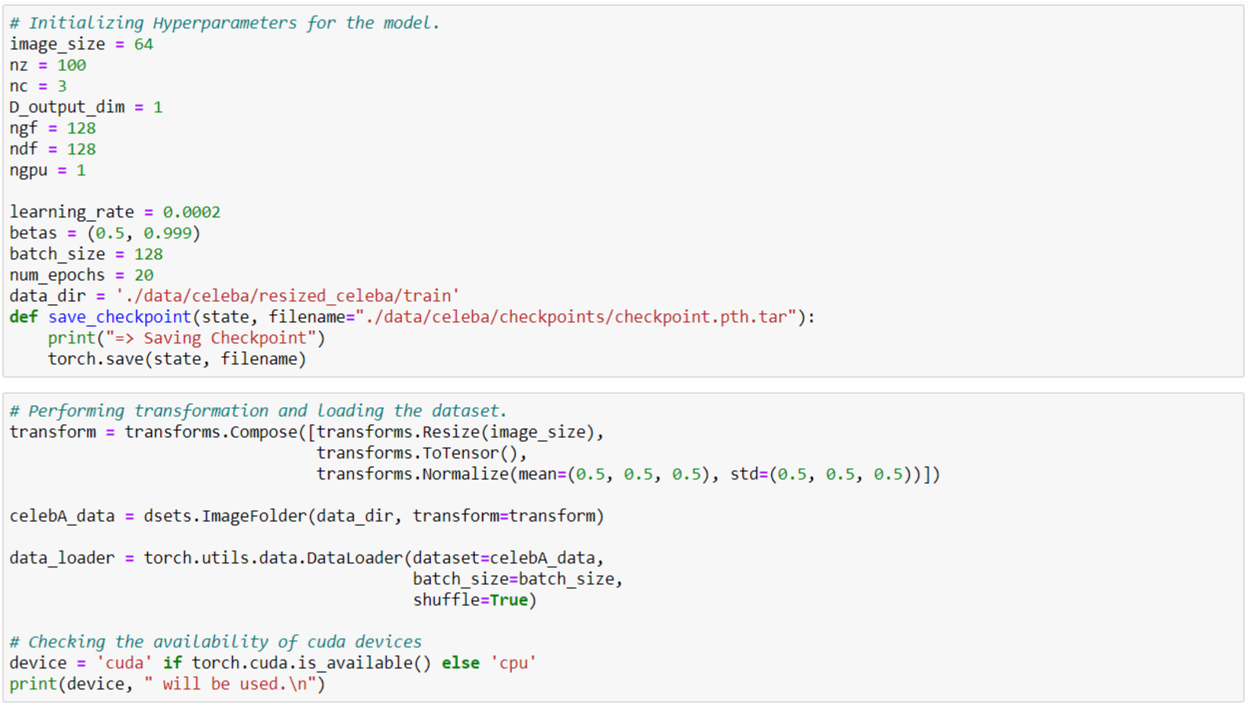
Figure 5: Code for Discriminator Network

These Neural Networks (Goodfellow *et al.*, 2014) are used for performing model training, prediction and inpainting.

3.3 Data Initialization

As mentioned earlier, each experiment is based on the complexity of the dataset images. Below code covers all variable initialization and data loading for all datasets.

