Model Description

- **Image Generation (2%)**

  Generator部分，透過好幾層的ConvTranspose2d，及BatchNorm2d，activation funciton是ReLU和Tanh。

  generator的objective function: $V=E_{x\sim Pgenerator}[-log(D(x))]$

```python
class NetG(nn.Module):
    def __init__(self, args):
        super(NetG, self).__init__()
        ngf = args.ngf  # 生成器feature map数

        self.main = nn.Sequential(
            # 输入是一个nz维度的噪声，我们可以认为它是一个1*1*nz的feature map
            nn.ConvTranspose2d(args.nz, ngf * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),
            # 上一步的输出形状：(ngf*8) x 4 x 4

            nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            # 上一步的输出形状： (ngf*4) x 8 x 8

            nn.ConvTranspose2d(ngf * 4, ngf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            # 上一步的输出形状： (ngf*2) x 16 x 16

            nn.ConvTranspose2d(ngf * 2, ngf, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),
            # 上一步的输出形状：(ngf) x 32 x 32

            nn.ConvTranspose2d(ngf, 3, 4, 2, 1, bias=False),
            nn.Tanh()  # 输出范围 -1~1 故而采用Tanh
            # 输出形状：3 x 64 x 64
        )

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

  Discriminator部分，透過好幾層的Conv2d，及BatchNorm2d，activation function LeakyReLU和Sigmoid。

  discriminator的objective function: $V=E_{x\sim Pdata}[logD(x)]+E_{x\sim Pgenerator}[log(1-D(x))]$

```python
class NetD(nn.Module):
    def __init__(self, args):
        super(NetD, self).__init__()
        ndf = args.ndf
        self.main = nn.Sequential(
            # 输入 3 x 64 x 64
            nn.Conv2d(3, ndf, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            # 输出 (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),
            # 输出 (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),
            # 输出 (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),
            # 输出 (ndf*8) x 4 x 4

            nn.Conv2d(ndf * 8, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()  # 输出一个数(概率)
        )

    def forward(self, input):
        return self.main(input).view(-1)
```

- **Text-to-image Generation (2%)**

Generator部分，noise的維度是100，condition的部分使用one-hot encoding，髮色共有12種，眼睛則有10種，分別用了12維和10維的向量。將髮色、眼睛和noise concat起來可得到122維向量，model會將其視為122 * 1 * 1，即大小1 * 1，且共122個channel的圖片。將這樣的圖片過Transpose CNN不斷升維最後得到3 * 64 * 64的RGB圖片。Transpose CNN的架構如下:

```
nn.ConvTranspose2d(in_channels=122, out_channels=512, kernel_size=4, stride=1, padding=0, bias=False),
nn.BatchNorm2d(num_features=512),
nn.ReLU(inplace=True),
# output: (batch, 512, 4, 4)

nn.ConvTranspose2d(in_channels=512, out_channels=256, kernel_size=4, stride=2, padding=1, bias=False),
nn.BatchNorm2d(num_features=256),
nn.ReLU(inplace=True),
# output: (batch, 256, 8, 8)

nn.ConvTranspose2d(in_channels=256, out_channels=128, kernel_size=4, stride=2, padding=1, bias=False),
nn.BatchNorm2d(num_features=128),
nn.ReLU(inplace=True),
# output: (batch, 128, 16, 16)

nn.ConvTranspose2d(in_channels=128, out_channels=64, kernel_size=4, stride=2, padding=1, bias=False),
nn.BatchNorm2d(num_features=64),
nn.ReLU(inplace=True),
# output: (batch, 64, 32, 32)

nn.ConvTranspose2d(in_channels=64, out_channels=3, kernel_size=4, stride=2, padding=1, bias=False),
nn.Tanh()
# output: (batch, 3, 64, 64)
```

Discriminator部分，會先用一個傳統的CNN將RGB圖片降維，CNN架構如下:

