ADLxMLDS HW4 GAN NETWORK

B03705013 資管四 徐逸然

Model description(2%)

GAN 的模型參考此篇 github : https://github.com/pavitrakumar78/Anime-Face-GAN-Keras

由於期末沒有很多時間調 hyperparameter，因此我嘗試先以此篇的模型為基礎並針對此次作業做修改(如改成 ACGAN)。下圖是我的 generator 的部分。我的生成器有三個 input，分別是 100 維的 noise、hair label 與 eyes label。後兩個 input 分別 embedding 成 5 維的矩陣後 concatenate 在 noise 後面。接著開始做 Conv2DTranspose，從(1,1,512) -> (4,4,256) -> (8,8,128) -> (16,16,64) -> (32,32,64) -> (64,64,3)。最後根據根據 Paper 中的方法採用 tanh activation。Optimizers 的設定則根據助教的 PPT。

```
gen_input = Input(shape = noise_shape) #if want to directly use with conv layer next

hair_input = Input(shape = (1,))
hair = Embedding(hair_len, 5)(hair_input)
hair_reshape = Reshape((1,1,5), input_shape=(1,5))(hair)

eyes_input = Input(shape = (1,))
eyes = Embedding(eyes_len, 5)(eyes_input)
eyes_reshape = Reshape((1,1,5), input_shape=(1,5))(eyes)

concat = concatenate([gen_input, hair_reshape, eyes_reshape])

generator = Conv2DTranspose(filters = 512, kernel_size = (4,4), strides = (1,1), padding = "valid", kernel_initializer = kernel_init)(concat)
generator = BatchNormalization(momentum = 0.5)(generator)
generator = LeakyReLU(0.2)(generator)

generator = Conv2DTranspose(filters = 256, kernel_size = (4,4), strides = (2,2), padding = "same", kernel_initializer = kernel_init)(generator)
generator = BatchNormalization(momentum = 0.5)(generator)
generator = LeakyReLU(0.2)(generator)

generator = Conv2DTranspose(filters = 128, kernel_size = (4,4), strides = (2,2), padding = "same", kernel_initializer = kernel_init)(generator)
generator = BatchNormalization(momentum = 0.5)(generator)
generator = LeakyReLU(0.2)(generator)

generator = Conv2DTranspose(filters = 64, kernel_size = (4,4), strides = (2,2), padding = "same", kernel_initializer = kernel_init)(generator)
generator = BatchNormalization(momentum = 0.5)(generator)
generator = LeakyReLU(0.2)(generator)

generator = Conv2DTranspose(filters = 64, kernel_size = (3,3), strides = (1,1), padding = "same", kernel_initializer = kernel_init)(generator)
generator = BatchNormalization(momentum = 0.5)(generator)
generator = LeakyReLU(0.2)(generator)

generator = Conv2DTranspose(filters = 3, kernel_size = (4,4), strides = (2,2), padding = "same", kernel_initializer = kernel_init)(generator)
generator = Activation('tanh')(generator)

gen_opt = Adam(lr=0.0002, beta_1=0.5)
generator_model = Model(input = [gen_input, hair_input, eyes_input], output = generator)
generator_model.compile(loss='binary_crossentropy', optimizer=gen_opt, metrics=['accuracy'])
```

Discriminator 的模型則是傳統的 CNN，但採用跟 Paper 一樣的做法不使用 maxpooling(2,2)