3.3.1 Experiment 1: Celeb-A Faces Dataset

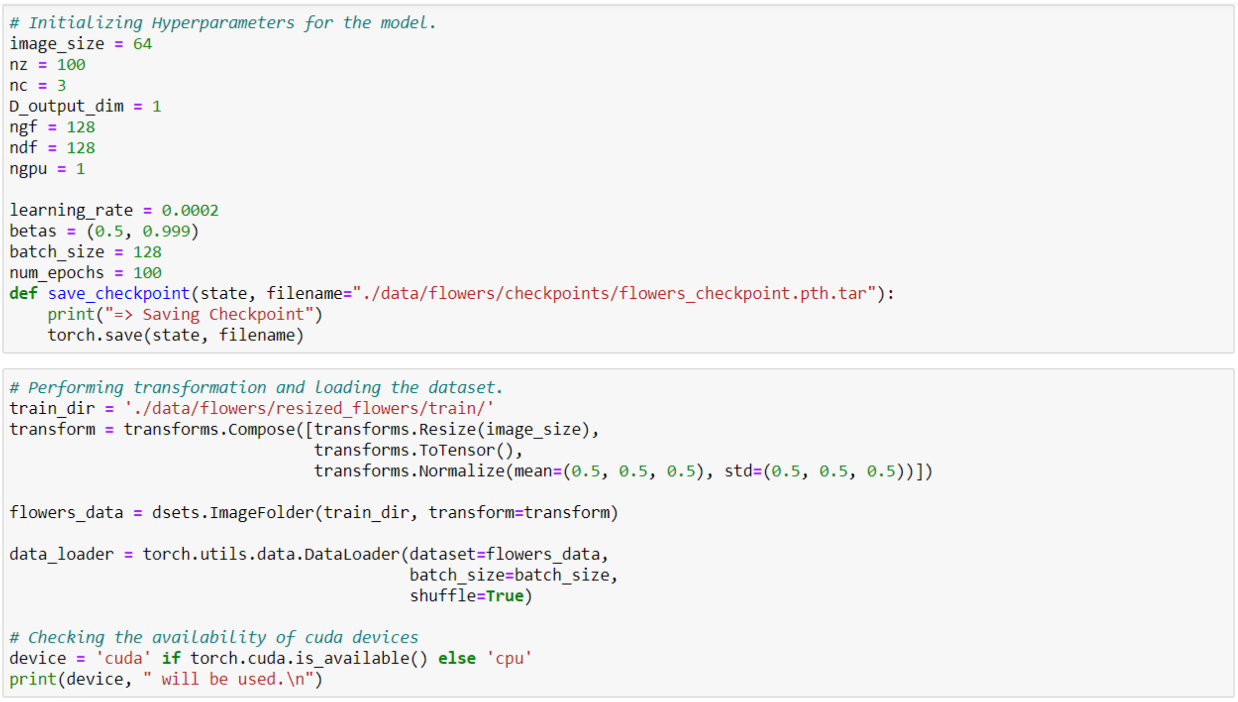


```python
# Initializing Hyperparameters for the model.
image_size = 64
nz = 100
nc = 3
D_output_dim = 1
ngf = 128
ndf = 128
ngpu = 1

learning_rate = 0.0002
betas = (0.5, 0.999)
batch_size = 128
num_epochs = 20
data_dir = './data/celeba/resized_celeba/train'
def save_checkpoint(state, filename="./data/celeba/checkpoints/checkpoint.pth.tar"):
    print("=> Saving Checkpoint")
    torch.save(state, filename)
```

```python
# Performing transformation and loading the dataset.
transform = transforms.Compose([transforms.Resize(image_size),
                                transforms.ToTensor(),
                                transforms.Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5))])

celebA_data = dsets.ImageFolder(data_dir, transform=transform)

data_loader = torch.utils.data.DataLoader(dataset=celebA_data,
                                          batch_size=batch_size,
                                          shuffle=True)

# Checking the availability of cuda devices
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print(device, " will be used.\n")
```