```
nn.Conv2d(in_channels=3, out_channels=64, kernel_size=4, stride=2, padding=1, bias=False),
nn.LeakyReLU(negative_slope=0.2, inplace=True),
# output: (batch, 64, 32, 32)

nn.Conv2d(in_channels=64, out_channels=128, kernel_size=4, stride=2, padding=1, bias=False),
nn.BatchNorm2d(num_features=128),
nn.LeakyReLU(negative_slope=0.2, inplace=True),
# output: (batch, 128, 16, 16)

nn.Conv2d(in_channels=128, out_channels=256, kernel_size=4, stride=2, padding=1, bias=False),
nn.BatchNorm2d(num_features=256),
nn.LeakyReLU(negative_slope=0.2, inplace=True),
# output: (batch, 256, 8, 8)

nn.Conv2d(in_channels=256, out_channels=512, kernel_size=4, stride=2, padding=1, bias=False),
nn.BatchNorm2d(num_features=512),
nn.LeakyReLU(negative_slope=0.2, inplace=True),
# output: (batch, 512, 4, 4)

nn.Conv2d(in_channels=512, out_channels=22, kernel_size=4, stride=1, padding=0, bias=False),
# output: (batch, 22, 1, 1)
```

3 * 64 * 64的的圖片會被降維成22 * 1 * 1的圖片，將其reshape可作為22維的feature。我目標期待這22維能代表每一個髮色、每一個眼睛顏色被activate的程度。因此discriminator將圖片降到22維後，會將這22維與該圖片真正的髮色、眼色做similarity。也就是說每張圖片根據其真正的髮色(12維one-hot)、眼色(10維one-hot)會有一個 label (22維，髮色在前，眼色在後)。而discriminator會將圖片被降維的22維向量與label做內積，得到一個分數，代表「正確髮色被activated的程度」與「正確眼色被activate的程度」的加總。最後，此分數再過sigmoid就得到discriminator最終的分數。Train generator和discriminator的loss function都是將最後圖片的分數和label (0或1) 做binary cross entropy。generator的label都是1，因為我們希望model越畫越好。而discriminator則分三種，分別是 (1) real image, correct condition (2) real image, error condition (3) fake image, correct condition。第一種的label為1，後兩者的label為0。

Experiment settings and observation

- **Image Generation (1%)**

  optimizer都是使用Adam。

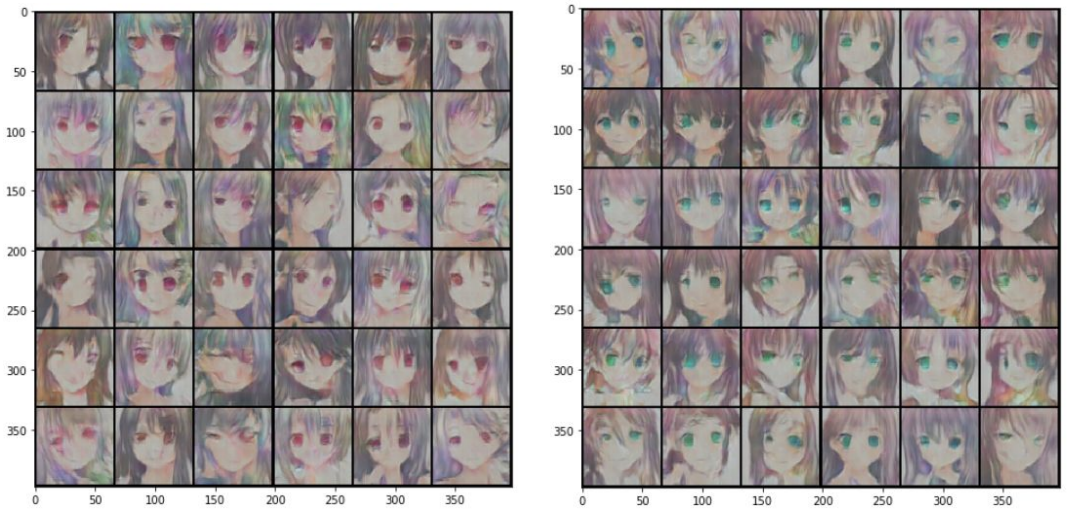
  50 epoch **100 epoch** **109 epoch**

  與Training tips for improvement裡的有tips的case(SGD for discriminator/Adam for generator)和沒有tip 3的case(Adam for discriminator/SGD for generator)相比，(以下用G/D分別代表generator和discriminator)G/D的optimizer皆為Adam的case更快產生較好的圖，其次為(Adam for D/SGD for G)的case，再來才是(SGD for D/Adam for G)。

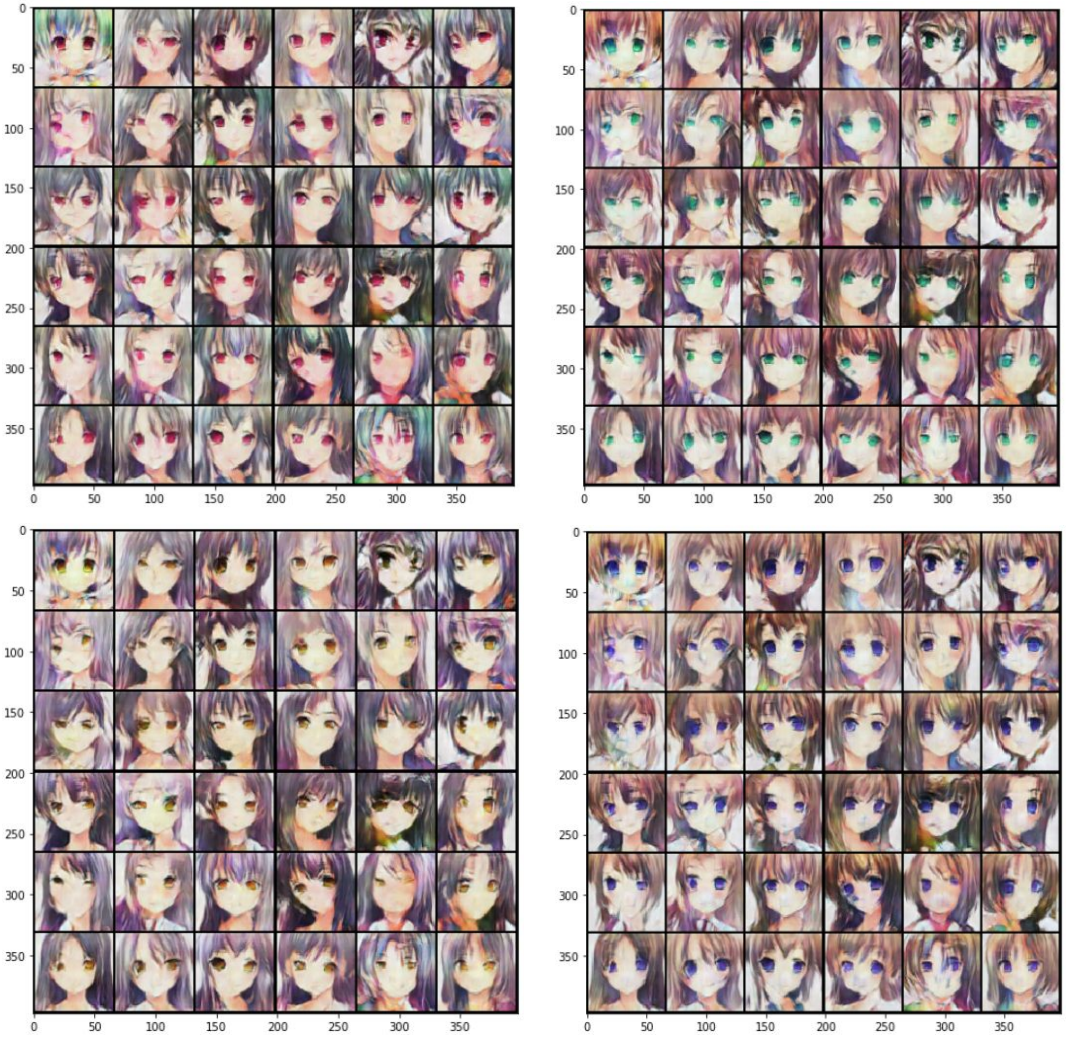
- **Text-to-image Generation (1%)**

