

RUN1(DCGAN + SELU):

Model:

根據最近的相關研究指出，SELU 能夠直接取代 BatchNorm + ReLU，同時解決了容易 Gradient Vanish 的問題，且讓模型有更佳的表現。

```
self.main = nn.Sequential(  
    nn.Conv2d(3, d_hidden,  
        kernel_size=4, stride=2, padding=1, bias=False),  
    # --> some regularization and activation  
    nn.SELU(),  
    # shape is (d_hidden, 32, 32),
```

Training:

- (1). 先訓練出一個有判別能力的 **Discriminator**，判斷圖片是真的還是假的。

我們先將真實的資料輸入 Discriminator，為了讓模型把這些圖片都判別成真圖片，我們將模型的輸出跟 1(real_label)算 loss 值並存入 errD_real。

```
# throw real image into discriminator  
output_real = net_d(imgs) ##  
errD_real = criterion(output_real, real_label) ##
```

再隨機生成雜訊輸入模型製作出假圖片，且為了讓模型把這些圖片都判別成假圖片，我們將模型的輸出跟 0(fake_label)算 loss 值並存入 errD_fake。

```
# generate fake image and throw into discriminator  
noise = torch.randn(batch_size, latent_dim, 1, 1, device=device)  
fake_imgs = net_g(noise) |  
output_fake = net_d(fake_imgs) ##  
errD_fake = criterion(output_fake, fake_label) ##
```

然後將兩種 loss 加起來讓模型反向傳播，更新 **Discriminator** 的 weight。

```
# calculate loss  
err_d = errD_real + errD_fake ##  
err_d.backward()  
loss_temp['err_d'].append(err_d.item())  
optim_d.step()
```

現在訓練後的 Discriminator 已經有能力能判別出真圖片及假圖片

- (2). 接下來要訓練 **Generator** 製作出盡可能貼近真實圖片的假圖片，目標是騙過 Discriminator。

我們一樣先將真實的資料輸入 Discriminator，為了讓模型把這些圖片都判別成真圖片，我們將模型的輸出跟 1(real_label)算 loss 值並存入 errD_real。

```
# throw real image into discriminator|
output_real = net_d(imgs) ##
errD_real = criterion(output_real, real_label)
```

然後再將雜訊輸入 Generator 生成假圖片，並將假圖片也輸入 Discriminator 輸出假圖片的 label，不同的地方在於，**假圖片的 label 要跟 1(real_label)算 loss**，也就是我們希望 Discriminator 判斷 Generator 輸出的假圖片的值，要盡可能的接近 1(real_label)。最後將 loss 值存入 errD_fake。

```
# throw fake image into discriminator
noise = torch.randn(batch_size, latent_dim, 1, 1, device=device) #
fake_imgs = net_g(noise) #
output_fake = net_d(fake_imgs) ##
errD_fake = criterion(output_fake, real_label) ## computing G's loss using real labels
```

一樣將兩種 loss 加起來讓模型反向傳播，這次要更新 **Generator** 的 weight。

```
err_g = errD_real + errD_fake
err_g.backward()
loss_temp['err_g'].append(err_g.item())
optim_g.step()
```

理論上在兩個 Discriminator 跟 Generator 的不斷競爭下，Discriminator 會為了要判別出 Generator 生成的假圖片而別的越來越強，Generator 會為了要騙過 Discriminator 生成越來越接近真實的圖片。

