NCTU DLP Lab6-Report
Let’s Play GAN

廖家鴻 0786009

2020/5/26

Outline

- **Introduction**
- **Implementation Details**
  - A. Dataloader
  - B. cGAN Selection
  - C. Discriminator
  - D. Generator
  - E. Loss
  - F. Hyper Parameters
- **Results and Discussion**
  - A. Training Experience
  - B. Best Training Results
  - C. Best Training Setting
  - D. Best Model Setting

Introduction

- **Implementation Details**
  A. Dataloader
  B. cGAN Selection
  C. Discriminator
  D. Generator
  E. Loss
  F. Hyper Parameters

首先講解幾個重要的component的implementation細節，及重要的超參數。

- **Results and Discussion**
  A. Training Experience
  B. Best Training Results
  C. Best Training Setting
  D. Best Model Setting

在Result and Discussion中，會呈現：

1. 一開始用cDCGAN練手的經驗，及加LSGAN、WGAN-GP手法的心得。
2. 結合前面的training經驗，改用projection discriminator paper方法的最佳結果。
3. 最佳的training設置與重要model調整手法。

Implementation Details

Implementation Details-Detail of Dataloader-1

下圖即為將三個json檔的讀取處理

```python
def getData(mode):
    with open(info_path+'objects.json', 'r', encoding='utf-8') as f:
            objects_info = json.load(f)
    if mode == 'train':
        with open(info_path+'train.json', 'r', encoding='utf-8') as f:
            train_info = json.load(f)
        inputs = []
        labels = []
        for fn, obj_ls in train_info.items():
            inputs.append(fn)
            objects = np.zeros(24)
            for obj in obj_ls:
                objects[objects_info[obj]]=1
            labels.append(objects)
        return inputs, labels
    else:
        with open(info_path+'test.json', 'r', encoding='utf-8') as f:
            test_info = json.load(f)
        inputs = []
        labels = []
        for obj_ls in test_info:
            objects = []
            for obj in obj_ls:
                objects.append(objects_info[obj])
            labels.append(objects)
        return inputs, labels
```

下圖即為data_loader for training pair的codes

```python
class iclevrLoader(data.Dataset):
    def __init__(self, root, mode, transform=None):
        self.root = root
        self.img_name, self.label = getData(mode)
        self.mode = mode
        print("> Found %d images..." % (len(self.img_name)))
        self.transform = transform

    def __len__(self):
        return len(self.img_name)

    def __getitem__(self, idx):
        if torch.is_tensor(idx):
            idx = idx.tolist()
        #step1.Get the image path from 'self.img_name' and load it.
        img_fn = self.img_name[idx]
        img_fn = join(self.root,img_fn)
        image_fn = Image.open(img_fn).convert('RGB')

        #step2. Get the ground truth label from self.label
        label_fn = torch.Tensor([self.label[idx]]).long().squeeze()

        #step3. Transform the images during the training phase
        if self.transform:
            image_fn = self.transform(image_fn)
        else:
            image_fn = io.imread(img_fn)
            image_fn = np.array(image_fn)
            image_fn = torch.FloatTensor(image_fn)

        #step4. Return processed image and label
        sample = {'image': image_fn, 'label': label_fn}
        return sample
```

Implementation Details-Detail of Dataloader-2

```
##### Training Dataloader
normalize = transforms.Normalize(mean=[0.5, 0.5, 0.5],
                                 std=[0.5, 0.5, 0.5])
data_transform = transforms.Compose([
        transforms.Resize((64,64)),
        # transforms.RandomAffine(0, shear=10, scale=(0.8,1.2)),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        normalize])

data_path = 'iclevr/'
train_datasets = iclevrLoader(root=data_path, mode='train', transform=data_transform)
train_loader = DataLoader(train_datasets, batch_size=32, shuffle=True) #, num_workers=4)
```

左圖為Training DataLoader與transform的相關設置。

```
_, test_label = getData('test')
eval_y_test_lb = []
for i in range(len(test_label)):
    lb = torch.zeros(num_label, 1, 1)
    # lb = torch.zeros(num_label)
    for n in test_label[i]:
        lb += onehot[n]
    eval_y_test_lb.append(lb.unsqueeze(0))
eval_y_test_lb = torch.cat(eval_y_test_lb, axis=0).squeeze().squeeze()
```

上圖是為了算evaluator的acc值而做的label的onehot vector。而此vector長度為24，並且等於1的index對應object label index。所以每個vector有1~3個1出現，輸出的樣子如下圖：

```
tensor([[1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0.],
        [0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0.],
        [0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0.],
        [0., 0., 1., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0.],
```

```
##### 注意沒有object的index為0
y_label = []
for b in range(len(y_)):
    y_lb = np.nonzero(y_[b]) +1
    # print(y_lb)
    cy = torch.zeros([1, 3]).cuda() #[batch, channel, img_size, img_size]
    i = 0
    for n in y_lb:
        cy[0,i] += n[0]
        i+=1
    y_label.append(cy)
y_label_c = torch.cat(y_label, axis=0).type(torch.LongTensor)
```

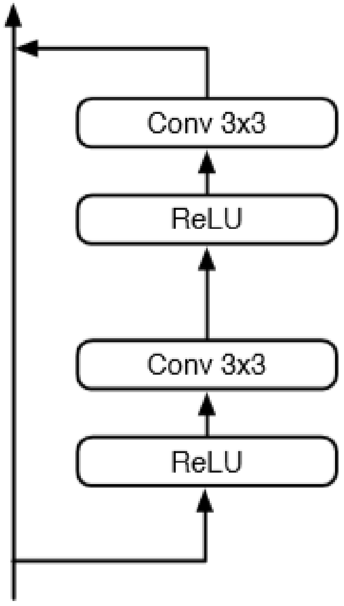
上圖為train時，為了embedding而做的label encoding。值得注意的是，我將所有label index+1，然後將”無object”的label index設為0。並且右圖即為送進embedding前的label tensor的樣子。

```
tensor([[22,  0,  0],
        [22, 23,  0],
        [ 5, 13,  0],
        [ 4,  0,  0],
        [12, 21, 23],
        [16,  0,  0],
        [23,  0,  0],
        [13, 19,  0],
        [ 5, 13, 20],
        [10,  0,  0],
        [ 2,  0,  0],
        [ 6, 11,  0],
```

