# Lab7: Let's Play GANs with Flows and friends

**Hsiang-Chun Yang**
Institute of Multimedia Engineering
National Yang Ming Chiao Tung University
yanghc.cs09g@nctu.edu.tw

## 1 Introduction

這次的 lab 要實作兩種生成式的模型，generative adversarial network (GAN) 以及 normalizing flow (NF)，並用這兩種模型完成以下兩個不同的 task。

### 1.1 Task 1

給定一個隨機產生的 $z$ 向量以及一個 condition 向量，利用 generator 生成指定顏色與形狀的物體，這部分必須使用 conditional 的 GAN 去實作。

### 1.2 Task 2

Task 2 又細分為下列三個項目。

1. 給定一個隨機產生的 $z$ 向量以及一個 condition 向量，利用 normalizing flow 生成具有特定特徵的人臉照片。這個部分必須使用 conditional NL 去實作。
2. 給定兩張人臉照片，對兩張照片的 latent representation 做 linear interpolation，並將內插後的 latent representation 轉回人臉照片，產生兩張照片的合成圖。這個部分的實作可以不使用 conditional NL。
3. 以 smiling 這個特徵為例，首先要找出含有 smiling 的所有人臉照片（positive）以及不包含 smiling 的所有照片（negative），計算這兩群照片的 latent representation 並各自取平均值，得到 $z_{pos}$ 與 $z_{neg}$ 兩個向量，將 $z_{pos} - z_{neg}$ 作為 smiling 這個特徵的向量。再來隨機挑選一張人臉照片，將該照片的 latent representation 加上或減去 smiling vector，並觀察生成的人臉照片的微笑情況。這個部分的實作可以不使用 conditional NL。

## 2 Implementation details

### 2.1 GAN

我選擇的架構是 Conditional DCGAN，DCGAN 的部分主要是依照 Unsupervised Representation Learning With Deep Convolutional Generative Adversarial Networks 這篇論文去實作，generator 中各層的 channel 數量比例直接參考論文中的設計，為 8:4:2:1 ，discriminator 也是採用相同比例唯獨順序顛倒。

#### 2.1.1 Generator

Conditional 的部分是先將輸入的 condition 向量用一個 fully-connected layer 轉換成長度為 $c$ 的向量，再將這個向量接在向量 $z$ 後面，這邊的 $z$ 是一個隨機抽樣產生的向量。圖 1 是 generator 的架構，紅色的部分就是轉換後的 condition 向量。

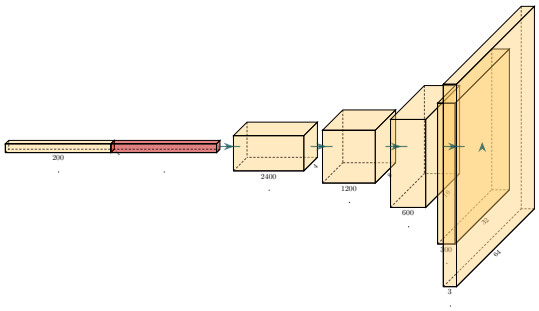



Figure 1: Generator of Conditional DCGAN

下面為 generator 的程式碼。`n_z` 為先前提到的 $z$ 向量的長度，`n_c` 則是 condition 向量經過 fully-connected layer 之後的長度。activation function 的部分保留原本論文中的設定，一樣是用 ReLU 並在最後一層使用 tanh。

```
class Generator(nn.Module):
    def __init__(self, args):
        super(Generator, self).__init__()
        self.n_z = args.n_z
        self.n_c = args.n_c
        n_ch = [args.n_ch_g*8, args.n_ch_g*4, args.n_ch_g*2, args.n_ch_g]
        self.embed_c= nn.Sequential(
            nn.Linear(args.num_conditions, args.n_c),
            nn.ReLU(inplace=True))
        model = [
            nn.ConvTranspose2d(
                args.n_z+args.n_c, n_ch[0], kernel_size=4, stride=2,
                bias=args.add_bias),
            nn.BatchNorm2d(n_ch[0]),
            nn.ReLU(inplace=True)
        ]
        for i in range(1, len(n_ch)):
            model += [
                nn.ConvTranspose2d(
                    n_ch[i-1], n_ch[i], kernel_size=4, stride=2, padding=1,
                    bias=args.add_bias),
                nn.BatchNorm2d(n_ch[i]),
                nn.ReLU(inplace=True)
            ]
        model += [
            nn.ConvTranspose2d(
                n_ch[-1], 3, kernel_size=4, stride=2, padding=1,
                bias=args.add_bias),
            nn.Tanh()
        ]
        self.model = nn.Sequential(*model)

    def forward(self, z, c):
        z = z.reshape(-1, self.n_z, 1, 1)
```

```
        c_embd = self.embed_c(c).reshape(-1, self.n_c, 1, 1)
        x = torch.cat((z, c_embd), dim=1)
        return self.model(x)
```