Figure 6: Data Initialization for Celeb-A Faces Dataset

3.3.2 Experiment 2: Flowers Dataset

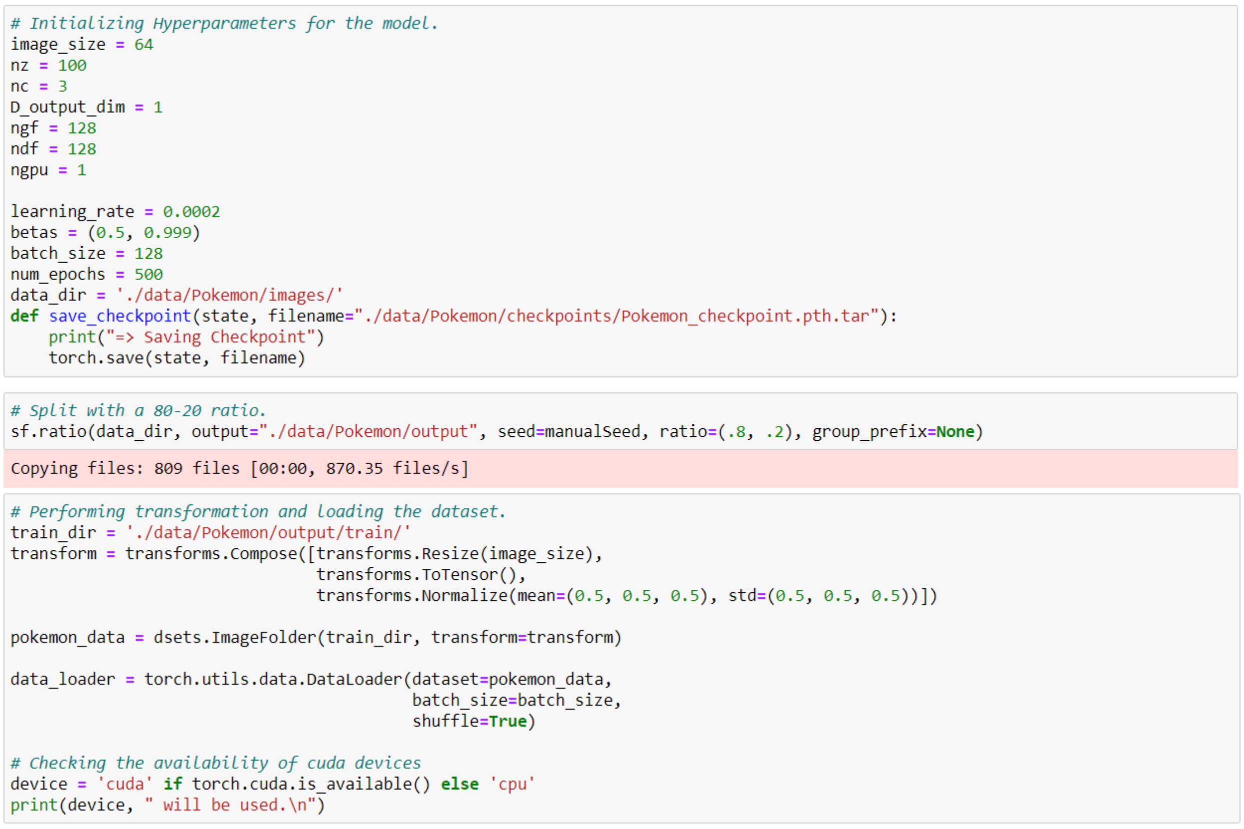


```python
# Initializing Hyperparameters for the model.
image_size = 64
nz = 100
nc = 3
D_output_dim = 1
ngf = 128
ndf = 128
ngpu = 1

learning_rate = 0.0002
betas = (0.5, 0.999)
batch_size = 128
num_epochs = 100
def save_checkpoint(state, filename="./data/flowers/checkpoints/flowers_checkpoint.pth.tar"):
    print("=> Saving Checkpoint")
    torch.save(state, filename)
```

```python
# Performing transformation and loading the dataset.
train_dir = './data/flowers/resized_flowers/train/'
transform = transforms.Compose([transforms.Resize(image_size),
                                transforms.ToTensor(),
                                transforms.Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5))])

flowers_data = dsets.ImageFolder(train_dir, transform=transform)

data_loader = torch.utils.data.DataLoader(dataset=flowers_data,
                                          batch_size=batch_size,
                                          shuffle=True)

# Checking the availability of cuda devices
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print(device, " will be used.\n")
```