  我的conditional GAN並沒有使用RNN來embed文字。因為這個作業的task比較單純，會有變化的就只有髮色和眼色，也沒有順序前後的問題，用RNN這個大工具感覺不會比較好而且會比較難train。因此改直接把髮色當作一個數字，眼色也當作一個數字，各自過embedding layer讓generator和discriminator直接學如何embed顏色 (共四個embedding layers)。但這失敗了，猜測可能這樣的model還是太大不容易學習，加上髮色12種，眼色10種，數量不多之外，顏色之間、髮眼之間也沒有什麼相關性，應該可以直接使用one-hot encodding。改用one-hot encodding後就能有比較合理的輸出了訓練過程使用Adam，結果如下圖: (綠髮紅眼、紅髮綠眼)

  發現眼睛比頭髮更容易train，眼睛的眼色都很明確，而頭髮雖然有些合理的輸出，但還是有滿多色差的。這可能是因為原本在train discriminator時只有使用 (1) real image, correct condition (3) fake image, correct condition 兩種情況。discriminator並沒有學會分辨不同顏色。因此後來加入了 (2) real image, error condition，讓discriminator去

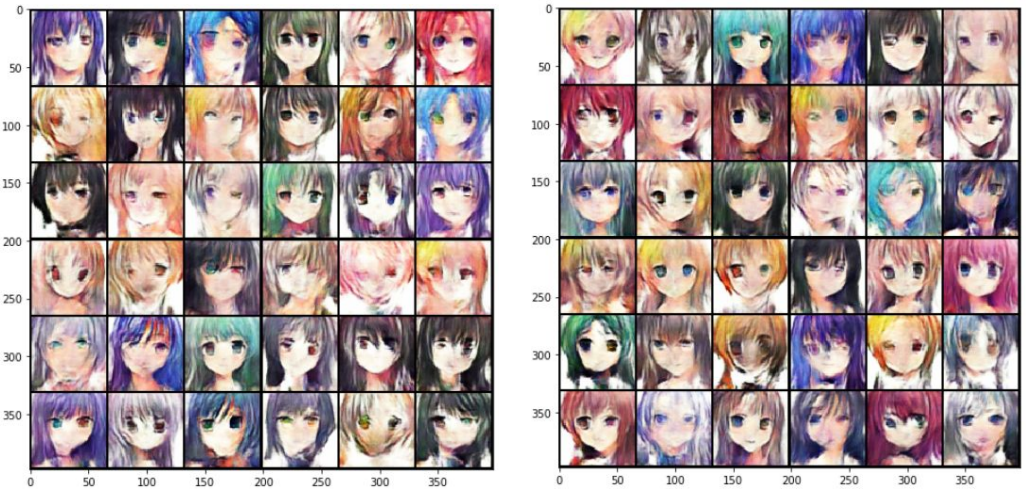
學mismatch的情況。加入mismatch後效果變好很多，不只更能分辨髮色，原本輸出圖片總有些灰灰的缺點也大大改善 (不知道為什麼原本的model在漸漸描出較清晰的人臉輪廓同時，圖片的彩度會漸漸降低)。下圖分別為綠髮紅眼、紅髮綠眼、紫髮橘眼、橘髮紫眼。從圖中可以發現conditional GAN的確是可以操控的，同樣的noise就會畫出同樣的臉型、髮型，再根據指定的condition來上不同顏色。



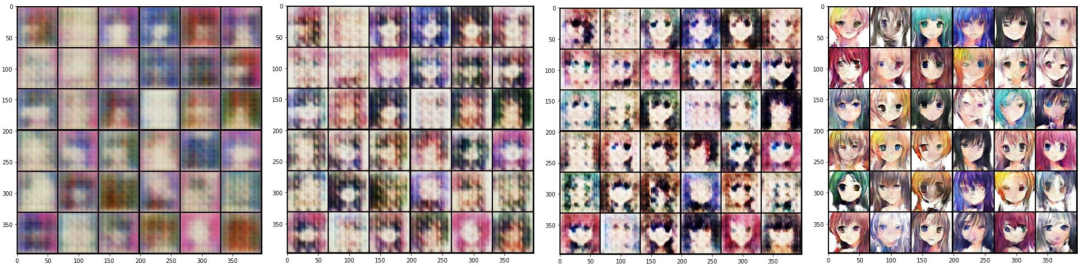
Compare your model with WGAN, WGAN-GP, LSGAN

- **Model Description of the choosed model (1%)**
  我使用WGAN，即discriminator的最後DNN的輸出直接當作分數，不通過sigmoid，並將update過的參數做clip (0.001, -0.001)。使用的model架構與上題類似，只是把原本做binary cross entropy的部分改直接使用discriminator的output。