雜訊進化史：



大概訓練到第三次後就有個還不錯的效果了



但是隨著訓練次數增多，模型的效果並不會越來越好，並且生成的圖片的差異性會越來越低，訓練到 30 次的時候可以發現，每次輸出的圖片基本上都變成了同一種臉。

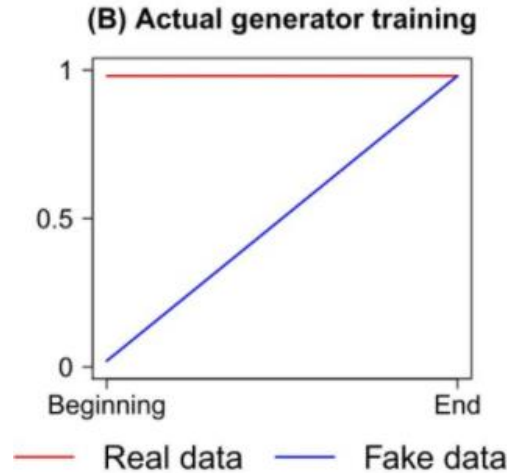
在經過查詢相關資料後，我認為應該是發生了模式坍塌 (Mode collapse)，也就是當 Generator 認為它可以透過鎖定單個模式來欺騙 Discriminator 時，就會發生這種情況。也就是說，生成器僅會從這種模式來生成樣本。Discriminator 最終會發現這種模式下的樣本是假的。但 Generator 就只會轉而鎖定到另一種模式。這個循環無限重複，從根本上限制了生成樣本的多樣性。









RUN2(Relativistic GAN + SELU):

Concept:

在 DCGAN，我們訓練 Discriminator 時，希望達到所有 $D(\text{img_real})=1$, $D(\text{img_fake})=0$ ，訓練 Generator 則希望達到 $D(\text{img_real})=D(\text{img_fake})=1$ ，如下：



而 RGAN 則加入的 Relativistic(相對)的概念，以下圖做舉例，Real 是麵包，Fake 是柯基， $C(X)$ 越大代表是麵包的機率越大。

| Scenario | Absolute probability (Standard GAN) | Relative probability (Relativistic average Standard GAN) |
|---|---|--|
| Real image looks real and fake images look fake |  $C(x_r) = 8$ $P(x_r \text{ is bread}) = 1$ |  $\bar{C}(x_f) = -5$ $P(x_r \text{ is bread} \bar{C}(x_f)) = 1$ |
| Real image looks real but fake images look similarly real on average |  $C(x_r) = 8$ $P(x_r \text{ is bread}) = 1$ |  $\bar{C}(x_f) = 7$ $P(x_r \text{ is bread} \bar{C}(x_f)) = .73$ |
| Real image looks fake but fake images look more fake on average |  $C(x_r) = -3$ $P(x_r \text{ is bread}) = .05$ |  $\bar{C}(x_f) = -5$ $P(x_r \text{ is bread} \bar{C}(x_f)) = .88$ |

圖中舉例三種情況：

(1)真的麵包 \leftrightarrow 真的柯基：

這種情況鑑別度很明顯，
所以 $P(X_r \text{ 是麵包} | X_f)=1$

(2)真的麵包 \leftrightarrow 柯基屁股（很像麵包）：

這種情況的鑑別就相對沒這麼明顯，所
以 $P(X_r \text{ 是麵包} | X_f)=0.73$ 有所下降

(3)像狗的面包 \leftrightarrow 真的柯基：

相對的鑑別度也不明顯，
 $P(X_r \text{ 是麵包} | X_f)=0.88$

而 RGAN 的想法就是，Discriminator 在衡量樣本的真實性的時候應該要同時利用 real data 和 fake data，衡量的是相對為真假的機率而不是絕對的真假。也就是第三個例子中，像狗的麵包雖然是 1 的機率較低，但是還是比柯基是 1 的機率高。

RGAN 加入這個概念後，讓 **Generator** 生成的樣本能夠影響 **Discriminator** 的判斷，讓 **real data** 相對 **fake data** 沒有真實太多，也就是 $D(\text{img_fake})$ 是 1 的機率上升的時候， $D(\text{img_real})$ 是 1 的機率則要相對應下降，這個也就是相對的真假。而實現的方式則很簡單，在算 **Loss** 的時候直接將兩種機率做相減即可，如下圖公式：

$$L_D^{RGAN*} = \mathbb{E}_{(x_r, x_f) \sim (\mathbb{P}, \mathbb{Q})} [f_1(C(x_r) - C(x_f))] \quad (14)$$

and

$$L_G^{RGAN*} = \mathbb{E}_{(x_r, x_f) \sim (\mathbb{P}, \mathbb{Q})} [f_1(C(x_f) - C(x_r))] . \quad (15)$$

Algorithm 1 shows how to train RGANs of this form.