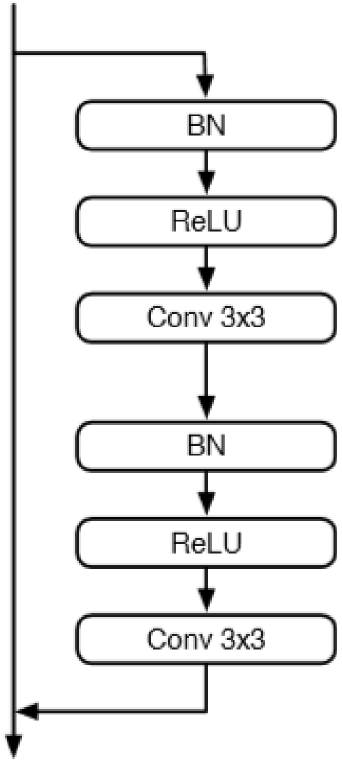
Implementation Details-cGAN selection

CGANS WITH PROJECTION DISCRIMINATOR

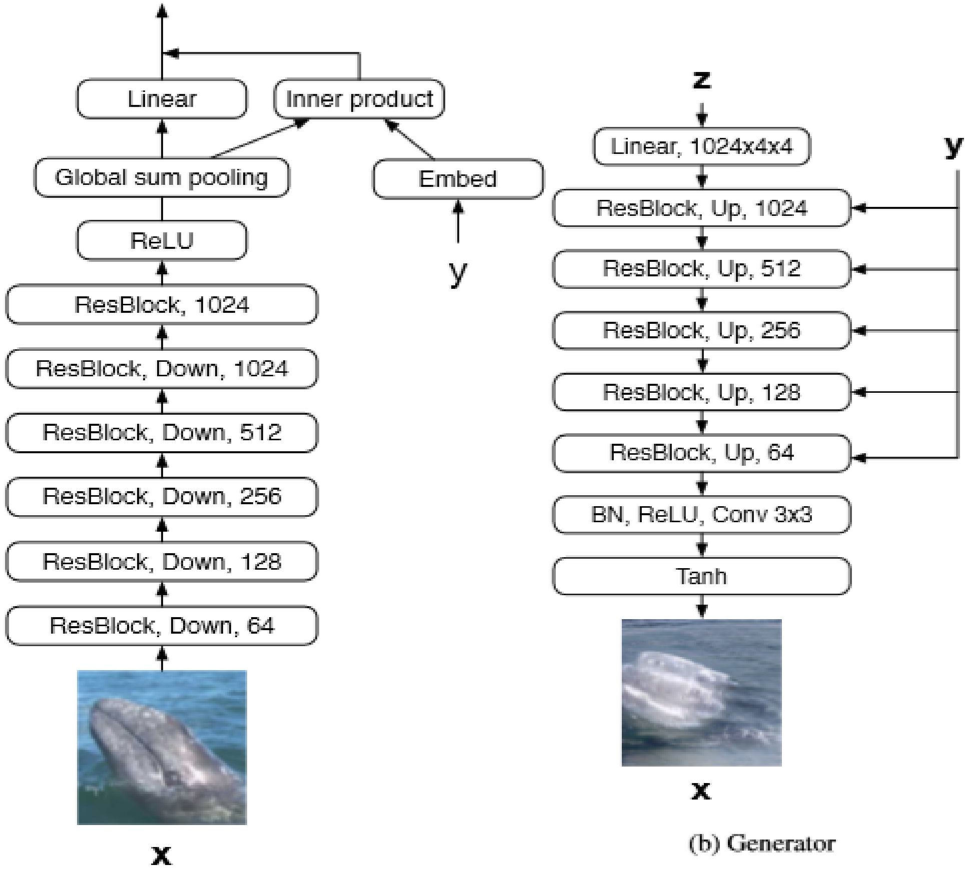
Takeru Miyato[1], Masanori Koyama[2]
miyato@preferred.jp
koyama.masanori@gmail.com
[1]Preferred Networks, Inc. [2]Ritsumeikan University



(a) ResBlock architecture for the discriminator. Spectral normalization (Miyato et al., 2018) was applied to each *conv* layer.



(b) ResBlock for the generator.



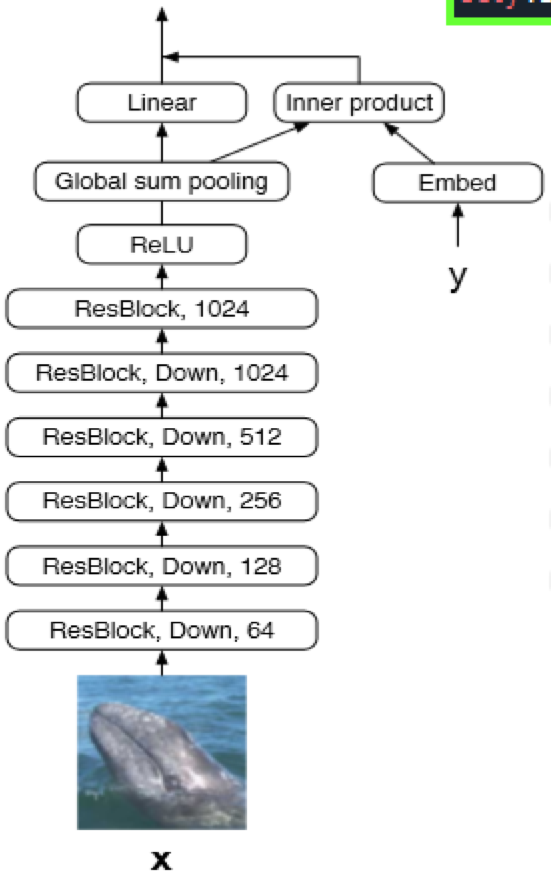
(b) Generator

(a) Discriminator

Figure 14: The models we used for the conditional image generation task.