- **Result of the model (1%)**

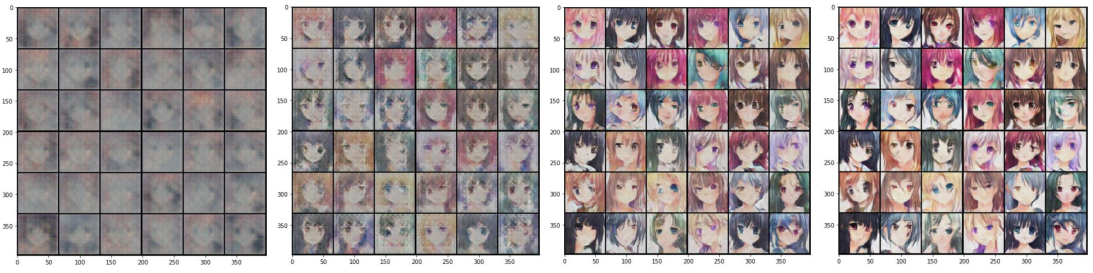


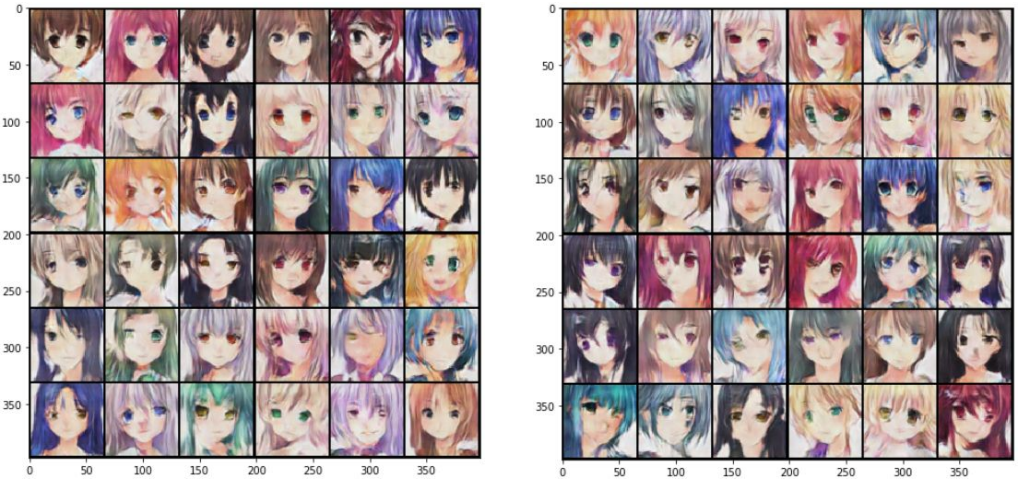
可以看到顏色還滿鮮豔的，但硬傷是臉型相當崩壞.....從下排的訓練過程可以發現，WGAN似乎很擅長顏色的彩度和對比，從一開始訓練輸出的圖片就都很明亮鮮豔，但臉型的輪廓線條卻一直精緻不起來。



- **Comparison Analysis (1%)**

下排是原本使用binary cross entropy的訓練過程與大圖結果:





與WGAN相比，binary cross entropy的訓練過程相對緩和，在線條和用色上都是。可以看到BCE一開始偏灰階，漸漸的描出輪廓然後彩度漸漸提升，訓練過程也比WGAN慢許多。WGAN訓練很快，彩度也很快到位，但用色和線條都相對粗糙。從大圖可以看到BCE的線條柔和許多，不像WGAN有些粗暴。我覺得這和WGAN在update參數的方法很有關係，因為WGAN在gradient大的地方就會直接走超大步 (然後再clip)，不像BCE一直都是小步更新。因此WGAN在顏色和線條上就會有大幅、直接的變動。以最終結果來說BCE是比較好的，WGAN雖然訓練比較快，但到後期卻無法畫出更細緻的東西，輸出的品質會漸漸被BCE超過。

Training tips for improvement

- **Which tip & implement details (1%)**

  tip 1. Normalize the inputs:

  a. normalize the images between -1 and 1

  b. Tanh as the last layer of the generator output

  c. Implementation details

  1.
```
transforms = tv.transforms.Compose([
    tv.transforms.Resize(args.image_size),
    tv.transforms.CenterCrop(args.image_size),
    tv.transforms.ToTensor(),
    tv.transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)) #[0, 1] -> [-1, 1]
    ])
```

  2.
```
        nn.ConvTranspos
        nn.Tanh()  # 输
        # 输出形状：3 x
    )
```

  tip 2. A modified loss function:

  a. In GAN papers, the loss function to optimize G is min(log 1-D), but in practically folks practically use max(log D)

  b. Implementation details

  i.
```
if ii % args.g_every == 0:
    optimizer_g.zero_grad()
    true_labels = Variable(t.ones(real_imgs.size(0)))
    fake_labels = Variable(t.zeros(real_imgs.size(0)))
    noises = Variable(t.randn(real_imgs.size(0), args.nz, 1, 1))
    #noises = Variable(t.rand(real_imgs.size(0), args.nz, 1, 1))
    if args.gpu:
        true_labels = true_labels.cuda()
        fake_labels = fake_labels.cuda()
        noises = noises.cuda()
    fake_imgs = netg(noises)
    output = netd(fake_imgs)
    err_g = criterion(output, true_labels)
    err_g.backward()
    #G_loss = t.mean(t.log(1. - output1))
    #G_loss.backward()
    optimizer_g.step()