### 2.1.2 Discriminator

Conditional 的部分是先將輸入的 condition 向量用一個 fully-connected layer 轉換成 $h * w$ 的向量，其中 $h$ 跟 $w$ 是輸入的圖片長寬，再將這個向量調整成跟圖片一樣的形狀，並接到圖片上做為第四個 channel。圖 2 是 discriminator 的架構，紅色的部分就是轉換後的 condition 向量。

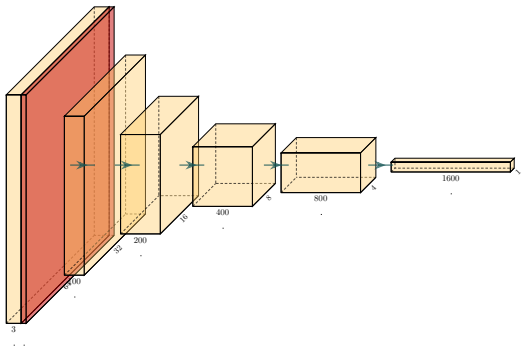



Figure 2: Discriminator of Conditional DCGAN

下面為 discriminator 的程式碼。activation function 的部分保留原本論文中的設定，一樣是用 Leaky ReLU 並在最後一層使用 tanh。

```
class Discriminator(nn.Module):
    def __init__(self, args):
        super(Discriminator, self).__init__()
        self.img_h = args.img_h
        self.img_w = args.img_w
        n_ch = [args.n_ch_d, args.n_ch_d*2, args.n_ch_d*4, args.n_ch_d*8]
        self.embed_c= nn.Sequential(
            nn.Linear(args.num_conditions, args.img_h*args.img_w),
            nn.ReLU(inplace=True))
        model = [
            nn.Conv2d(
                4, n_ch[0], kernel_size=4, stride=2, padding=1,
                bias=args.add_bias),
            nn.BatchNorm2d(n_ch[0]),
            nn.LeakyReLU(0.2, inplace=True)
        ]
        for i in range(1, len(n_ch)):
            model += [
                nn.Conv2d(
                    n_ch[i-1], n_ch[i], kernel_size=4, stride=2, padding=1,
                    bias=args.add_bias),
                nn.BatchNorm2d(n_ch[i]),
                nn.LeakyReLU(0.2, inplace=True)
            ]
```

```
        model += [
            nn.Conv2d(
                n_ch[-1], 1, kernel_size=4, stride=1, padding=0,
                bias=args.add_bias),
            nn.Sigmoid()
        ]
        self.model = nn.Sequential(*model)

    def forward(self, image, c):
        c_embd = self.embed_c(c).reshape(-1, 1, self.img_h, self.img_w)
        x = torch.cat((image, c_embd), dim=1)
        return self.model(x).reshape(-1)
```

### 2.2 NL

我選擇的架構是 Glow 與 Conditional Glow ，整體架構是依照 Glow: Generative Flow with Invertible 1x1 Convolutions 與 Structured Output Learning with Conditional Generative Flows 這兩篇論文去實作，不過由於 Condition Glow 論文中的 condition 與本次作業的形式不同，在 conditioning network 的部分我有自己修改架構。

Glow 與 Conditional Glow 的架構主要的差異就是在 Actnorm、Invertible 1x1 Convolution 與 Affine 中加入了 conditioning network 的設計，下面會針對這三個部份去說明。

#### 2.2.1 (Conditional) Actnorm

在 Glow 中輸入與輸出的關係可以表示成：

$$u_{i,j} = s \odot v_{i,j} + b$$

其中 $u$ 與 $v$ 分別是輸入與輸出，$s$ 為 scale，$b$ 是 bias，這其實就是在輸入與輸出之間做仿射轉換，而 $s$ 與 $b$ 這兩個參數都是直接透過神經網路學習，而 Conditional Glow 則是將其改寫成：