其實我一開始從最基本的cDCGAN加上WGAN-GP的方法開始試，但效果不佳，且mode collapse問題難解，然後一直都只有一個明顯的object，該兩個or三個object的都生成得不太明顯，所以後來決定嘗試projection的方法，並且主要參考這個source code： https://github.com/crcrpar/pytorch.sngan_projection

Implementation Details-Detail of Discriminator1



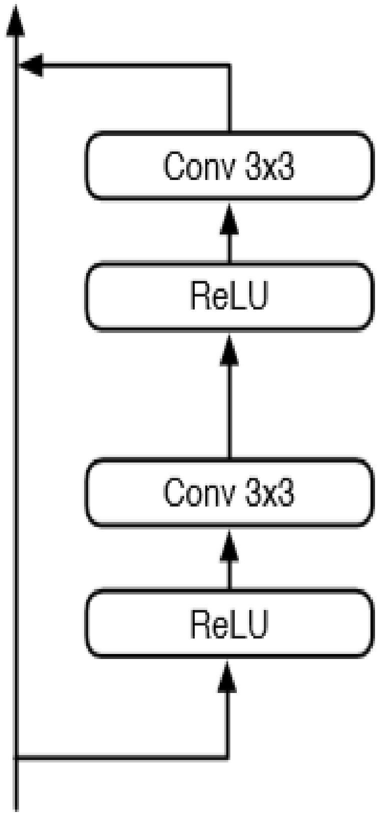

```
self.l_y_embed = utils.spectral_norm(nn.Embedding(num_classes, num_features * 16))
```

```
def forward(self, x, y_emb=None, y_lin=None):
    y_cond = self.cond_emb_in(y_emb).view(y_emb.size(0), -1,
                                          self.num_features, self.num_features)
    h = x
    h = torch.cat([x, y_cond],1)
    h = self.block1(h)
    h = self.block2(h)
    # h = torch.cat([h, y_cond_conv],1)
    h = torch.cat([h, y_cond_2],1)
    h = self.block3(h)
    h = self.block4(h)
    h = self.block5(h)
    h = self.activation(h)
    ##### Global pooling #####
    h = torch.sum(h, dim=(2, 3))
    output = self.l6(h)
    if y_emb is not None:
        l_y_sum = self.l_y_embed(y_emb)

    ##### Inner products for Embedding projections of 1 to 3 objects #####
    output1 = torch.sum(torch.mul(l_y_sum[:,0,:] ,h), dim=1, keepdim=True)
    output2 = torch.sum(torch.mul(l_y_sum[:,1,:] ,h), dim=1, keepdim=True)
    output3 = torch.sum(torch.mul(l_y_sum[:,2,:] ,h), dim=1, keepdim=True)
    output = output + (output1 + output2 +output3)/3

    return output
```

左二圖分別是Discriminator的架構主體與其code，較需留意的地方是，本次lab預計生成1至3個objects的圖，而我的處理方式是：

1. 增加一個”無object”的label，加上原本24種object，所以總共有25個label index。
2. 每次都有三個label index要embedding。(若只要生成一個object，那就會有兩個”無object”的label index)。
3. Labex index 被embedding後，分別與global pooling後的vector做內積，分別產生output1、2、3。
4. 以上三個projections最後再跟linear層的output相加，得到最後輸出值。

Implementation Details-Detail of Discriminator2



(a) ResBlock architecture for the discriminator. Spectral normalization (Miyato et al., 2018) was applied to each *conv* layer.

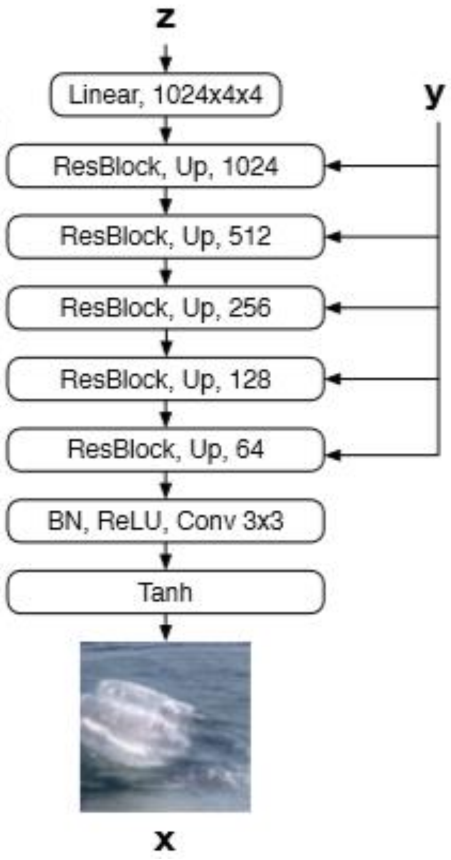
```python
class Block(nn.Module):
    def __init__(self, in_ch, out_ch, h_ch=None, ksize=3, pad=1,
                 activation=F.relu, downsample=False):
        super(Block, self).__init__()
        self.activation = activation
        self.downsample = downsample
        self.learnable_sc = (in_ch != out_ch) or downsample
        if h_ch is None:
            h_ch = in_ch
        else:
            h_ch = out_ch
        self.c1 = utils.spectral_norm(nn.Conv2d(in_ch, h_ch, ksize, 1, pad))
        self.c2 = utils.spectral_norm(nn.Conv2d(h_ch, out_ch, ksize, 1, pad))
        if self.learnable_sc:
            self.c_sc = utils.spectral_norm(nn.Conv2d(in_ch, out_ch, 1, 1, 0))
        self._initialize()

    def forward(self, x):
        return self.shortcut(x) + self.residual(x)

    def shortcut(self, x):
        if self.learnable_sc:
            x = self.c_sc(x)
        if self.downsample:
            return F.avg_pool2d(x, 2)
        return x

    def residual(self, x):
        h = self.c1(self.activation(x))
        h = self.c2(self.activation(h))
        if self.downsample:
            h = F.avg_pool2d(h, 2)
        return h
```

左二圖分別是Discriminator的架構中，residual block的結構與其code，較需留意的地方是，每次convolution與shortcut都有做spectral norm，而這是這篇paper作者的前一篇paper的方法。