Figure 7: Data Initialization for Flowers Dataset

3.3.3 Experiment 3: Pokémon Dataset



```
# Initializing Hyperparameters for the model.
image_size = 64
nz = 100
nc = 3
D_output_dim = 1
ngf = 128
ndf = 128
ngpu = 1

learning_rate = 0.0002
betas = (0.5, 0.999)
batch_size = 128
num_epochs = 500
data_dir = './data/Pokemon/images/'
def save_checkpoint(state, filename="./data/Pokemon/checkpoints/Pokemon_checkpoint.pth.tar"):
    print("=> Saving Checkpoint")
    torch.save(state, filename)
```

```
# Split with a 80-20 ratio.
sf.ratio(data_dir, output="./data/Pokemon/output", seed=manualSeed, ratio=(.8, .2), group_prefix=None)
```

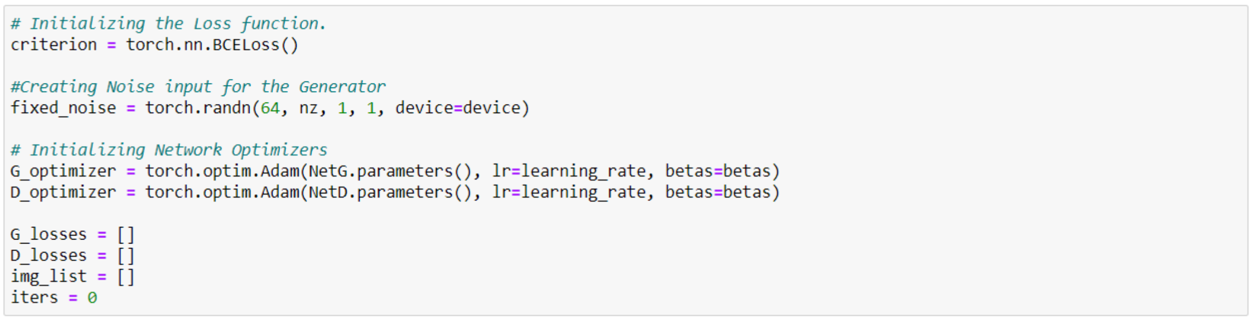
```
Copying files: 809 files [00:00, 870.35 files/s]
```

```
# Performing transformation and loading the dataset.
train_dir = './data/Pokemon/output/train/'
transform = transforms.Compose([transforms.Resize(image_size),
                                transforms.ToTensor(),
                                transforms.Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5))])

pokemon_data = dsets.ImageFolder(train_dir, transform=transform)

data_loader = torch.utils.data.DataLoader(dataset=pokemon_data,
                                          batch_size=batch_size,
                                          shuffle=True)

# Checking the availability of cuda devices
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print(device, " will be used.\n")
```