Training:

```
# ===== Update Discriminator =====
# ----- You Should Modify -----
net_d.zero_grad()

# throw real image into discriminator
output_real = net_d(imgs) ##

# generate fake image and throw into discriminator
noise = torch.randn(batch_size, latent_dim, 1, 1, device=device)
fake_imgs = net_g(noise)
output_fake = net_d(fake_imgs) ##

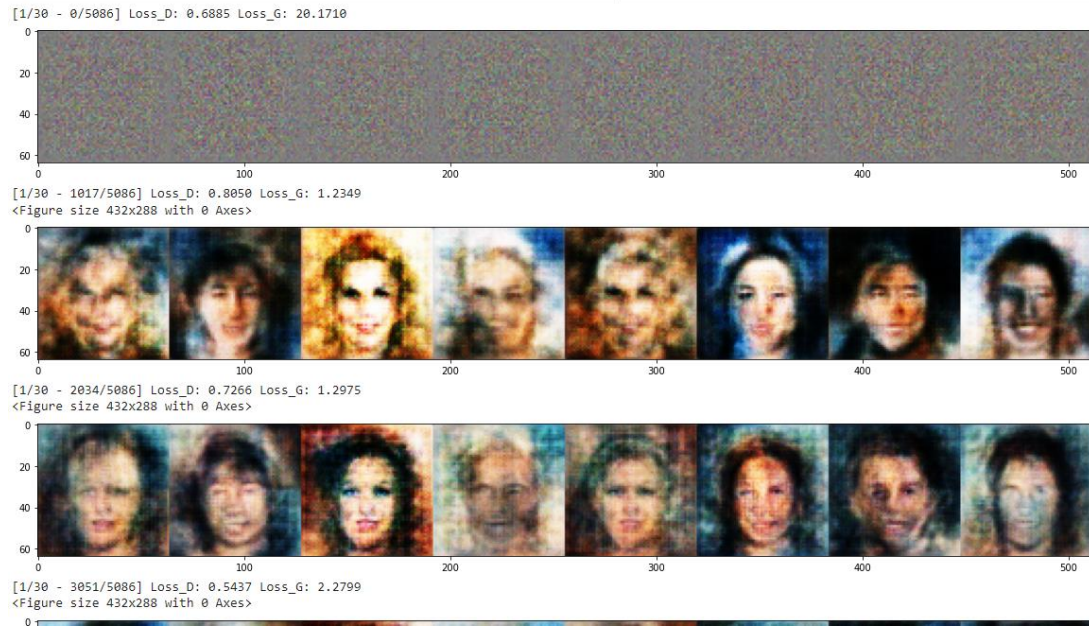
# calculate loss
err_d = criterion(output_real - output_fake, real_label) ##
err_d.backward()
loss_temp['err_d'].append(err_d.item())
optim_d.step()
```

```
# ===== Update Generator =====
# ----- You Should Modify -----
net_g.zero_grad()

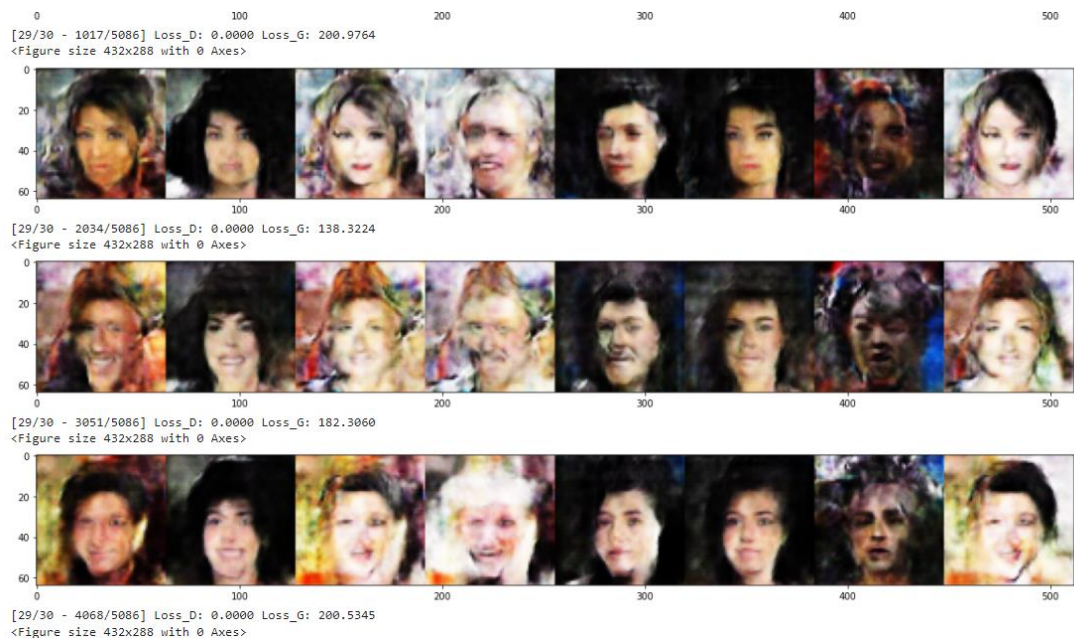
# throw real image into discriminator
output_real = net_d(imgs) ##

# throw fake image into discriminator
noise = torch.randn(batch_size, latent_dim, 1, 1, device=device) #
fake_imgs = net_g(noise) #
output_fake = net_d(fake_imgs) ##
#errD_fake = criterion(output_fake, real_label) ## computing G's loss using real labels

# ----- You Should Modify -----
err_g = criterion(output_fake - output_real, real_label)
err_g.backward()
loss_temp['err_g'].append(err_g.item())
optim_g.step()
```