而左圖的Block class建好後，則由下圖的code來將各block stack起來形成discriminator的主體。

```python
class SNResNetProjectionDiscriminator(nn.Module):
    def __init__(self, num_features=64, num_classes=24, activation=F.relu):
        super(SNResNetProjectionDiscriminator, self).__init__()
        self.num_features = num_features
        self.num_classes = num_classes
        self.activation = activation

        # self.block1 = OptimizedBlock(3, num_features)
        self.block1 = OptimizedBlock(3+3, num_features)
        self.block2 = Block(num_features, num_features * 2,
                            activation=activation, downsample=True)
        self.block3 = Block(num_features * 2, num_features * 4,
                            activation=activation, downsample=True)
        self.block4 = Block(num_features * 4, num_features * 8,
                            activation=activation, downsample=True)
        self.block5 = Block(num_features * 8, num_features * 16,
                            activation=activation, downsample=True)
        self.l6 = utils.spectral_norm(nn.Linear(num_features * 16, 1))
        if num_classes > 0:
            self.cond_emb_in = utils.spectral_norm(
                nn.Embedding(num_classes, num_features ** 2))
            self.l_y_embed = utils.spectral_norm(
                nn.Embedding(num_classes, num_features * 16))
```

Implementation Details-Detail of Generator1



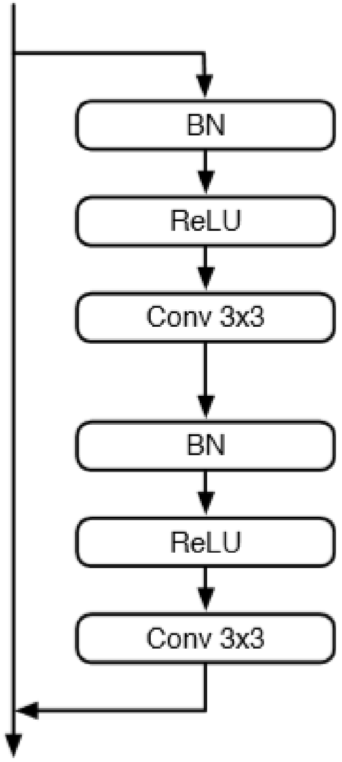

```python
class ResNetGenerator(nn.Module):
    """Generator generates 64x64."""
    def __init__(self, num_features=64, dim_z=100, bottom_width=4,
                 activation=F.relu, num_classes=0, distribution='normal'):
        super(ResNetGenerator, self).__init__()
        self.num_features = num_features
        self.dim_z = dim_z
        self.bottom_width = bottom_width
        self.activation = activation
        self.num_classes = num_classes
        self.distribution = distribution
        self.cond_emb = nn.Embedding(num_classes, num_features ** 2)

        self.l1 = nn.Linear(dim_z, (16-4) * num_features * bottom_width ** 2)
        self.block2 = Block(num_features * (16-4) * 2, num_features * 8,
                            activation=activation, upsample=True, num_classes=num_classes)
        self.block3 = Block(num_features * 8, num_features * 4,
                            activation=activation, upsample=True, num_classes=num_classes)
        self.block4 = Block(num_features * 4, num_features * 2,
                             activation=activation, upsample=True, num_classes=num_classes)
        self.block5 = Block(num_features * 2, num_features,
                             activation=activation, upsample=True, num_classes=num_classes)
        self.b6 = nn.BatchNorm2d(num_features)
        self.conv6 = nn.Conv2d(num_features, 3, 1, 1)

    def forward(self, z, y=None, **kwargs):
        y_cond = self.cond_emb(y).view(y.size(0), -1, self.bottom_width, self.bottom_width)
        h = self.l1(z).view(z.size(0), -1, self.bottom_width, self.bottom_width)
        h = torch.cat([h, y_cond], 1)
        for i in range(2, 6):
            h = getattr(self, 'block{}'.format(i))(h, y, **kwargs)
        h = self.activation(self.b6(h))
        return torch.tanh(self.conv6(h))
```

左二圖分別是Generator的架構主體與其code。

至於原paper作法的conditional y是如何融進各layer後面說明。

黃色框的部分，是我在paper的作法之外，另用concat的方式將conditional_y跟經一次linear後的noise_z合併後再餵進generator主體裡。這樣做發現雖然並沒有提高eval()的accuracy，但有加速收斂的效果。

Implementation Details-Detail of Generator2



(b) ResBlock for the generator.

```python
def _upsample(x):
    h, w = x.size()[2:]
    return F.interpolate(x, size=(h*2, w*2), mode='bilinear', align_corners=True)

class Block(nn.Module):
    def __init__(self, in_ch, out_ch, h_ch=None, ksize=3, pad=1,
                 activation=F.relu, upsample=False, num_classes=0):
        super(Block, self).__init__()
        self.activation = activation
        self.upsample = upsample
        self.learnable_sc = in_ch != out_ch or upsample
        if h_ch is None:
            h_ch = out_ch
        self.num_classes = num_classes

        # Register layers
        self.c1 = nn.Conv2d(in_ch, h_ch, ksize, 1, pad)
        self.c2 = nn.Conv2d(h_ch, out_ch, ksize, 1, pad)
        if self.num_classes > 0:
            self.b1 = CategoricalConditionalBatchNorm2d(
                num_classes, in_ch)
            self.b2 = CategoricalConditionalBatchNorm2d(
                num_classes, h_ch)
        else:
            self.b1 = nn.BatchNorm2d(in_ch)
            self.b2 = nn.BatchNorm2d(h_ch)
        if self.learnable_sc:
            self.c_sc = nn.Conv2d(in_ch, out_ch, 1)

    def forward(self, x, y=None, z=None, **kwargs):
        return self.shortcut(x) + self.residual(x, y, z)
```

左二圖分別是Generator的架構主體中，各residual block的coding細節。黃框裡有調用Categorical Conditional BatchNorm2d的部分下張slide說明。

