AI Project Report

Team members: 0812250 林澤宇 0812251 李秉諺 0812252 潘翰

Codes (training with cat faces): Code

Brief Introduction About Problem

Our goal is to generate cat pictures using GAN.

Our first attempt is to generate cats including its body, however the result is not that good (can be seen in the experiment result part). Thus we tried to generate only cat faces.

The result using only cat faces to train is brilliant. After some epochs, the cat faces generated by the generator are really similar to the real cat faces (can be seen in the experiment result part).

Brief Introduction About Github Code We Refer To

Codes we refer to: DCGAN_Tutorial

This tutorial uses DCGAN to generate new celebrities from real celebrities' images.

Brief Introduction About Codes

1. Original datasets: Cat_dataset_1, Cat_dataset_2

First of all, we try to train with the original data without preprocessing, only resizing every image into 64x64 picture.

As we can see in the experiment result, the first two attempts which are trained with the whole body of cat seems not to be a good idea.

We then assume that it is because the image data is just a chaos, with some pictures show only human faces but rather cat's, and the posture of each cat can be very different

So we decide to focus on generating cat's face only, with some processed data

2. Datasets after processing: Cat_face_dataset

Cut cat face in image:

We use openCV to identify the cat's face in the original data and cut images to form a new dataset.

```
import os
import cv2
import logging
logging.basicConfig(level=logging.INFO,format='%(asctime)s - %(levelname)s: %(message)s')
logger=logging.getLogger(__name__)
ImagePath='./img/'
cnt=0
for filename in os.listdir(ImagePath):
  temp_path=ImagePath+filename
  try:
    image=cv2.imread(temp_path)
    gray=cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    face_cascade=cv2.CascadeClassifier(r'./opencv/data/haarcascades/haarcascade_frontalcatface.xml')
    faces=face_cascade.detectMultiScale(gray,scaleFactor=1.15,minNeighbors=5,minSize=(3, 3))
    for(x,y,w,h)in faces:
      crop_img=image[y:y+h,x:x+w]
      cv2.imwrite("./cat_face/cat_face_{0}.jpg".format(cnt),crop_img)
      cnt+=1
    cv2.waitKey(0)
  except:
    continue
```

Download Data

We upload our new dataset to kaggle, and download them in colab.

```
import os
import zipfile
os.environ['KAGGLE_USERNAME']="lyeeo139"
os.environ['KAGGLE_KEY']="5964a26d48f2aa2fa733ce4a76627317"
!kaggle datasets download -d lyeeo139/cat-face
local_zip='cat-face.zip'
zip_ref=zipfile.ZipFile(local_zip,'r')
zip_ref.extractall('.')
zip_ref.close()
%rm cat-face.zip
%cd data/
```

```
Downloading cat-face.zip to /content
100% 260M/261M [00:07<00:00, 66.2MB/s]
100% 261M/261M [00:07<00:00, 36.4MB/s]
/content/data
```

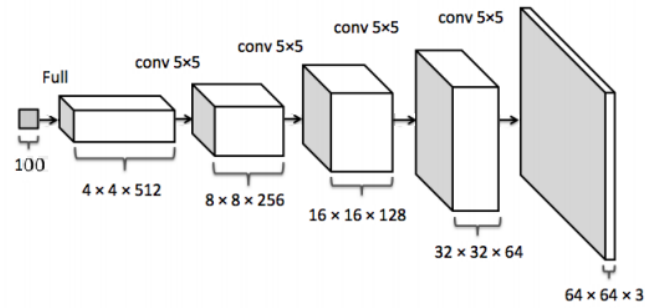
Check point

Each 5 epochs we will save the model, so that we can load the trained model when we want to train more epochs.