而是設定 stride = (2,2)。ACGAN 除了判斷照片是否為 fake 之外還需要能 predict 出正確的 class label，因此 Discriminator 的 output 有三個，分別使用 sigmoid(判斷照片真偽), softmax(判斷 hair label)與 softmax(判斷 eyes label) activation。Optimizers 的設定則與 generator 相同。

```
kernel_init = 'glorot_uniform'

dis_input = Input(shape = image_shape)

discriminator = Conv2D(filters = 64, kernel_size = (3,3), strides = (2,2), padding = "same", kernel_initializer = kernel_init)(dis_input)
discriminator = LeakyReLU(0.2)(discriminator)

discriminator = Conv2D(filters = 128, kernel_size = (3,3), strides = (2,2), padding = "same", kernel_initializer = kernel_init)(discriminator)
discriminator = BatchNormalization(momentum = 0.5)(discriminator)
discriminator = LeakyReLU(0.2)(discriminator)

discriminator = Conv2D(filters = 256, kernel_size = (3,3), strides = (2,2), padding = "same", kernel_initializer = kernel_init)(discriminator)
discriminator = BatchNormalization(momentum = 0.5)(discriminator)
discriminator = LeakyReLU(0.2)(discriminator)

discriminator = Conv2D(filters = 512, kernel_size = (3,3), strides = (2,2), padding = "same", kernel_initializer = kernel_init)(discriminator)
discriminator = BatchNormalization(momentum = 0.5)(discriminator)
discriminator = LeakyReLU(0.2)(discriminator)

discriminator = Conv2D(filters = 1024, kernel_size = (3,3), strides = (2,2), padding = "same", kernel_initializer = kernel_init)(discriminator)
discriminator = BatchNormalization(momentum = 0.5)(discriminator)
discriminator = LeakyReLU(0.2)(discriminator)

discriminator = Flatten()(discriminator)

img = Dense(1)(discriminator)
img = Activation('sigmoid')(img)

hair_output = Dense(hair_len)(discriminator)
hair_output = Activation('softmax')(hair_output)

eyes_output = Dense(eyes_len)(discriminator)
eyes_output = Activation('softmax')(eyes_output)

dis_opt = Adam(lr=0.0002, beta_1=0.5)
discriminator_model = Model(input = dis_input, output = [img, hair_output, eyes_output])
discriminator_model.compile(loss=['binary_crossentropy', 'categorical_crossentropy', 'categorical_crossentropy'],
                            optimizer=dis_opt,
                            metrics=['accuracy'])
```

ACGAN 的 objextive function 與作業 slide 上的相同，分為 Ls 與 Lc 兩個部分，分別是 Source 的 Log Likelihood 與 Class 的 Log Likelihood。Discriminator 會嘗試最大化 Ls+Lc，Generator 則是最小化 Lc-Ls。

$$L_S = E[\log P(S = real \mid X_{real})] + E[\log P(S = fake \mid X_{fake})]$$

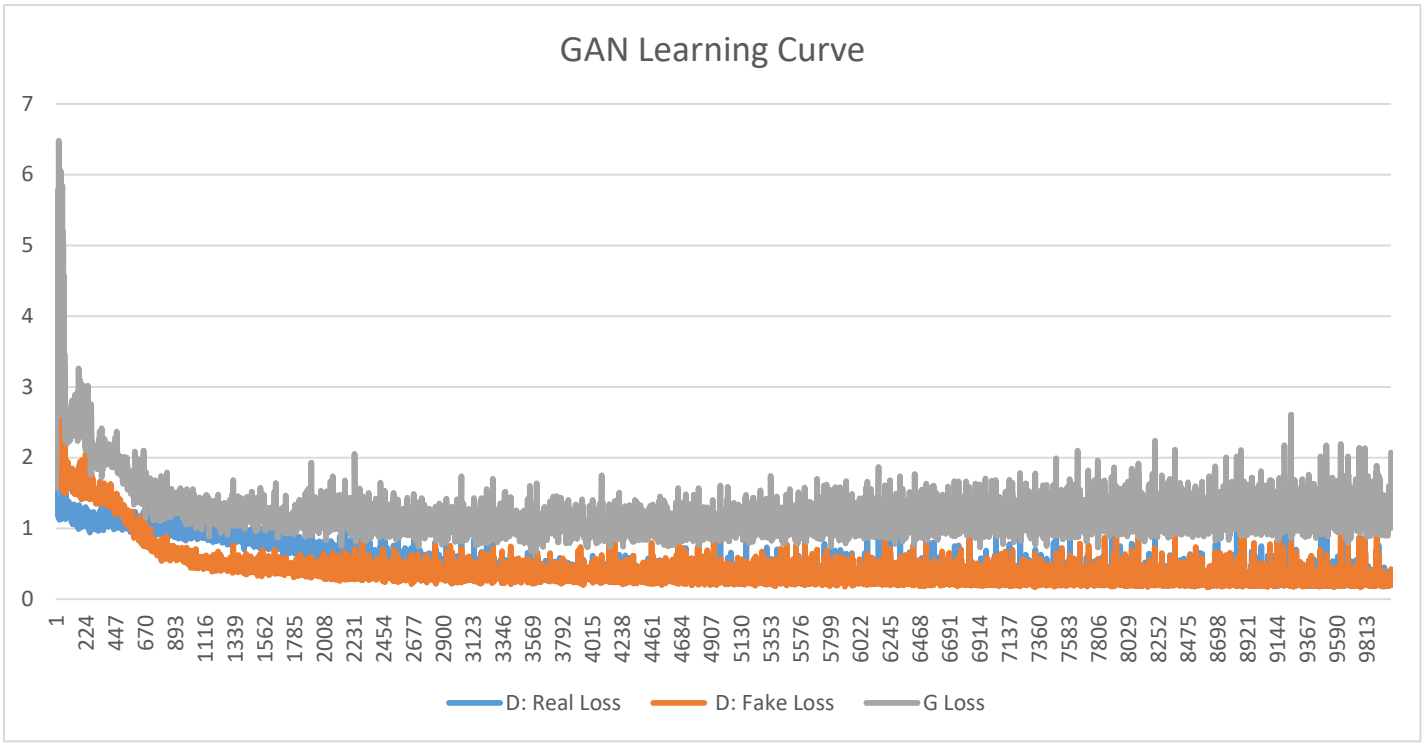
$$L_C = E[\log P(C = c \mid X_{real})] + E[\log P(C = c \mid X_{fake})]$$