下圖則是同一個Block class裡，shortcut及residual的處理：

```python
def shortcut(self, x, **kwargs):
    if self.learnable_sc:
        if self.upsample:
            h = _upsample(x)
        h = self.c_sc(h)
        return h
    else:
        return x

def residual(self, x, y=None, z=None, **kwargs):
    if y is not None:
        h = self.b1(x, y, **kwargs)
    else:
        h = self.b1(x)
    h = self.activation(h)
    if self.upsample:
        h = _upsample(h)
    h = self.c1(h)
    if y is not None:
        h = self.b2(h, y, **kwargs)
    else:
        h = self.b2(h)
    return self.c2(self.activation(h))
```

Implementation Details-Detail of Generator3

```python
class ConditionalBatchNorm2d(nn.BatchNorm2d):
    """Conditional Batch Normalization"""
    def __init__(self, num_features, eps=1e-05, momentum=0.1,
                 affine=False, track_running_stats=True):
        super(ConditionalBatchNorm2d, self).__init__(
            num_features, eps, momentum, affine, track_running_stats)

    def forward(self, input, weight, bias, **kwargs):
        self._check_input_dim(input)

        exponential_average_factor = 0.0
        if self.training and self.track_running_stats:
            self.num_batches_tracked += 1
            if self.momentum is None:  # use cumulative moving average
                exponential_average_factor = 1.0 / self.num_batches_tracked.item()
            else:  # use exponential moving average
                exponential_average_factor = self.momentum
        output = F.batch_norm(input, self.running_mean, self.running_var,
                              self.weight, self.bias,
                              self.training or not self.track_running_stats,
                              exponential_average_factor, self.eps)

        weight1, weight2, weight3 = weight[:,0,:], weight[:,1,:], weight[:,2,:]
        bias1, bias2, bias3 = bias[:,0,:], bias[:,1,:], bias[:,2,:]
        size = output.size()
        weight1 = weight1.unsqueeze(-1).unsqueeze(-1).expand(size)
        bias1 = bias1.unsqueeze(-1).unsqueeze(-1).expand(size)
        weight2 = weight2.unsqueeze(-1).unsqueeze(-1).expand(size)
        bias2 = bias2.unsqueeze(-1).unsqueeze(-1).expand(size)
        weight3 = weight3.unsqueeze(-1).unsqueeze(-1).expand(size)
        bias3 = bias3.unsqueeze(-1).unsqueeze(-1).expand(size)

        out1 = weight1 * output + bias1
        out2 = weight2 * output + bias2
        out3 = weight1 * output + bias3
        output = (out1 + out2 + out3)/3
        return output
```

左圖和下圖是projection discriminator的paper中，如何在Generator的架構裡融進conditional_y的方法。weight和bias是下圖forward裡面傳來的向量，是根據三個label_condition的embedding而定的，而向量長度和特徵圖channel數一樣。因為向量和label condition有一一對應的關係，所以channel將output生成weight，可融合condition資訊。weight和bias是經過一系列reshape操作，才能和output相乘加。另外，只在生成器中使用Categorical Conditional BatchNorm

```python
class CategoricalConditionalBatchNorm2d(ConditionalBatchNorm2d):
    def __init__(self, num_classes, num_features, eps=1e-5, momentum=0.1,
                 affine=False, track_running_stats=True):
        super(CategoricalConditionalBatchNorm2d, self).__init__(
            num_features, eps, momentum, affine, track_running_stats)

        self.weights_emb = nn.Embedding(num_classes, num_features)
        self.biases_emb = nn.Embedding(num_classes, num_features)

        self._initialize()

    def _initialize(self):
        init.ones_(self.weights_emb.weight.data)
        init.zeros_(self.biases_emb.weight.data)

    def forward(self, input, c_emb, **kwargs):
        weight_emb = self.weights_emb(c_emb)
        bias_emb = self.biases_emb(c_emb)

        return super(CategoricalConditionalBatchNorm2d, self).forward(
                                            input, weight_emb, bias_emb)
```

Implementation Details-Detail of Training Loss

```python
def dis_hinge(dis_fake, dis_real):
    loss = torch.mean(torch.relu(1. - dis_real)) + \
           torch.mean(torch.relu(1. + dis_fake))
    return loss

def gen_hinge(dis_fake, dis_real=None):
    return -torch.mean(dis_fake)
```

左上圖為cGAN with projection discriminator採用的Hinge loss。

```python
###### gradient penalty ######
# Loss weight for gradient penalty
lambda_gp = 10
def compute_gradient_penalty(D, real_samples, condition_y, fake_samples, mini_batch
    """Calculates the gradient penalty loss for WGAN GP"""
    # Random weight term for interpolation between real and fake samples
    alpha = torch.from_numpy(np.random.random((mini_batch, 1, 1, 1))).cuda()

    # Get random interpolation between real and fake samples
    intplo_real = torch.mul(alpha, real_samples)
    intplo_fake = torch.mul((1 - alpha), fake_samples)
    interpolates = (intplo_real + intplo_fake).requires_grad_(True)
    d_interpolates = D(interpolates.type(torch.FloatTensor).cuda(), condition_y)

    fake = Variable(torch.ones(mini_batch, 1), requires_grad=False)

    # Get gradient w.r.t. interpolates
    gradients = autograd.grad(outputs=d_interpolates.cuda(),
                              inputs=interpolates.cuda(),
                              grad_outputs=fake.cuda(), create_graph=True,
                              retain_graph=True, only_inputs=True)[0]
    gradients = gradients.view(gradients.size(0), -1)
    gradient_penalty = ((gradients.norm(2, dim=1) - 1) ** 2).mean()
    return gradient_penalty
```

```python
### Hinge loss + gradient penalty
D_train_loss = dis_loss(D_result_fake, D_result_real) + lambda_gp*gradient_penalty
```

左下圖即為WGAN-GP的Gradient penalty，來對model training來做 regularization。

試用心得是，在hinge loss外也加上這個term，對整個training是有較穩定，起伏變小。不過training history中，沒加gradient penalty的有出現更高一點點的test_accuracy。所以有沒有加這個penalty項的trained model參數都備存，等new_test data出來再來比較。

Implementation Details-Hyper Parameters

```python
######## data_loader #########
normalize = transforms.Normalize(mean=[0.5, 0.5, 0.5],
                                 std=[0.5, 0.5, 0.5])
data_transform = transforms.Compose([
        transforms.Resize((64,64)),
        # transforms.RandomAffine(0, shear=10, scale=(0.8,1.2)),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        normalize
        ])
```