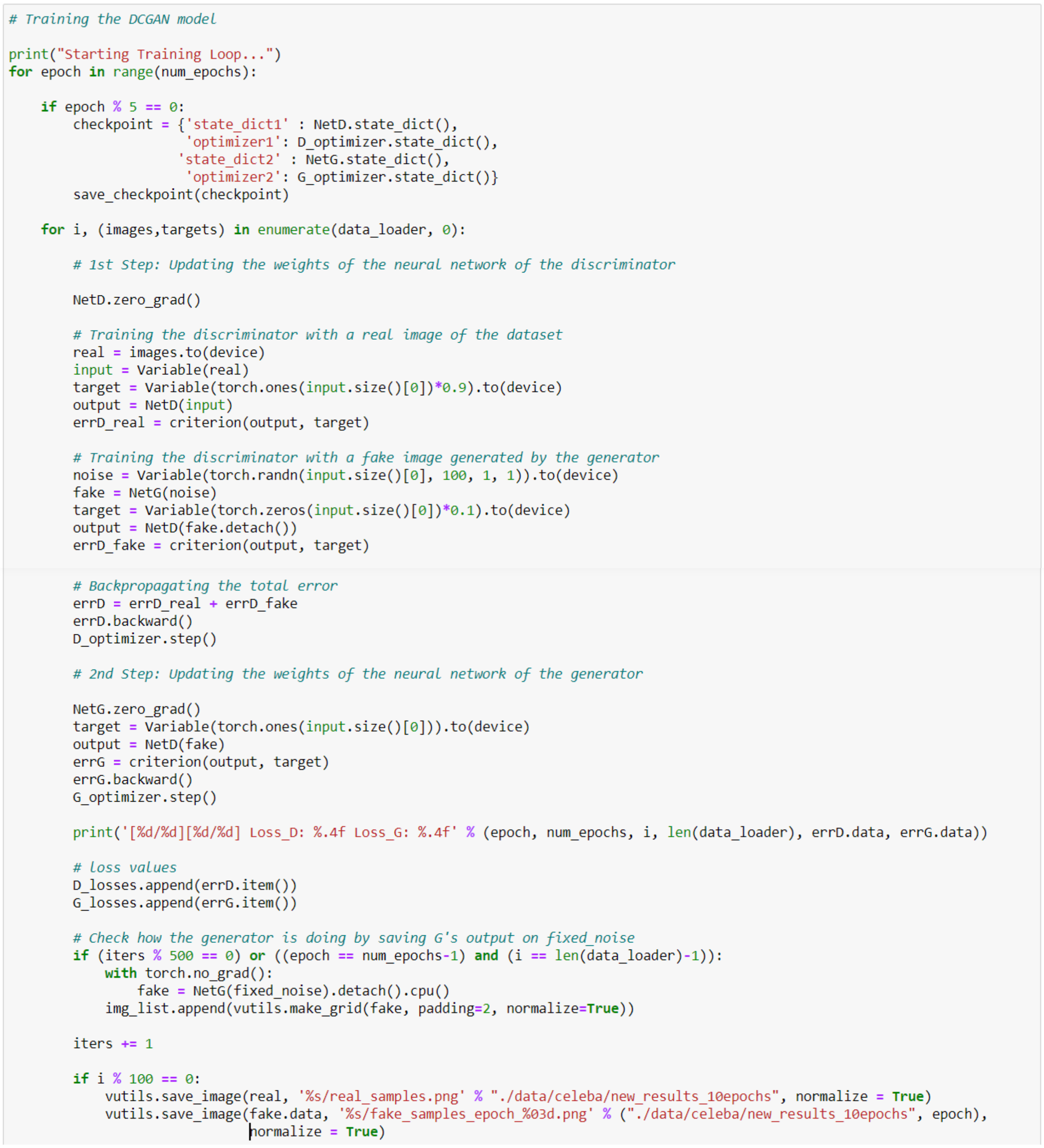
Figure 8: Data Initialization for Pokémon Dataset