How do you improve your performance (2%)

1. Feature Embedding：原本採用 one-hot 的方式將 hair 與 eyes 直接 concat 在 noise 後面，後來參考幾篇 github 後決定採用 embedding。Embedding 的好處是相近的顏色應該會 embedding 成相近的 vector，而不是完全正交的兩個 one-hot vector。
2. Smooth Label：參考此篇 https://github.com/soumith/ganhacks，GAN 之所以不太穩定有可能是一位 gererate 出來的 distribution 其實很怪，或是甚至與真實 data 的 distribution 沒有重疊，因此做 label smoothing 可以使 network 不要 predict 出很極端的值。
3. Normalize 照片至[1,-1]區間：按照 paper 的方法，generator 的 output 應該採用 tanh activation，因此應該將照片 map 到[1, -1]區間。

Experiment settings and observation (2%)

我將 learning curve 繪製成下圖。可以發現 Discriminator 前期針對真實照片的分辨能力較高，而對於假的照片要到約 900 個 step 以後才顯著下降。我自己認為合理的 gan 模型中，discriminator 應該要稍微比 generator 強一點，才可以發揮類似老師的作用，引導 generator 生出更接近真實的照片，從圖中可以看見 generator 的 loss 永遠都是比 discriminator 高的，也代表每一個 step 由於 discriminator 都會進步一些，generator 訓練後即使變強了，也沒辦法騙過 discriminator，loss 才會維持再 1-2 區間，甚至有小幅上升的趨勢。本次作業我認為最困難的部分是 generator 與 discriminator 強度的調整，要使他們保持平衡不太容易，網路上有多篇的 github 也提到 gan 模型對參數非常敏感，可能多疊個一層 CNN 就會讓整個訓練過程壞掉。未來有時間的話希望能夠嘗試實作 WGAN，並比較其與 ACGAN 之差異。


GAN Learning Curve
7
6
5
4
3
2
1
0
1
224
447
670
893
1116
1339
1562
1785
2008
2231
2454
2677
2900
3123
3346
3569
3792
4015
4238
4461
4684
4907
5130
5353
5576
5799
6022
6245
6468
6691
6914
7137
7360
7583
7806
8029
8252
8475
8698
8921
9144
9367
9590
9813
D: Real Loss
D: Fake Loss
G Loss