可以發現隨著訓練次數增加，效果也是逐漸降低，訓練到快 30 次時，Discriminator 已經無法區分出真假圖片，無力幫助 Generator 進步。



RUN3(WGAN + wight clipping):

Algorithm 1 WGAN, our proposed algorithm. All experiments in the paper used the default values $\alpha = 0.00005$, $c = 0.01$, $m = 64$, $n_{\text{critic}} = 5$.

Require: : α , the learning rate. c , the clipping parameter. m , the batch size. n_{critic} , the number of iterations of the critic per generator iteration.

Require: : w_0 , initial critic parameters. θ_0 , initial generator's parameters.

```
1: while  $\theta$  has not converged do
2:   for  $t = 0, \dots, n_{\text{critic}}$  do
3:     Sample  $\{x^{(i)}\}_{i=1}^m \sim \mathbb{P}_r$  a batch from the real data.
4:     Sample  $\{z^{(i)}\}_{i=1}^m \sim p(z)$  a batch of prior samples.
5:      $g_w \leftarrow \nabla_w [\frac{1}{m} \sum_{i=1}^m f_w(x^{(i)}) - \frac{1}{m} \sum_{i=1}^m f_w(g_\theta(z^{(i)}))]$ 
6:      $w \leftarrow w + \alpha \cdot \text{RMSPProp}(w, g_w)$ 
7:      $w \leftarrow \text{clip}(w, -c, c)$ 
8:   end for
9:   Sample  $\{z^{(i)}\}_{i=1}^m \sim p(z)$  a batch of prior samples.
10:   $g_\theta \leftarrow -\nabla_\theta \frac{1}{m} \sum_{i=1}^m f_w(g_\theta(z^{(i)}))$ 
11:   $\theta \leftarrow \theta - \alpha \cdot \text{RMSPProp}(\theta, g_\theta)$ 
12: end while
```

不同於原始 GAN 的 Discriminator 做的是真假二元分類，最後一層要使用 Sigmoid，但是在 WGAN 中的 Discriminator 做的是近似擬合 Wasserstein 距離，屬於回歸任務，所以要把將 Discriminator 最後一層的 Sigmoid 移除。

```
nn.Conv2d(d_hidden * 8, 1,
          kernel_size=4, stride=1, padding=0, bias=False),
#nn.Sigmoid()
```

WGAN 透過 wight clipping 的方式限制參數不會超過一個固定範圍，我們這邊設定邊界值為正負 0.01。且原作者在實驗時發現，使用基於動量的優化演算法(例如 Adam)會造成梯度不穩定的問題，所以我們這邊依照原作者的建議選用 RMSprop，並使用原始論文的 $\text{lr} = 0.00005$

```
#WGAN optimizer(RMSprop instead of Adam)
optim_d = torch.optim.RMSprop(net_d.parameters(),lr=0.00005) ##
optim_g = torch.optim.RMSprop(net_g.parameters(),lr=0.00005) ##
weight_clipping_limit = 0.01 #WGAN clip gradient
```

```
# ===== Update Discriminator =====
# ----- You Should Modify -----

net_d.zero_grad()

# modification: clip param for discriminator
for parm in net_d.parameters():
    parm.data.clamp_(-weight_clipping_limit, weight_clipping_limit) ##
```