$$s, b = CN(x)$$
$$u_{i,j} = s \odot v_{i,j} + b$$

利用 condition $x$ 與 conditioning network $CN$ 來產生 scale 與 bias。

下面分別是 Actnorm 與 Conditional Actnorm 的程式碼。因為作業的 condition 是以 one-hot vector 的形式表現，所以我把 Conditional Glow 論文中 conditioning network 前半部的 convolution 捨棄，只保留後面 fully-connected layer 的部分。而利用 log scale 是因為 scale 理論上要是大於等於零的數字，但是神經網路計算出來的數字有正有負，所以改成去學習 log scale，並在計算的時候取自然對數 $e^{\log \text{scale}}$。另外因為 scale 本身其實是對角矩陣，在計算行列式的時候是將對角項相乘，而取 log 就等於直接將對角項全部加起來。

```
# Glow
class Actnorm(nn.Module):
    def __init__(self, in_chs):
        super(Actnorm, self).__init__()
        size = [1, in_chs, 1, 1]
        self.bias = nn.Parameter(torch.zeros(size))
        self.log_scale = nn.Parameter(torch.zeros(size))
        self.inited = False
```

```
    def init(self, x):
        if not self.training:
            raise ValueError('In eval() mode, but Actnorm not initialized')
        with torch.no_grad():
            flatten = x.permute(1, 0, 2, 3).reshape(x.shape[1], -1)
            mean = (flatten.mean(1)
                    .unsqueeze(1).unsqueeze(2)
                    .unsqueeze(3).permute(1, 0, 2, 3)
            std = (flatten.std(1)
                    .unsqueeze(1).unsqueeze(2)
                    .unsqueeze(3).permute(1, 0, 2, 3)
            log_scale = torch.log(1/(std+1e-6))
            self.bias.data.copy_(-mean)
            self.log_scale.data.copy_(log_scale)
            self.inited = True

    def forward(self, x):
        if not self.inited:
            self.init(x)
        dims = x.shape[2] * x.shape[3]
        x = x + self.bias
        x = x * torch.exp(self.log_scale)
        dlog_det = torch.sum(self.log_scale) * dims
        return x, dlog_det

    def reverse(self, x):
        x = x * torch.exp(-self.log_scale)
        x = x - self.bias
        return x


# c-Glow
class CondActnorm(nn.Module):
    def __init__(self, in_chs, cond_sz, cond_fc_fts):
        super(CondActnorm, self).__init__()
        self.cond_net = nn.Sequential(
            nn.Linear(cond_sz, cond_fc_fts),
            nn.ReLU(inplace=True),
            nn.Linear(cond_fc_fts, 2*in_chs),
            nn.Tanh())
        self.cond_net[0].weight.data.zero_()
        self.cond_net[0].bias.data.zero_()
        self.cond_net[2].weight.data.zero_()
        self.cond_net[2].bias.data.zero_()

    def forward(self, x, cond):
        cond_b, _, = cond.shape
        cond = self.cond_net(cond).reshape(cond_b, -1, 1, 1)
        log_scale, bias = cond.chunk(2, dim=1)
```

```
        dims = x.shape[2] * x.shape[3]
        x = (x+bias) * torch.exp(log_scale)
        dlog_det = torch.sum(log_scale, dim=(1, 2, 3)) * dims
        return x, dlog_det

    def reverse(self, x, cond):
        cond_b, _, = cond.shape
        cond = self.cond_net(cond).reshape(cond_b, -1, 1, 1)
        log_scale, bias = cond.chunk(2, dim=1)
        dims = x.shape[2] * x.shape[3]
        x = x*torch.exp(-log_scale) - bias
        return x
```