3.4 Training Phase

This part covers the coding implementation of the training phase for all three experiments. Note that, models for all experiments are same with change in variables.



```
# Initializing the Loss function.
criterion = torch.nn.BCELoss()

#Creating Noise input for the Generator
fixed_noise = torch.randn(64, nz, 1, 1, device=device)

# Initializing Network Optimizers
G_optimizer = torch.optim.Adam(NetG.parameters(), lr=learning_rate, betas=betas)
D_optimizer = torch.optim.Adam(NetD.parameters(), lr=learning_rate, betas=betas)

G_losses = []
D_losses = []
img_list = []
iters = 0
```



```python
# Training the DCGAN model

print("Starting Training Loop...")
for epoch in range(num_epochs):

    if epoch % 5 == 0:
        checkpoint = {'state_dict1' : NetD.state_dict(),
                      'optimizer1': D_optimizer.state_dict(),
                     'state_dict2' : NetG.state_dict(),
                      'optimizer2': G_optimizer.state_dict()}
        save_checkpoint(checkpoint)

    for i, (images,targets) in enumerate(data_loader, 0):

        # 1st Step: Updating the weights of the neural network of the discriminator

        NetD.zero_grad()

        # Training the discriminator with a real image of the dataset
        real = images.to(device)
        input = Variable(real)
        target = Variable(torch.ones(input.size()[0])*0.9).to(device)
        output = NetD(input)
        errD_real = criterion(output, target)

        # Training the discriminator with a fake image generated by the generator
        noise = Variable(torch.randn(input.size()[0], 100, 1, 1)).to(device)
        fake = NetG(noise)
        target = Variable(torch.zeros(input.size()[0])*0.1).to(device)
        output = NetD(fake.detach())
        errD_fake = criterion(output, target)

        # Backpropagating the total error
        errD = errD_real + errD_fake
        errD.backward()
        D_optimizer.step()

        # 2nd Step: Updating the weights of the neural network of the generator

        NetG.zero_grad()
        target = Variable(torch.ones(input.size()[0])).to(device)
        output = NetD(fake)
        errG = criterion(output, target)
        errG.backward()
        G_optimizer.step()

        print('[%d/%d][%d/%d] Loss_D: %.4f Loss_G: %.4f' % (epoch, num_epochs, i, len(data_loader), errD.data, errG.data))

        # loss values
        D_losses.append(errD.item())
        G_losses.append(errG.item())

        # Check how the generator is doing by saving G's output on fixed_noise
        if (iters % 500 == 0) or ((epoch == num_epochs-1) and (i == len(data_loader)-1)):
            with torch.no_grad():
                fake = NetG(fixed_noise).detach().cpu()
            img_list.append(vutils.make_grid(fake, padding=2, normalize=True))

        iters += 1

        if i % 100 == 0:
            vutils.save_image(real, '%s/real_samples.png' % "./data/celeba/new_results_10epochs", normalize = True)
            vutils.save_image(fake.data, '%s/fake_samples_epoch_%03d.png' % ("./data/celeba/new_results_10epochs", epoch),
                              normalize = True)
```

Figure 9: Code for Training the DCGAN model

3.5 Prediction Phase

This part covers the coding implementation of the Prediction model for all experiments. The model takes the saved sate of the Generator Network and implements the model.

```python
output_dir = './data/celeba/dcgan_results/'
```

```python
if not os.path.isdir(output_dir):
    os.mkdir(output_dir)
```

```python
# Loading the Generator State from the training phase
generator = Generator(ngpu).to(device)
checkpoint = torch.load('./data/celeba/checkpoints/checkpoint.pth.tar')
generator.load_state_dict(checkpoint['state_dict2'])
print("Generator Info:")
print(generator)
```

```python
# Sample a batch from generator
sample_batch_z = torch.randn(batch_size, nz, 1, 1).to(device)
```

```python
# Function for performing Image Prediction
def predict():
    # sample images from generator
    fake_batch_images = generator(sample_batch_z)
    vutils.save_image(fake_batch_images.detach(), "%s/sample_from_generator_with_seed_{%d}.png" %
                                (output_dir, manualSeed), normalize=True)
```

```python
# Function for plotting the predicted image
def imshow(img):
    img = img / 2 + 0.5 # unnormalize
    npimg = img.cpu().numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.figure(figsize=(10,5))
    plt.show()
```

```python
predict()
imshow(vutils.make_grid(generator(sample_batch_z).detach()))
print("Sampling Done! Image saved at %s/sample_from_generator_with_seed_{%d}.png" % ((output_dir, manualSeed)))
```