為了能夠計算，我們預期假圖片的 label 是-1，真圖片的 label 是 1

```
one = torch.FloatTensor([1]) ##  
minus_one = one * -1 ##
```

且 Discriminator 跟 Generator 的 loss 都不取 log，而是直接將 Discriminator 的輸出做平均後，直接反向傳播。

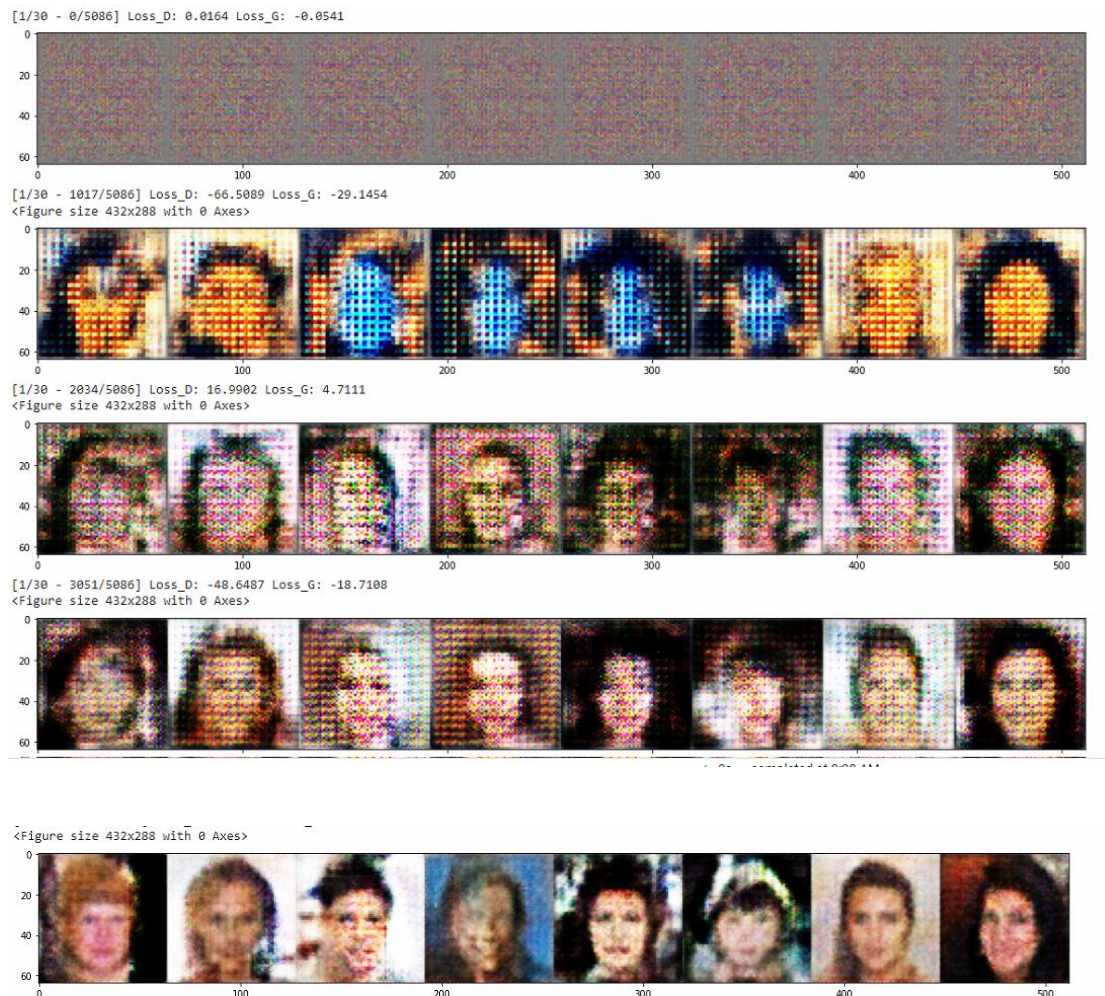
```
# throw real image into discriminator  
output_real = net_d(imgs) ##  
loss_real = output_real.mean(0).view(1)  
loss_real.backward(one) ##
```

```
# generate fake image and throw into discriminator  
noise = torch.randn(batch_size, latent_dim, 1, 1, device=device)  
fake_imgs = net_g(noise)  
output_fake = net_d(fake_imgs) ##  
loss_fake = output_fake.mean(0).view(1)  
loss_fake.backward(minus_one)
```