### 2.2.2 (Conditional) Invertible 1x1 Convolution

在 Glow 中輸入與輸出的關係可以表示成：

$$u_{i,j} = Wv_{i,j}$$

其中 $u$ 與 $v$ 分別是輸入與輸出，$W$ 就是置換矩陣，在 Glow 中是直接透過神經網路學習，而 Conditional Glow 則是將其改寫成：

$$W = CN(x)$$
$$u_{i,j} = Wv_{i,j}$$

利用 condition $x$ 與 conditioning network $CN$ 來產生置換矩陣 $W$。

下面是 Invertible 1x1 Convolution 與 Conditional Invertible 1x1 Convolution 的程式碼。conditional 的部分比照 Conditional Actnorm，只保留 conditioning network 後半部的 fully-connected layer。我在一般的 Invertible 1x1 Convolution 有使用 LU decomposition 來計算逆矩陣，conditional 的版本因為 weight matrix 是透過網路學習的，所以只能直接用 `torch.inverse` 取逆矩陣。

```
# Glow
class Invertible1x1Conv(nn.Module):
    def __init__(self, in_chs):
        super(Invertible1x1Conv, self).__init__()
        weight = np.random.randn(in_chs, in_chs)
        q, _ = la.qr(weight)
        w_p, w_l, w_u = la.lu(q.astype(np.float32))
        w_s = np.diag(w_u)
        w_u = np.triu(w_u, 1)
        u_mask = np.triu(np.ones_like(w_u), 1)
        l_mask = u_mask.T
        w_p = torch.from_numpy(w_p.copy())
        w_l = torch.from_numpy(w_l.copy())
        w_s = torch.from_numpy(w_s.copy())
        w_u = torch.from_numpy(w_u.copy())
        self.register_buffer('w_p', w_p)
        self.register_buffer('u_mask', torch.from_numpy(u_mask.copy()))
        self.register_buffer('l_mask', torch.from_numpy(l_mask.copy()))
```

```
        self.register_buffer('s_sign', torch.sign(w_s))
        self.register_buffer('l_eye', torch.eye(l_mask.shape[0]))
        self.w_l = nn.Parameter(w_l)
        self.w_s = nn.Parameter(log_abs(w_s))
        self.w_u = nn.Parameter(w_u)

    def get_weight(self):
        weight = (
            self.w_p
            @ (self.w_l*self.l_mask+self.l_eye)
            @ ((self.w_u*self.u_mask)
                +torch.diag(self.s_sign*torch.exp(self.w_s)))
        )
        return weight.unsqueeze(2).unsqueeze(3)

    def forward(self, x):
        weight = self.get_weight()
        z = F.conv2d(x, weight)
        dlog_det = torch.sum(self.w_s) * x.shape[2] * x.shape[3]
        return z, dlog_det

    def reverse(self, x):
        weight = self.get_weight()
        weight = weight.squeeze().inverse().unsqueeze(2).unsqueeze(3)
        z = F.conv2d(x, weight)
        return z


# c-Glow
class CondInvertible1x1Conv(nn.Module):
    def __init__(self, in_chs, cond_sz, cond_fc_fts):
        super(CondInvertible1x1Conv, self).__init__()
        self.cond_net = nn.Sequential(
            nn.Linear(cond_sz, cond_fc_fts),
            nn.ReLU(inplace=True),
            nn.Linear(cond_fc_fts, cond_fc_fts),
            nn.ReLU(inplace=True),
            nn.Linear(cond_fc_fts, in_chs*in_chs),
            nn.Tanh())
        self.cond_net[0].weight.data.zero_()
        self.cond_net[0].bias.data.zero_()
        self.cond_net[2].weight.data.zero_()
        self.cond_net[2].bias.data.zero_()
        self.cond_net[4].weight.data.normal_(0, 0.05)
        self.cond_net[4].bias.data.normal_(0, 0.05)

    def get_weight(self, x, cond, inverse=False):
        x_c = x.shape[1]
        cond_b, _ = cond.shape
```

```
        cond = self.cond_net(cond)
        cond = torch.tanh(cond)
        weight = cond.reshape(cond_b, x_c, x_c)
        dims = x.shape[2] * x.shape[3]
        dlog_det = torch.slogdet(weight)[1] * dims
        if inverse:
            weight = torch.inverse(weight.cpu()).to(device)
        weight = weight.reshape(cond_b, x_c, x_c, 1, 1)
        return weight, dlog_det

    def forward(self, x, cond):
        weight, dlog_det = self.get_weight(x, cond)
        x_b, x_c, x_h, x_w = x.shape
        x = x.reshape(1, x_b*x_c, x_h, x_w)
        w_b, w_c, _, w_h, w_w = weight.shape
        assert x_b==w_b and x_c==w_c
        weight = weight.reshape(w_b*w_c, w_c, w_h, w_w)
        z = F.conv2d(x, weight, groups=x_b)
        z = z.reshape(x_b, x_c, x_h, x_w)
        return z, dlog_det

    def reverse(self, x, cond):
        weight, dlog_det = self.get_weight(x, cond, inverse=True)
        x_b, x_c, x_h, x_w = x.shape
        x = x.reshape(1, x_b*x_c, x_h, x_w)
        w_b, w_c, _, w_h, w_w = weight.shape
        assert x_b==w_b and x_c==w_c
        weight = weight.reshape(w_b*w_c, w_c, w_h, w_w)
        z = F.conv2d(x, weight, groups=x_b)
        z = z.reshape(x_b, x_c, x_h, x_w)
        return z
```

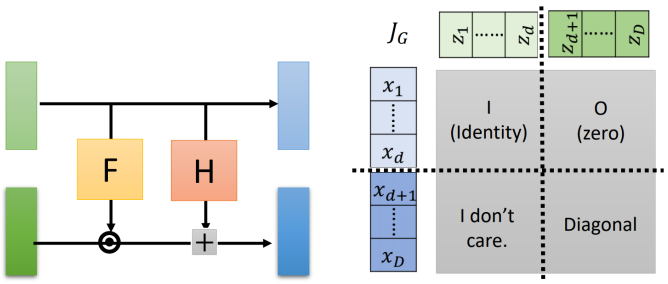
### 2.2.3 (Conditional) Affine

在 Glow 中輸入與輸出的關係可以表示成：

$$
\begin{aligned}
v_1, v_2 &= split(v) \\
s_2, b_2 &= NN(v_1) \\
u_2 &= s_2 \odot v_2 + b_2 \\
u &= concat(v_1, v_2)
\end{aligned}
$$

參考圖 3(a)，$v_1$ 與 $v_2$ 分別是左邊上下兩個綠色向量，圖中是分別將 $v_1$ 送進兩個網路得到 $s$ 與 $b$，而論文中其實是直接將 $v_1$ 作為 $NN$ 這個神經網路的輸入，再把輸出分成 $s$ 與 $b$，利用這組參數對 $v_2$ 做仿射轉換後得到 $u_2$，並將 $v_1$ 與 $u_2$ 接在一起作為 affine layer 的輸出。