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

def load_checkpoint():
    model_path = "model.pt"

    if os.path.isfile(model_path):
        netG = Generator(ngpu).to(device)
        netD = Discriminator(ngpu).to(device)
        optimizerD = optim.Adam(netD.parameters(), lr=lr, betas=(beta1, 0.999))
        optimizerG = optim.Adam(netG.parameters(), lr=lr, betas=(beta1, 0.999))
        checkpoint = torch.load(model_path)
        netG.load_state_dict(checkpoint['netG_state_dict'])
        netD.load_state_dict(checkpoint['netD_state_dict'])
        optimizerG.load_state_dict(checkpoint['optimizerG_state_dict'])
        optimizerD.load_state_dict(checkpoint['optimizerD_state_dict'])
        G_losses = checkpoint['netG_loss']
        D_losses = checkpoint['netD_loss']

        netG.train()
        netD.train()
```

Algorithm(GAN)

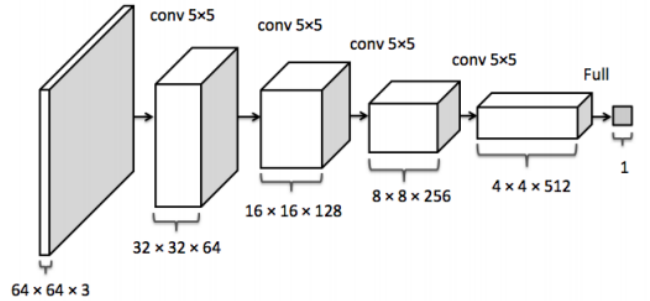
Generator:



We use the latent vector with size of 100, and then train through the above architecture,using ConvTransposed2d to expand to the next level, BatchNorm2d to get normalized data so it can fit better when ReLU is applied. Tanh function is used to transformed the data into the range of [-1,1]

Finally, with some noise add in,we get generated pictures.

Discrimimator:



Architecture is as above

Discriminator is a binary classifier,using Conv2d as layers, BatchNorm2d to normalize data and LeakyReLU as activation function. We finally use the sigmoid function to get the probability of the input picture.
We transform the 64x64 picture into a single feature indicating whether it is generated or not.

When we train these two models, we need to first train the discriminator and then switch to the generator, so they can compete with each other and get better loss.
We add random noise when generating pictures, so we can have a better chance to get more similar result as real image

Also note that we should not use mixed pictures, that is , some of them are fake, some of them are real, to train the discriminator in order to get better performance.

After training, we get a list of images generated by Generator in every 500 batches.

Differents from the code we refer to:

1. Pre-processing:

   As above.

2. Load data:

   As above.

3. Parameters:

   We tried many kinds of batch size, epochs, learning rate, the number of layers, the number of neurons in each layer and we found out that the parameters given by the website perform the best.

4. Check points:

   As above

Experiment Result

Evalutation:

1. D(x):

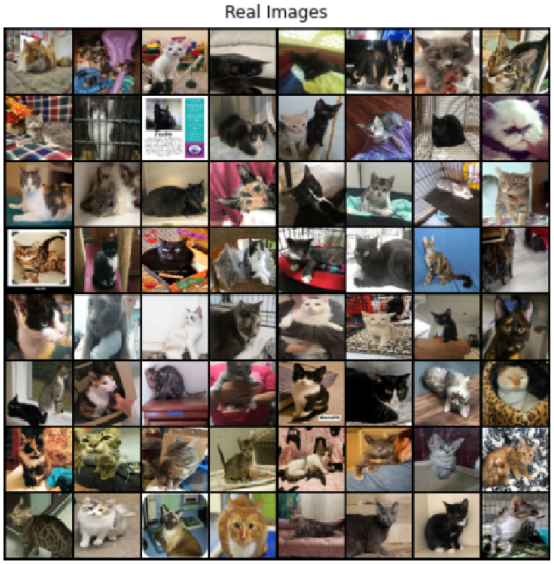
   The probability that the discriminator thinks of a real picture is real.
   For D, the more the value approaches 1 the better it is.
   This should start near 1 and converge to 0.5 when G gets better.

2. D(G(z)):

   The probability that the discriminator thinks of a fake picture is real.
   For D, the more the value approaches 0 the better it is.
   For G, the more the value approaches 1 the better it is.
   These numbers should start near 0 and converge to 0.5 as G gets better.

Training with whole cat

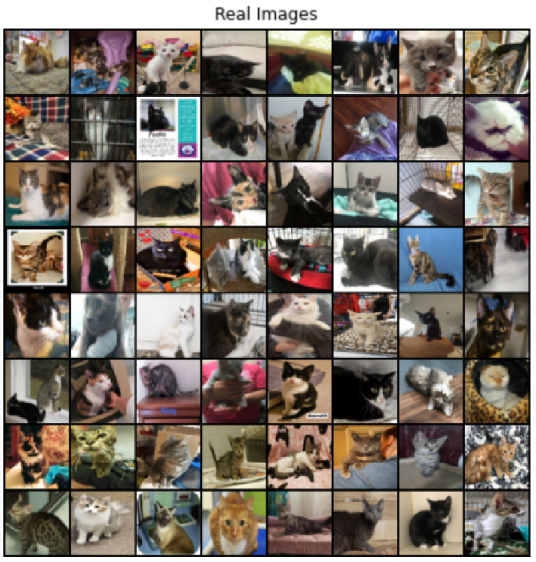
Result after training 25 epochs (batch_size = 128):

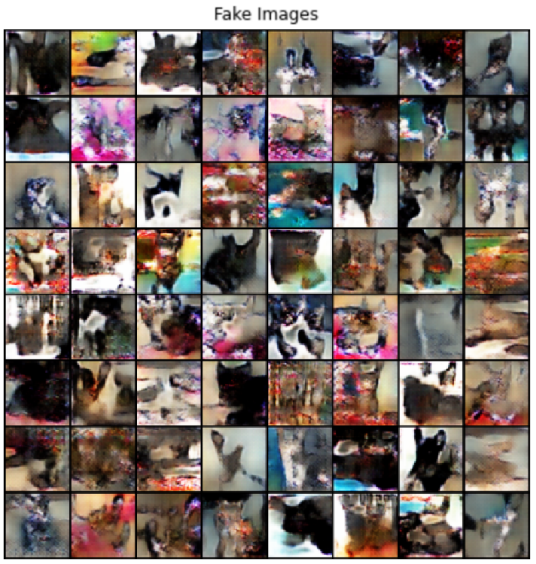


[epoch][batch]