左上圖為dataloader transform的相關參數

```python
##### training parameters
z_dim = 100
batch_size = 32
lr = 0.0002
train_epoch = 60
# clip_value = 0.03
dis_loss = L.DisLoss(loss_type='hinge')
gen_loss = L.GenLoss(loss_type='hinge')
G_optimizer = optim.Adam(G.parameters(), lr=lr, betas=(0.3, 0.9))
D_optimizer = optim.Adam(D.parameters(), lr=lr, betas=(0.3, 0.9))
lr_decay = np.linspace(lr, lr/500, train_epoch)
n_critic = 2
```

左中圖為GAN training的相關參數

值得一提的是，我是參考projection paper方法的linear decay learning rate。

Loss採用paper的hinge loss。

優化器用adam，並且beta1=0.3，beta2=0.9

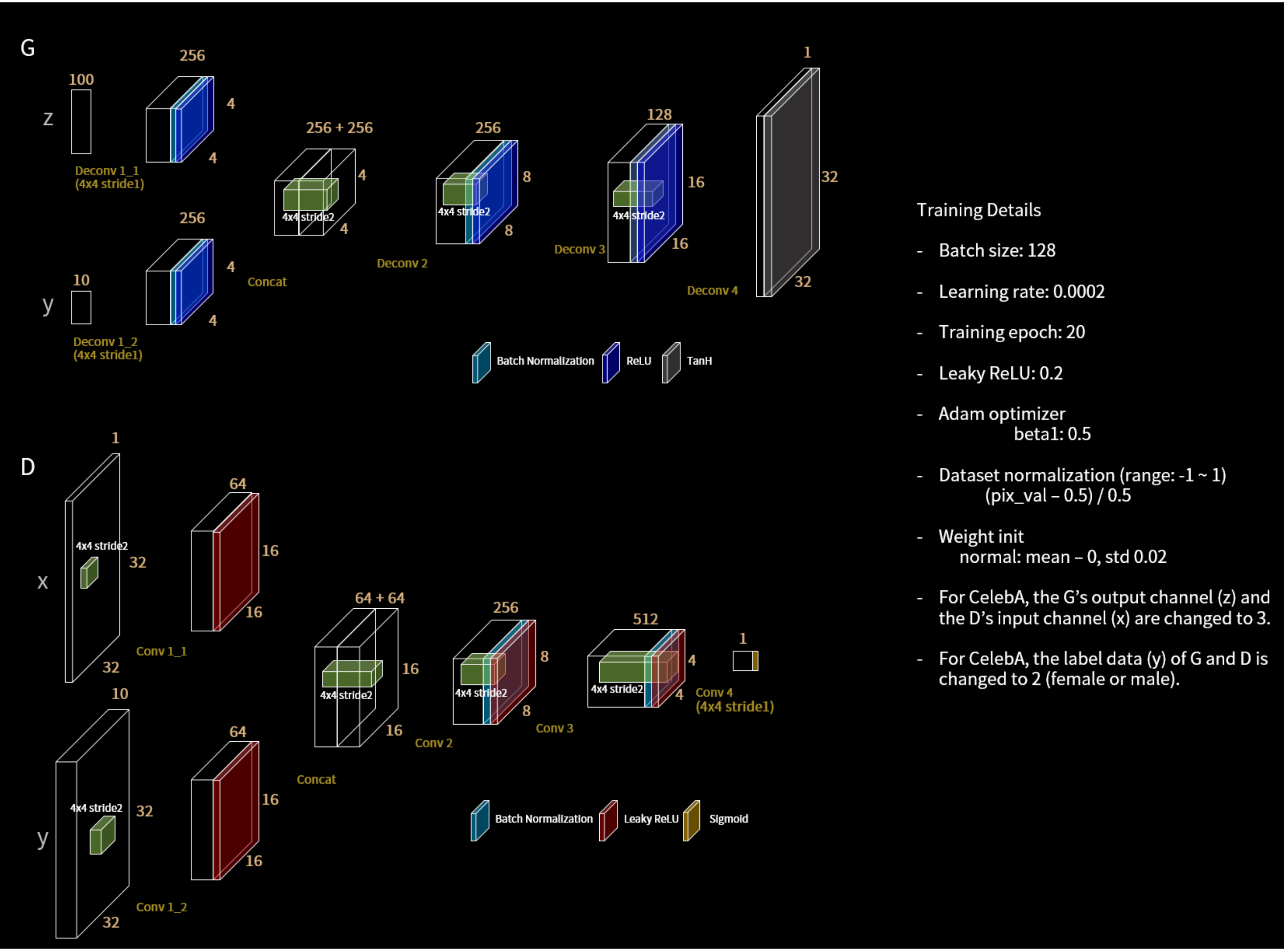
```python
'''######## train Generator G ######'''
if (epoch+1) >= 20:
    n_critic=3
if (epoch+1) >= 40:
    n_critic=4

if i_batch % n_critic == 0:
    G.zero_grad()
```

n_critic即為每update D的若干次後，才update G一次。

並且我將n_critic依每20個epoch共分成三個階段增加。

Results and Discussion

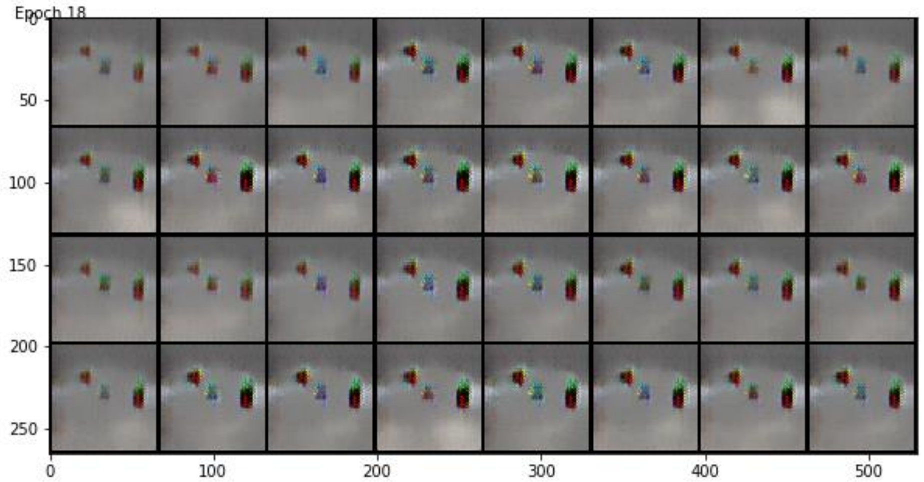
Results and Discussion– Training Experience1