而 Conditional Glow 則是將其改寫成：




(a) Affine coupling (b) Jacobian matrix

Figure 3: Affine coupling layer
（來源：李宏毅老師 Flow-based Generative Model 講義）

$$
\begin{aligned}
v_1, v_2 &= split(v) \\
x_r &= CN(x) \\
s_2, b_2 &= NN(v_1, x_r) \\
u_2 &= s_2 \odot v_2 + b_2 \\
u &= concat(v_1, u_2)
\end{aligned}
$$

將 condition $x$ 向量經過 conditioning network $CN$ 轉過後，接在 $v_1$ 上再利用 $NN$ 計算 $s$ 與 $b$。

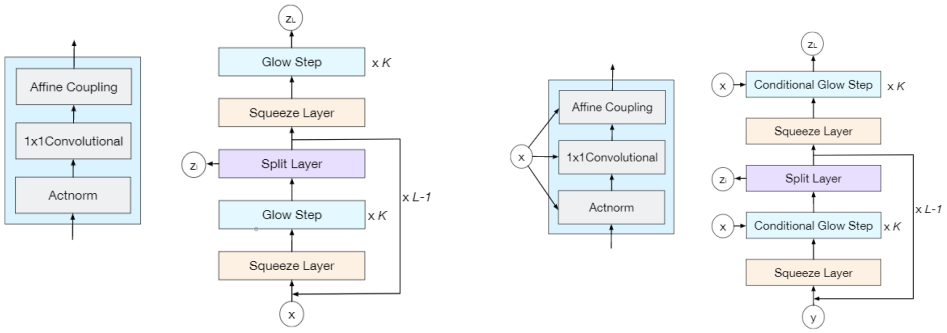
下面是 Affine Coupling 與 Conditional Affine Coupling 的程式碼。Conditional Glow 論文中這部分的 conditioning network 完全是透過 convolution 來達成，因為作業用的 condition 形式不一樣，這邊我是利用 fully-connected layer 產生與 convolution 結果相同數量的值，再去調整向量的形狀並接上 $v_1$。另外計算行列式的方法可以參考圖 3(b)，因為 $u_1$ 是直接複製 $v_1$，所以 Jacobian matrix 左上角部分就會是單位矩陣，由於 $v_2$ 跟 $u_1$ 之間完全沒有關係，Jacobian matrix 右上角就會都是 0，正因為右上角都是 0，在計算行列式的時候就不用去管左下角的數字，所以右下角區塊的行列式就會是整個 Jacobian matrix 的行列式。

```
# Glow
class AffineCoupling(nn.Module):
    def __init__(self, in_chs, affine_conv_chs=512):
        super(AffineCoupling, self).__init__()
        self.net = nn.Sequential(
            nn.Conv2d(in_chs//2, affine_conv_chs, 3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(affine_conv_chs, affine_conv_chs, 1, padding=0),
            nn.ReLU(inplace=True),
            ZeroConv2d(affine_conv_chs, in_chs))
        self.net[0].weight.data.normal_(0, 0.05)
        self.net[0].bias.data.zero_()
        self.net[2].weight.data.normal_(0, 0.05)
        self.net[2].bias.data.zero_()

    def forward(self, x):
        z1, z2 = x.chunk(2, dim=1)
        log_scale, shift = self.net(z1).chunk(2, dim=1)
        scale = torch.sigmoid(log_scale+2)
        z2 = (z2+shift) * scale
        dlog_det = torch.sum(torch.log(scale), dim=(1, 2, 3))
```

```
        z = torch.cat((z1, z2), dim=1)
        return z, dlog_det

    def reverse(self, x):
        z1, z2 = x.chunk(2, dim=1)
        log_scale, shift = self.net(z1).chunk(2, dim=1)
        scale = torch.sigmoid(log_scale+2)
        z2 = z2/scale - shift
        z = torch.cat((z1, z2), dim=1)
        return z


# c-GLow
class CondAffineCoupling(nn.Module):
    def __init__(self, in_sz, cond_sz, cond_fc_fts, affine_conv_chs):
        super(CondAffineCoupling, self).__init__()
        self.cond_net = nn.Sequential(
            nn.Linear(cond_sz, cond_fc_fts),
            nn.ReLU(inplace=True),
            nn.Linear(cond_fc_fts, cond_fc_fts),
            nn.ReLU(inplace=True),
            nn.Linear(cond_fc_fts, (in_sz[0]//2)*in_sz[1]*in_sz[2]),
            nn.ReLU(inplace=True))
        self.cond_net[0].weight.data.zero_()
        self.cond_net[0].bias.data.zero_()
        self.cond_net[2].weight.data.zero_()
        self.cond_net[2].bias.data.zero_()
        self.cond_net[4].weight.data.zero_()
        self.cond_net[4].bias.data.zero_()
        self.net = nn.Sequential(
            nn.Conv2d(in_sz[0], affine_conv_chs, 3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(affine_conv_chs, affine_conv_chs, 1, padding=0),
            nn.ReLU(inplace=True),
            ZeroConv2d(affine_conv_chs, in_sz[0]))
        self.net[0].weight.data.zero_()
        self.net[0].bias.data.zero_()
        self.net[2].weight.data.zero_()
        self.net[2].bias.data.zero_()

    def forward(self, x, cond):
        z1, z2 = x.chunk(2, dim=1)
        cond = self.cond_net(cond)
        cond = cond.reshape(cond.shape[0], x.shape[1]//2, *x.shape[2:])
        tmp = torch.cat((z1, cond), dim=1)
        tmp = self.net(tmp)
        log_scale, shift = tmp.chunk(2, dim=1)
        scale = torch.sigmoid(log_scale+2)
        z2 = (z2+shift) * scale
```

```
        dlog_det = torch.sum(torch.log(scale), dim=(1, 2, 3))
        z = torch.cat((z1, z2), dim=1)
        return z, dlog_det

    def reverse(self, x, cond):
        z1, z2 = x.chunk(2, dim=1)
        cond = self.cond_net(cond)
        cond = cond.reshape(cond.shape[0], x.shape[1]//2, *x.shape[2:])
        tmp = torch.cat((z1, cond), dim=1)
        tmp = self.net(tmp)
        log_scale, shift = tmp.chunk(2, dim=1)
        scale = torch.sigmoid(log_scale+2)
        z2 = z2/scale - shift
        z = torch.cat((z1, z2), dim=1)
        return z
```