```
[4/5][450/1029] Loss_D: 0.3672  Loss_G: 2.1952  D(x): 0.8394    D(G(z)): 0.1544 / 0.1409
[4/5][500/1029] Loss_D: 1.4235  Loss_G: 7.0669  D(x): 0.9708    D(G(z)): 0.6522 / 0.0020
[4/5][550/1029] Loss_D: 0.6021  Loss_G: 2.3041  D(x): 0.7043    D(G(z)): 0.1547 / 0.1366
[4/5][600/1029] Loss_D: 0.3313  Loss_G: 2.9471  D(x): 0.8402    D(G(z)): 0.1163 / 0.0739
[4/5][650/1029] Loss_D: 0.3823  Loss_G: 2.6022  D(x): 0.7959    D(G(z)): 0.1152 / 0.1085

[4/5][700/1029] Loss_D: 0.4054  Loss_G: 3.3726  D(x): 0.9208    D(G(z)): 0.2508 / 0.0492
[4/5][750/1029] Loss_D: 1.2356  Loss_G: 5.5123  D(x): 0.9685    D(G(z)): 0.6276 / 0.0067
[4/5][800/1029] Loss_D: 1.0047  Loss_G: 5.2770  D(x): 0.9492    D(G(z)): 0.5284 / 0.0081
[4/5][850/1029] Loss_D: 0.4203  Loss_G: 2.6573  D(x): 0.7623    D(G(z)): 0.1077 / 0.0997
[4/5][900/1029] Loss_D: 0.3516  Loss_G: 3.3966  D(x): 0.8571    D(G(z)): 0.1552 / 0.0534
[4/5][950/1029] Loss_D: 0.3590  Loss_G: 1.9396  D(x): 0.8161    D(G(z)): 0.1201 / 0.2004
[4/5][1000/1029]        Loss_D: 0.4046  Loss_G: 2.8247  D(x): 0.7640    D(G(z)): 0.0929 / 0.0873
```

Result after training 13 epochs (batch_size = 192):





[epoch][batch]