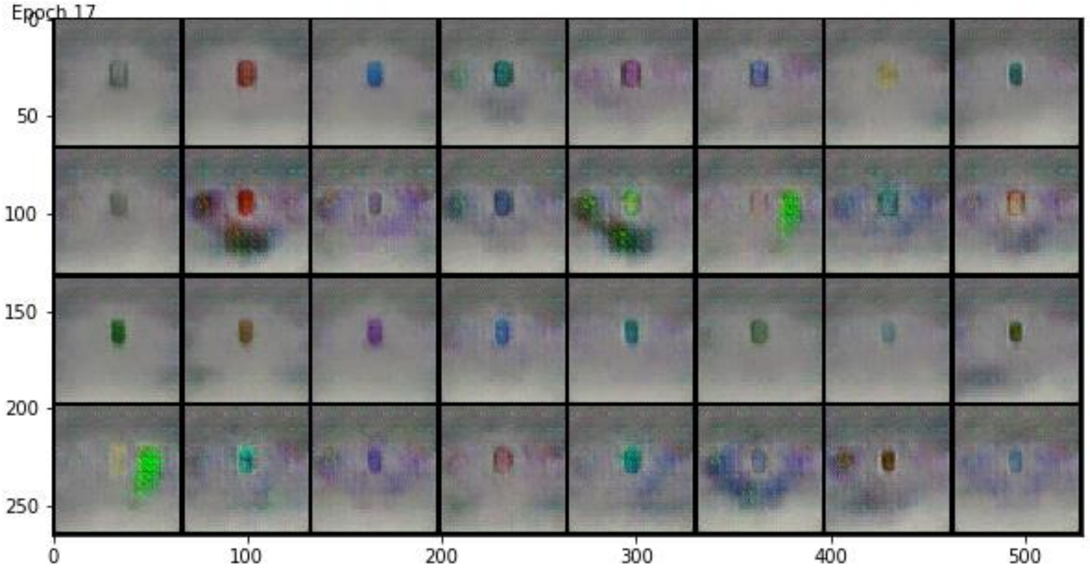
https://github.com/znxlwm/pytorch-MNIST-CelebA-cGAN-cDCGAN

左圖是我一開始從cDCGAN開始學習GAN的build up和training，參考的resource如左下連結。

然而發現mode collapse的問題相當嚴重。

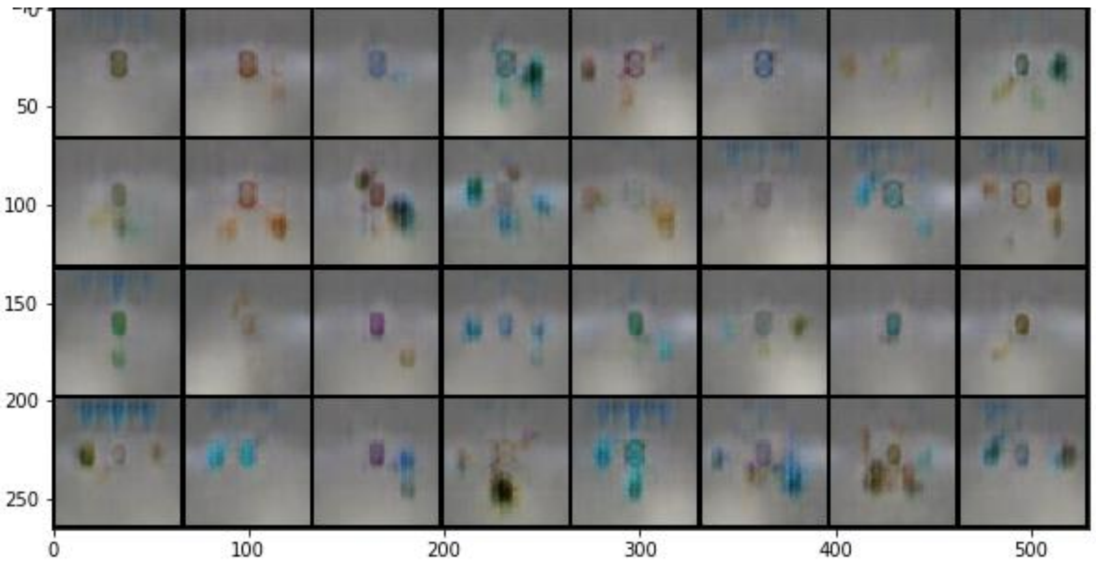


Results and Discussion– Training Experience2



左上圖Least Square GAN(LSGAN)的結果。

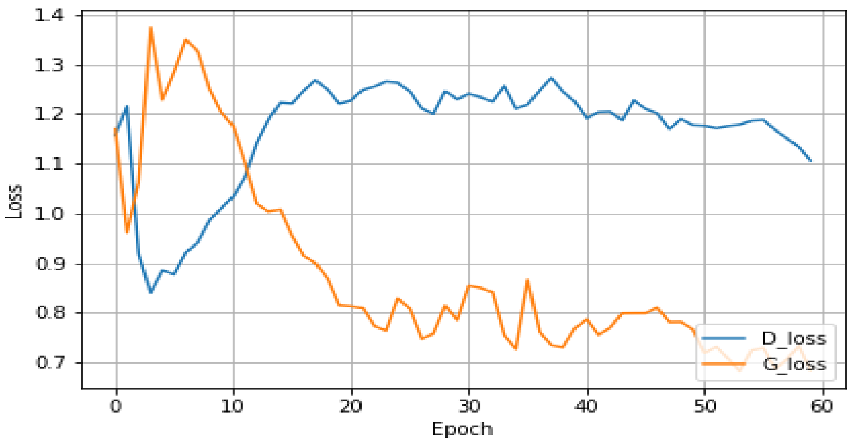
為了解決上張slide中，mode collapse的問題，我參考LSGAN把Discriminator的sigmoid output改成linear，也就是從classification變成regression的概念，但問題還是沒解決。(至少顏色變豐富了)