### 2.2.4 (Conditional) Glow architecture




(a) Glow (b) Conditional Glow

Figure 4: NF architecture

圖 4 為 Glow 與 Conditional Glow 的整體架構，會有 $L-1$ 個 Squeeze-Glow-Split 這樣的組合，而每個 Glow Step 則是由 Actnorm、1x1 Convolution 與 Affine Coupling 組成，建構 Glow Step 的程式碼如下：

```
# Glow
class GlowStep(nn.Module):
    def __init__(self, in_chs, affine_conv_chs, actnorm_inited=False):
        super(GlowStep, self).__init__()
        self.actnorm = Actnorm(in_chs, actnorm_inited)
        self.conv1x1 = Invertible1x1Conv(in_chs)
        self.affine_coupling = AffineCoupling(in_chs, affine_conv_chs)

    def forward(self, x):
        out, dlog_det1 = self.actnorm.forward(x)
        out, dlog_det2 = self.conv1x1.forward(out)
        out, dlog_det3 = self.affine_coupling.forward(out)
        log_det = dlog_det1 + dlog_det2 + dlog_det3
```

```
        return out, log_det

    def reverse(self, x):
        out = self.affine_coupling.reverse(x)
        out = self.conv1x1.reverse(out)
        out = self.actnorm.reverse(out)
        return out


# c-Glow
class CondGlowStep(nn.Module):
    def __init__(self, in_sz, cond_sz, cond_fc_fts, affine_conv_chs):
        super(CondGlowStep, self).__init__()
        self.actnorm = CondActnorm(in_sz[0], cond_sz, cond_fc_fts)
        self.conv1x1 = CondInvertible1x1Conv(
                                    in_sz[0], cond_sz, cond_fc_fts)
        self.affine_coupling = CondAffineCoupling(
                                    in_sz, cond_sz,
                                    cond_fc_fts, affine_conv_chs)

    def forward(self, x, cond):
        out, dlog_det1 = self.actnorm.forward(x, cond)
        out, dlog_det2 = self.conv1x1.forward(out, cond)
        out, dlog_det3 = self.affine_coupling.forward(out, cond)
        log_det = dlog_det1 + dlog_det2 + dlog_det3
        return out, log_det

    def reverse(self, x, cond):
        out = self.affine_coupling.reverse(x, cond)
        out = self.conv1x1.reverse(out, cond)
        out = self.actnorm.reverse(out, cond)
        return out
```

### 2.3 Hyperparameters

#### 2.3.1 Conditional DCGAN

- Image size: [64, 64, 3]
- 300 epochs
- Batch size: 128
- Learning rate: 2e-4
- Adam optimizer
- Channels of feature maps of generator: [2400, 1200, 600, 300, 3]
- Channels of feature maps of discriminator: [4, 100, 200, 400, 800]

#### 2.3.2 Glow

- Image size: [64, 64, 3]
- 100000 iterations
- Batch size: 16

- Learning rate: 2e-4
- Adam optimizer
- Flow depth (K): 32
- Number of level (L): 4