    err_g_plot += err_g.data[0] * true_labels.size(0)
    #err_g_plot += G_loss.data[0] * true_labels.size(0)
    total_g += true_labels.size(0)
```

  ii. criterion = torch.nn.BCELoss()

  tip 9. Use the ADAM Optimizer:

  a. Use SGD for discriminator and ADAM for generator

b. Implementation details

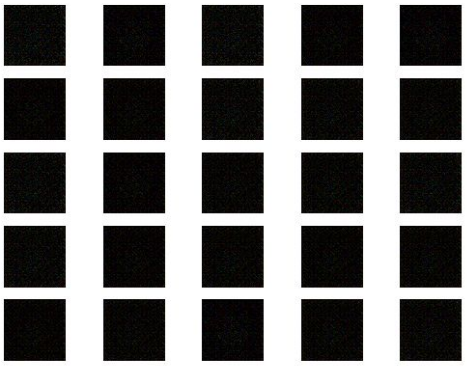
i.

```
optimizer_g = t.optim.Adam(netg.parameters(), args.lr1, betas=(args.beta1, 0.999))
#optimizer_g = t.optim.SGD(netg.parameters(), args.lr1)
#optimizer_d = t.optim.Adam(netd.parameters(), args.lr2, betas=(args.beta1, 0.999))
optimizer_d = t.optim.SGD(netd.parameters(), args.lr2)
```

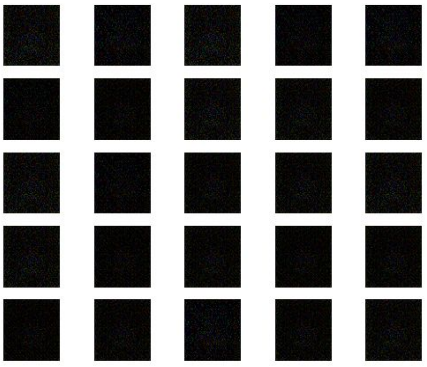
- **Result and Analysis (1%)**

有tips:

50-th epoch:

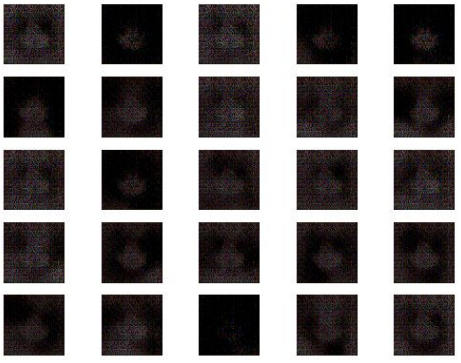


100-th epoch:



150-th epoch:

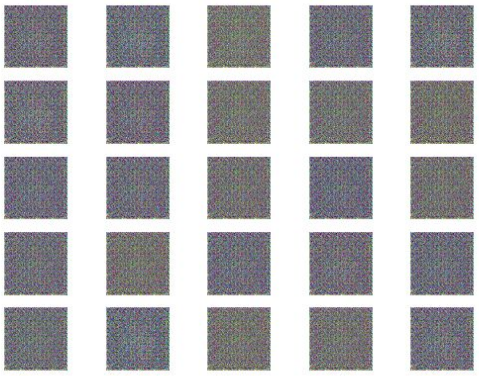
200-th epoch:

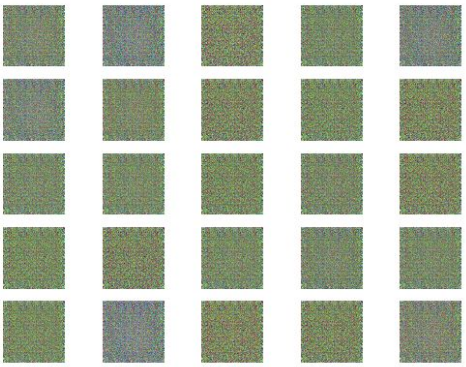


沒有tip 1:

50-th epoch:

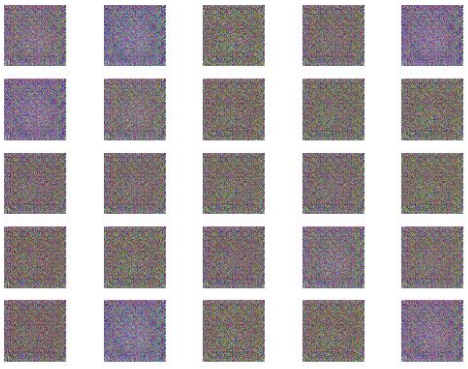
100-th epoch:

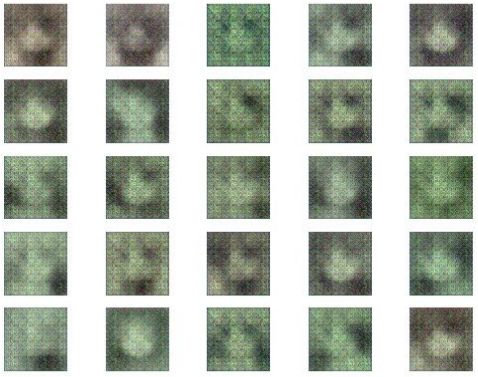




150-th epoch:

200-th epoch:



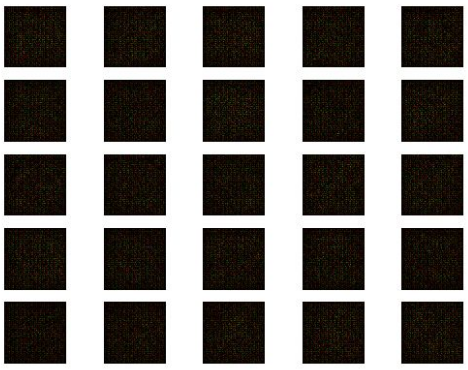


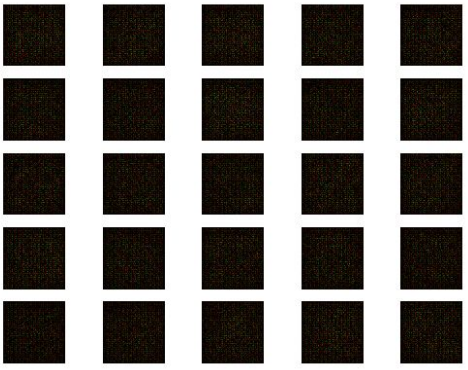
analysis:沒有tip 1的情況下，會從一個充滿雜訊的地方開始產生圖。