Figure 10: Code for implementing the Prediction model

3.6 Inpainting Phase

This part covers the Image Reconstruction code using the pre-trained DCGAN model. Saved states of Generator as well as Discriminator are used (Yu *et al.*, 2018).

```python
num_iters = 2000
lamd = 0.1
save_dir = './data/celeba/inpainted_data'

test_data = './data/celeba/resized_celeba/data/000046.jpg'
```

```python
# Loading Generator and Discriminator Networks
generator = Generator(ngpu)
discriminator = Discriminator(ngpu)

# Moving generator and discimator to cuda device
generator.to(device)
discriminator.to(device)
```

```python
# loading Generator and Discriminator network states
checkpoint = torch.load('./data/celeba/checkpoints/checkpoint.pth.tar')
print("load trained state dict from local files...")
generator.load_state_dict(checkpoint['state_dict2'])
discriminator.load_state_dict(checkpoint['state_dict1'])
print("generator and discriminator state dict loaded, done.")

print("Generator Info:")
print(generator)
print("Discriminator Info:")
print(discriminator)
```

```
image_shape = [nc, image_size, image_size]

criteria = nn.BCELoss()
```

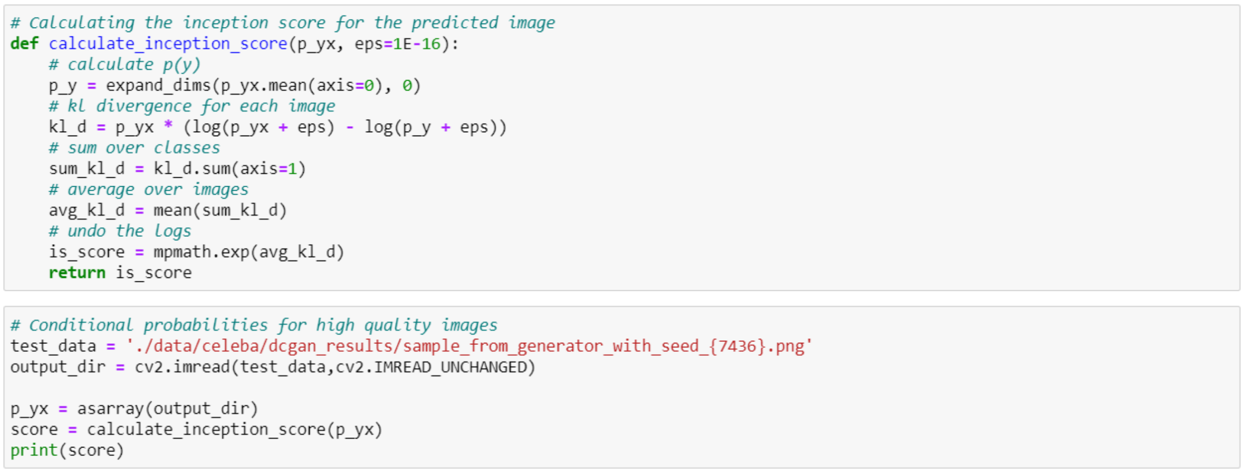
```
# Function for performing Image Inpainting
def impainting():
    # Create output folder
    source_imagedir = os.path.join(save_dir, "source_images")
    masked_imagedir = os.path.join(save_dir, "masked_images")
    impainted_imagedir = os.path.join(save_dir, "impainted_images")
    os.makedirs(source_imagedir, exist_ok=True)
    os.makedirs(masked_imagedir,exist_ok=True)
    os.makedirs(impainted_imagedir,exist_ok=True)

    # How many pictures need to be repaired in total
    num_images = len(test_data)
    # How many batches can be divided into total for processing
    num_batches = int(np.ceil(num_images / batch_size))
    for idx in range(num_batches):
        # Perform the following processing for each batch image
        lidx = idx * batch_size
        hidx = min(num_images, (idx + 1) * batch_size)
        realBatchSize = hidx - lidx

        batch_images = [get_tensor_image(imgpath) for imgpath in glob.glob(test_data[lidx:hidx])]
        batch_images = torch.stack(batch_images).to(device)

        # The input original picture is ready, start to prepare the mask
        # Temporarily only provide center mask
        mask = torch.ones(size=image_shape).to(device)
        imageCenterScale = 0.3
        lm = int(image_size * imageCenterScale)
        hm = int(image_size * (1 - imageCenterScale))
        #Mask the center of the image to 0
        mask[:,lm:hm, lm:hm] = 0.0
        masked_batch_images = torch.mul(batch_images, mask).to(device)

        # First save the original picture and masked picture
        save_tensor_images(batch_images.detach(),
                    os.path.join(source_imagedir,"source_image_batch_{}.png".format(idx)))

        save_tensor_images(masked_batch_images.detach(), os.path.join(masked_imagedir, "masked_image_batch_{}.png".format(idx)))


        z_hat = torch.rand(size=[realBatchSize,nz,1,1],dtype=torch.float32,requires_grad=True,device=device)
        z_hat.data.mul_(2.0).sub_(1.0)
        opt = optim.Adam([z_hat],lr=learning_rate)
        print("start impainting iteration for batch : {}".format(idx))
        v = torch.tensor(0,dtype=torch.float32,device=device)
        m = torch.tensor(0,dtype=torch.float32,device=device)

        for iteration in range(num_iters):
            # Iterate impainting for each batch image separately
            if z_hat.grad is not None:
                z_hat.grad.data.zero_()
            generator.zero_grad()
            discriminator.zero_grad()
            batch_images_g = generator(z_hat)
            batch_images_g_masked = torch.mul(batch_images_g,mask)
            impainting_images = torch.mul(batch_images_g,(1-mask)) + masked_batch_images
            if iteration % 100==0:
                # Save impainting picture result
                print("\nsaving impainted images for batch: {} , iteration:{}".format(idx,iteration))
                save_tensor_images(impainting_images.detach(), os.path.join(impainted_imagedir,
                                                        "impainted_image_batch_{}_iteration_{}.png".format(idx,iteration)))

            loss_context = torch.norm(
                (masked_batch_images-batch_images_g_masked),p=1)
            dis_output = discriminator(impainting_images)

            batch_labels = torch.full((realBatchSize,), 1, device=device)
            loss_perceptual = criteria(dis_output,batch_labels)

            total_loss = loss_context + lamd*loss_perceptual
            print("\r batch {} : iteration : {:4} , context_loss:{:.4f},
                perceptual_loss:{:4f}".format(idx,iteration,loss_context,loss_perceptual),end="")
            total_loss.backward()
            opt.step()
```

```
impainting()
```