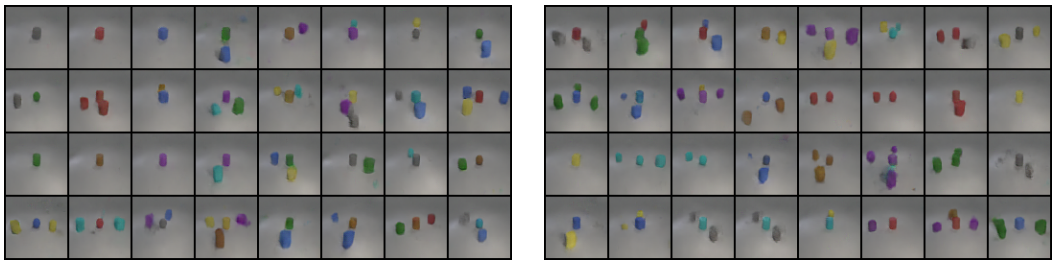
#### 2.3.3 Conditional Glow

- Image size: [64, 64, 3]
- 100000 iterations
- Batch size: 16
- Learning rate: 2e-4
- Adam optimizer
- Flow depth (K): 32
- Number of level (L): 4

## 3 Results

### 3.1 Task 1

#### 3.1.1 GAN



(a) test.json (b) new_test.json

Figure 5: Conditional DCGAN

圖 5 為 Conditional DCGAN 在 `test.json` 與 `new_test.json` 的測試結果，各執行十次的平均準確度如下：

- `test.json`：74.72%
- `new_test.json`：63.30%

#### 3.1.2 NF

Not yet finished.

#### 3.1.3 Discussion

在訓練 Conditional DCGAN 的過程中，我發現取得 generator 跟discriminator 的平衡很重要，一開始我讓兩個網路一樣大，結果在訓練階段前期 generator 的 loss 很快地就下降到 0，使往後的訓練失敗。我想應該是因為網路太大導致 discriminator 根本還來不及學好，generator 很輕易地就能夠騙過 discriminator。後來我把 discriminator 的網路寬度縮小，我認為參數量變少應該會比較快收斂，而結果也如我預期一樣，discriminator 在訓練前期很快就收斂到一定的幅度，後面的部分就是與 generator 互相去對抗，這樣做出來的結果也好很多。

在訓練 Conditional NL 的過程中，我在一直遇到 weight matrix 為 singular 導致程式執行終止的情況，雖然 loss function 的設計理論上要去避免這種情況發生，我想是因為 weight matrix 是用神經網路計算的，很難保證 back-propagation 之後不會剛好出現 singular 的情況，

一般的 Glow 在訓練時就完全不會遇到這種狀況，但我也不確定是不是 Condition Glow 有寫錯。

比較兩個網路架構，以訓練的難易度來說，我認為 GAN 是比較容易訓練的，因為只要有一個好的 discriminator 去與 generator 抗衡，就比較容易得到好的訓練結果；反之 NL 需要考慮的東西很多，除了先前提到可能會有 singular matrix 的問題外，要有幾個 level 以及每個 level 裡面 Glow step 的數量也都會影響到模型的表現；但是如果以模型的强大與否，我認為 NF 優於 GAN，NF 中的每個 block 其實都很簡單，透過不斷堆疊增加模型的複雜度，直接找出與常態分佈機率模型之間的轉換關係，而且不需要 discriminator 來幫助訓練，NF 本身就可以透過 negative log-likelihood 去學習，另外因為整個模型是可逆的，訓練完 encoder 的同時 decoder 也訓練好了，使得我們能夠在 latent space 對數據進行修改，再轉換回圖片，這類的應用是 GAN 所無法辦到的。

## 3.2 Task 2

### 3.2.1 Conditional face generation

Not yet finished.

### 3.2.2 Linear interpolation

這部分的實作方法為：每次從所有相片中隨機挑選兩張，並用 NF 計算兩張照片各自的 $z$ 向量，並利用這兩個向量內插出之間的向量，再將得到的向量用 NF 轉換回去。內差的實際運算方法如下，其中 $\alpha$ 為右圖 $z$ 向量的占比：

$$z_{new} = (1-\alpha) * z_{left} + \alpha * z_{right}$$

這部分的程式碼如下：

```
n = args.interpolate_step
# sample 6 photos (3 pairs)
target_idx = random.choices(indices, k=6)
target_zs, zs = [], []
for idx in target_idx:
    img = read_image(
            os.path.join(root_dir, 'CelebA-HQ-img', img_list[idx]), trans)
    _, _, z = model.forward(img)
    target_zs.append(z)
zs_l, zs_r = target_zs[:3], target_zs[3:]
for z_idx in range(len(zs_l[0])):
    z = []
    for z_pair in zip(zs_l, zs_r):
        for i in range(n):
            new_z = (n-i)*z_pair[0][z_idx]/n + i*z_pair[1][z_idx]/n
            z.append(new_z)
    zs.append(torch.cat(z))
gen_images = model.reverse(zs, reconstruct=True)
# generated images are in [-0.5, 0.5], shift them to [0, 1] by adding 0.5
save_image(
    gen_images+0.5,
    os.path.join(args.outout_dir, 'task2_interpolate.png'),
    nrow=n)
```