沒有tip 2:

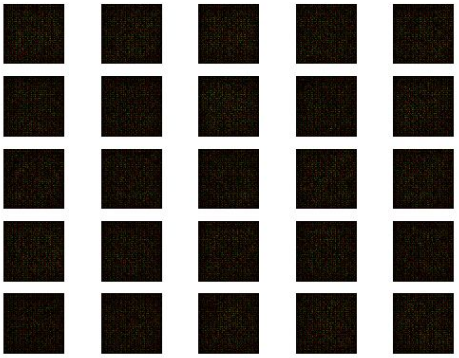
50-th epoch:

100-th epoch:

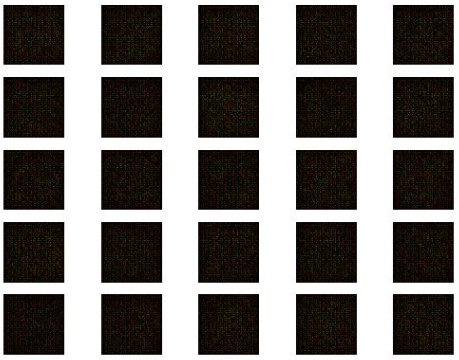




150-th epoch:



200-th epoch:

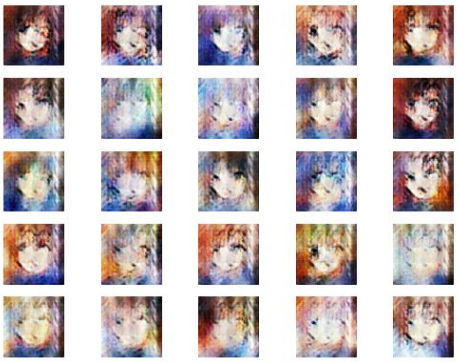


analysis:沒有tips 2的情況下，很難把圖顯示出來

沒有tip 9: (我把Adam用在discriminator、SGD用在generator)

50-th epoch:

100-th epoch:



150-th epoch:

200-th epoch:

analysis:我把optimizer反過來給generator和discriminator，反而圖更快顯示出來。

Style Transfer (2%)

- **show your result (1%)**

左一：原图；左二：黑头发；左三：金头发；左四：褐头发；左五；老年；左六：性别转换

- **Analysis (1%)**
  我们使用StarGAN，对人像进行五个domain的同时转换。模型拥有五个转换类别，input size为256x256。我们经观察发现左三、左五效果普遍优良，而左二、左四在脸部的表现比较糟糕，因此分析在训练GAN时数据集的质量与多样性会影响最终生成结果的好坏。同时第三行数据面部较有歪斜，因此产生的数据略有些扭曲，同时发现非面部与头发特征也有改变，因此模型并未完全收敛。

分工表

- b03902125，林映廷
  - 3-1 GAN
  - Report
    - Model Description: Image generation
    - Experiment settings and observation: Image generation
    - Training tips for improvement
- b03902130，楊書文
  - 3-1 WGAN
  - 3-2 Conditional GAN
  - Report
    - Model Description: Conditional GAN

        - Experiment settings and observation: Conditional GAN
        - Compare your model with WGAN
- t06902115，張晉之
    - 3-3 StarGAN
    - Report
        - Style Transfer