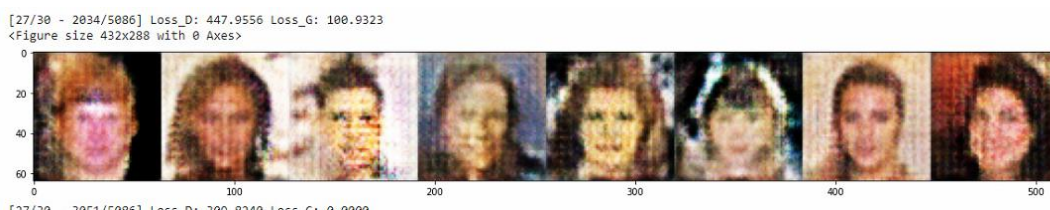
```
optim_d.step()
```

Generator 的部分則是根據原始論文，Discriminator 訓練 5 次 Discriminator 才相對訓練 1 次。由於 WGAN 解決了原始 GAN 的 Discriminator 越好，會導致 Discriminator 的梯度消失的問題，故 WGAN 可以盡量將 Discriminator 訓練的強一點，會有助於 Generator 的效果提升。

```
# ===== Update Generator =====  
# ----- You Should Modify -----  
err_g = torch.FloatTensor([0])  
if (i+1)%5 ==0 or epoch<2:  
    net_g.zero_grad()  
    noise = torch.randn(batch_size, latent_dim, 1, 1, device=device) #  
    fake_imgs = net_g(noise) #  
    output_fake = net_d(fake_imgs) ##  
    loss_fake = output_fake.mean(0).view(1)  
    loss_fake.backward(one)  
    err_g = loss_fake ##  
    loss_temp['err_g'].append(err_g.item())  
    optim_g.step()
```

雖然生成的圖片效果可能也沒有到很好，**WGAN** 基本上解決了模式坍塌(**Mode collapse**)的問題，確保了生成樣本的多樣性。可以發現即使訓練到快 30 次，樣本之間都還是保有一定的差異。不會像之前兩種方法(**Run1&Run2**)訓練到最後，所有的圖片會趨於相近。



RUN4(WGAN + Gradient Penalty):

由於原始 WGAN 採用 wight clipping 的方式強行限制參數，會對模型造成一些不好的效果。後來就有人提出了 gradient penalty 的方法。該方法直接遲罰 Discriminator 判別出假圖片的梯度。

$$L = \underbrace{\mathbb{E}_{\tilde{x} \sim P_g} [D(\tilde{x})] - \mathbb{E}_{x \sim P_r} [D(x)]}_{\text{Original critic loss}} + \underbrace{\lambda \mathbb{E}_{\hat{x} \sim P_{\hat{x}}} [(\|\nabla_{\hat{x}} D(\hat{x})\|_2 - 1)^2]}_{\text{Our gradient penalty}}.$$

利用生成的假圖片跟真圖片算出 **gradient penalty** 後，再併入原本的損失函數。

```
# calculate loss
err_d_real = output_real.mean(0).view(1)
err_d_fake = output_fake.mean(0).view(1)

gradient_penalty = calculate_gradient_penalty(imgs.data, fake_imgs.data)
err_d = err_d_fake - err_d_real + gradient_penalty
err_d.backward()
loss_temp['err_d'].append(err_d)

optim_d.step()
```

由於解決了 **wight clipping** 造成的不穩定的問題，論文建議可以將 **Optimizer** 改回使用 **Adam** 取代原論文的 **RMSPprop**。我這邊則使用改進版的 **AdamW**。

```
optim_d = torch.optim.AdamW(net_d.parameters(), lr=lr, betas=(beta1, 0.99))
optim_g = torch.optim.AdamW(net_g.parameters(), lr=lr, betas=(beta1, 0.99))
```