圖 6 為 linear interpolation 的結果。

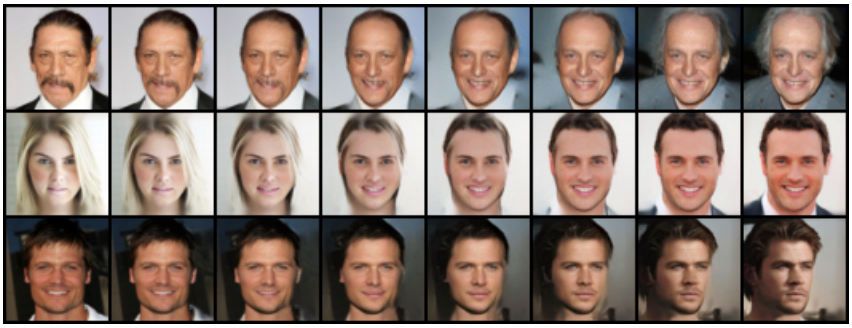


Figure 6: Linear interpolation

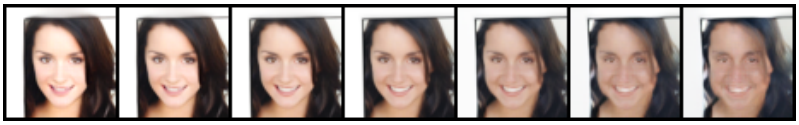
### 3.2.3 Attribute manipulation

這部分我有修改助教提供的 `CelebADataset`，可以選取針對特定的特徵，這樣就可以套用 `DataLoader` ，計算 $z_{pos}$ 跟 $z_{neg}$ 的時候會比較有效率。這部分的實作方法為：挑選一個 attribute 並抓出 $image_{pos}$ 跟 $image_{neg}$，分別計算這兩組圖片的 $z$ 的平均值，算出 $z_{pos}$ 與 $z_{neg}$ 之後將其相減得到所謂的 attribute vector，再隨機選取一張相片並用 NF 轉成 $z$ 向量，做 manipulation 的時候我會將 $z$ 減去或加上 0.5、1 與 1.5 倍的 attribute vector，再用 NF 轉換回去得到圖片。

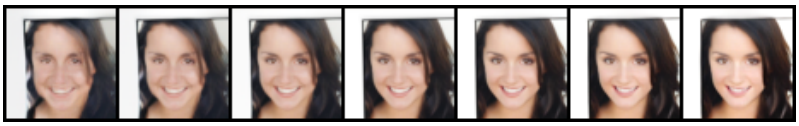
這部分的程式碼如下：

```
n = args.manipulation_step
idx = random.choice(indices)
img = read_image(
        os.path.join(root_dir, 'CelebA-HQ-img', img_list[idx]), trans)
_, _, z_img = model.forward(img)
for target_attr in ['chubby', 'young', 'blond_hair']:
    z_pos = compute_average_z(target_attr, True, root_dir, trans, model)
    z_neg = compute_average_z(target_attr, False, root_dir, trans, model)
    zs = []
    for z_idx in range(len(z_pos)):
        attr_vec = z_pos[z_idx] - z_neg[z_idx]
        z = []
        for i in range(-n, 0):
            new_z = z_img[z_idx] + attr_vec*i/2
            z.append(new_z)
        z.append(z_img[z_idx])
        for i in range(1, n+1):
            new_z = z_img[z_idx] + attr_vec*i/2
            z.append(new_z)
        zs.append(torch.cat(z))
    gen_images = model.reverse(zs, reconstruct=True)
    save_image(
        gen_images+0.5,
        os.path.join(args.output_dir, f'task3_{target_attr}.png'),
        nrow=2*n+1)
```

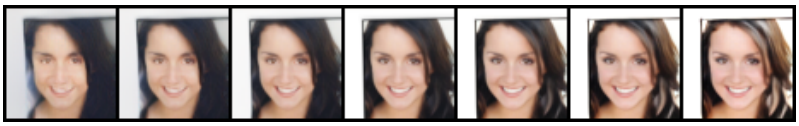
圖 7 為 attribute manipulation 的結果，可以發現圖 7(a) 與圖 7(b) 很像是左右顛倒，可以猜測 Chubby 與 Young 這兩個特徵某種程度上是方向相反的。



(a) Chubby



(b) Young



(c) Blond Hair

(d) Chubby

(e) Blond Hair

Figure 7: Attribute manipulation