```
[4/5][0/686]    Loss_D: 0.5964  Loss_G: 2.0879  D(x): 0.6275    D(G(z)): 0.0403 / 0.1799
[4/5][50/686]   Loss_D: 0.5315  Loss_G: 2.7125  D(x): 0.6512    D(G(z)): 0.0207 / 0.1002
[4/5][100/686]  Loss_D: 0.4604  Loss_G: 3.6943  D(x): 0.9052    D(G(z)): 0.2751 / 0.0341
[4/5][150/686]  Loss_D: 0.4030  Loss_G: 2.4925  D(x): 0.7786    D(G(z)): 0.1121 / 0.1121
[4/5][200/686]  Loss_D: 0.4525  Loss_G: 3.6135  D(x): 0.8947    D(G(z)): 0.2576 / 0.0383
[4/5][250/686]  Loss_D: 0.4211  Loss_G: 3.3283  D(x): 0.8183    D(G(z)): 0.1695 / 0.0555
[4/5][300/686]  Loss_D: 0.4935  Loss_G: 2.5595  D(x): 0.7077    D(G(z)): 0.0846 / 0.1137
[4/5][350/686]  Loss_D: 0.4792  Loss_G: 3.4017  D(x): 0.8853    D(G(z)): 0.2654 / 0.0491
[4/5][400/686]  Loss_D: 0.4262  Loss_G: 2.9709  D(x): 0.7978    D(G(z)): 0.1464 / 0.0770
[4/5][450/686]  Loss_D: 0.8188  Loss_G: 5.7189  D(x): 0.9484    D(G(z)): 0.4885 / 0.0048
[4/5][500/686]  Loss_D: 0.5241  Loss_G: 3.9172  D(x): 0.8552    D(G(z)): 0.2464 / 0.0319
[4/5][550/686]  Loss_D: 0.7302  Loss_G: 1.9036  D(x): 0.5550    D(G(z)): 0.0385 / 0.2015
[4/5][600/686]  Loss_D: 0.6300  Loss_G: 3.6013  D(x): 0.8568    D(G(z)): 0.3307 / 0.0408
[4/5][650/686]  Loss_D: 0.4196  Loss_G: 3.6096  D(x): 0.8154    D(G(z)): 0.1592 / 0.0392
```

Training with only cat faces

Result after training 40 epochs (batch_size = 128):

[epoch][batch]

```
[38/40][100/369]        Loss_D: 0.2954  Loss_G: 3.9450  D(x): 0.9211    D(G(z)): 0.1732 / 0.0273
[38/40][150/369]        Loss_D: 0.2896  Loss_G: 2.3724  D(x): 0.8100    D(G(z)): 0.0583 / 0.1325
[38/40][200/369]        Loss_D: 0.9080  Loss_G: 5.7011  D(x): 0.9849    D(G(z)): 0.5242 / 0.0065
[38/40][250/369]        Loss_D: 0.5304  Loss_G: 2.6349  D(x): 0.7852    D(G(z)): 0.2088 / 0.1038
[38/40][300/369]        Loss_D: 0.2672  Loss_G: 2.9715  D(x): 0.8324    D(G(z)): 0.0640 / 0.0769
[38/40][350/369]        Loss_D: 0.2863  Loss_G: 2.9851  D(x): 0.8937    D(G(z)): 0.1435 / 0.0716
[39/40][0/369]  Loss_D: 0.6026  Loss_G: 4.9078  D(x): 0.9560    D(G(z)): 0.3759 / 0.0126
[39/40][50/369] Loss_D: 0.2809  Loss_G: 3.4994  D(x): 0.9273    D(G(z)): 0.1714 / 0.0441
[39/40][100/369]        Loss_D: 0.3707  Loss_G: 3.3075  D(x): 0.9580    D(G(z)): 0.2421 / 0.0557
[39/40][150/369]        Loss_D: 1.7849  Loss_G: 1.1449  D(x): 0.2757    D(G(z)): 0.0544 / 0.3819
[39/40][200/369]        Loss_D: 0.5901  Loss_G: 2.7769  D(x): 0.8115    D(G(z)): 0.2583 / 0.0979
[39/40][250/369]        Loss_D: 0.2255  Loss_G: 3.7976  D(x): 0.9129    D(G(z)): 0.1159 / 0.0331
[39/40][300/369]        Loss_D: 0.5694  Loss_G: 1.4661  D(x): 0.6335    D(G(z)): 0.0473 / 0.2837
[39/40][350/369]        Loss_D: 0.4062  Loss_G: 3.1213  D(x): 0.8845    D(G(z)): 0.2240 / 0.0630
```