Figure 11: Code for implementing the Image Inpainting model

3.7 Model Evaluation

This modeling technique makes use of the Inception Score to evaluate the Prediction model for all three experiments (Salimans *et al.*, 2016).



```
# Calculating the inception score for the predicted image
def calculate_inception_score(p_yx, eps=1E-16):
    # calculate p(y)
    p_y = expand_dims(p_yx.mean(axis=0), 0)
    # kl divergence for each image
    kl_d = p_yx * (log(p_yx + eps) - log(p_y + eps))
    # sum over classes
    sum_kl_d = kl_d.sum(axis=1)
    # average over images
    avg_kl_d = mean(sum_kl_d)
    # undo the logs
    is_score = mpmath.exp(avg_kl_d)
    return is_score
```

```
# Conditional probabilities for high quality images
test_data = './data/celeba/dcgan_results/sample_from_generator_with_seed_{7436}.png'
output_dir = cv2.imread(test_data,cv2.IMREAD_UNCHANGED)

p_yx = asarray(output_dir)
score = calculate_inception_score(p_yx)
print(score)
```

Figure 12: Model Evaluation using Inception Score Metric

References

Goodfellow, I. J. *et al.* (2014) 'Generative adversarial nets', *Advances in Neural Information Processing Systems*, 3(January), pp. 2672–2680.

Radford, A., Metz, L. and Chintala, S. (2016) 'Unsupervised representation learning with deep convolutional generative adversarial networks', *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*, pp. 1–16.

Salimans, T. *et al.* (2016) 'Improved techniques for training GANs', *Advances in Neural Information Processing Systems*, pp. 2234–2242.

Yu, J. *et al.* (2018) 'Generative Image Inpainting with Contextual Attention', *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 5505–5514. doi: 10.1109/CVPR.2018.00577.