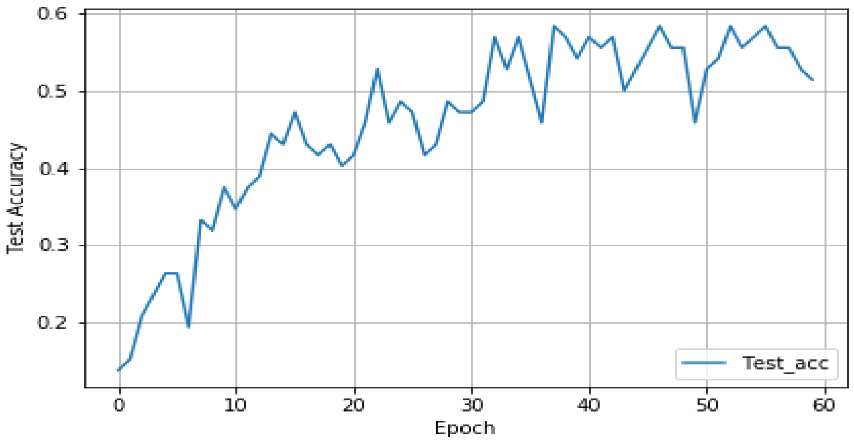


左下圖為cDCGAN加上WGAN-GP的手法，也就是改成Wasserstein distance based的loss、加weight clipping、加gradient penalty。Objects的生成有更豐富了、mode collapse問題解決了，但是畫質很差。

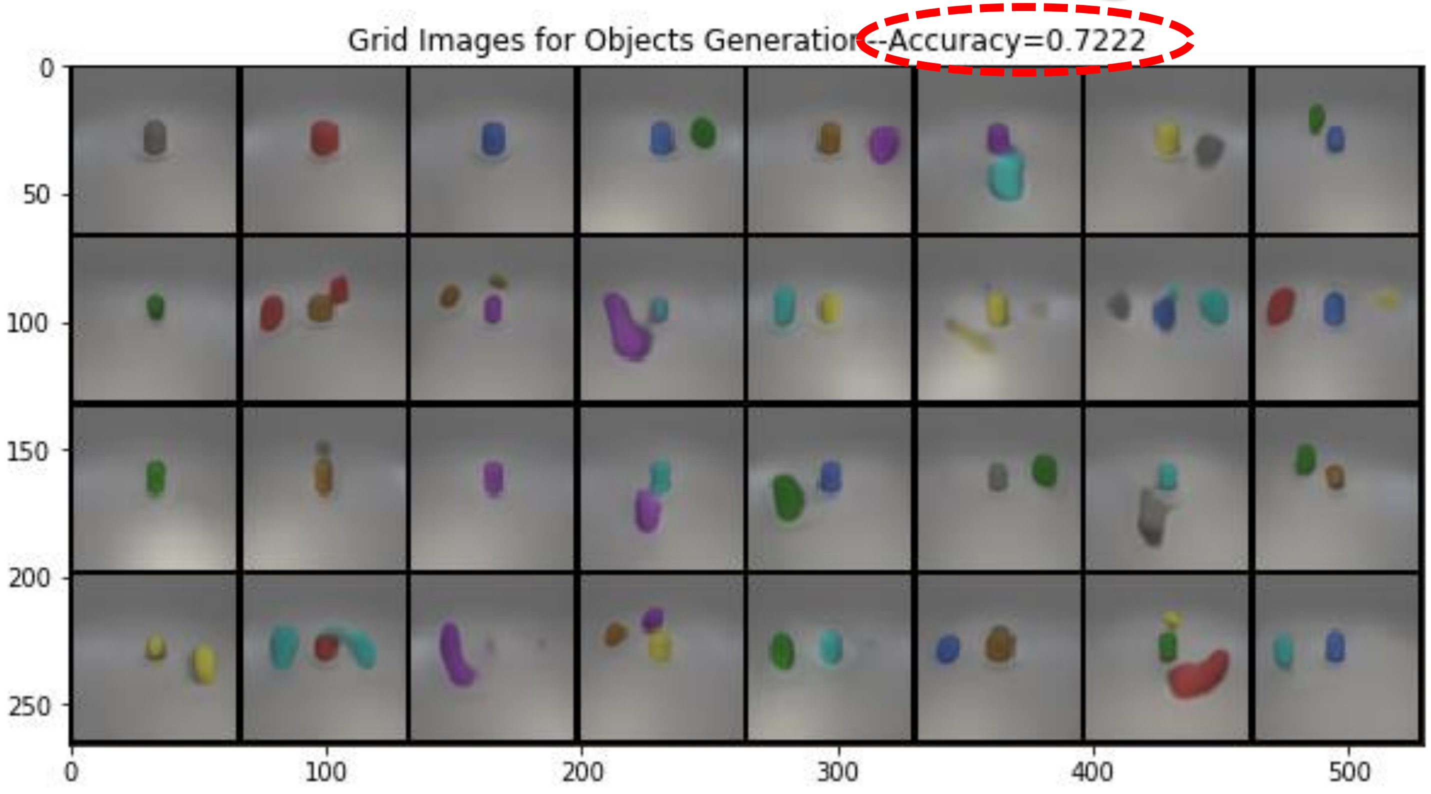
入門練習到這邊，而最後打算研究projection discriminator的方法。

Results and Discussion– Best Training Results






```
learning_rate=1.923e-05
epoch:56 -- iter:100 complete! (test_acc=0.6250)
epoch:56 -- iter:200 complete! (test_acc=0.6111)
epoch:56 -- iter:300 complete! (test_acc=0.5556)
epoch:56 -- iter:400 complete! (test_acc=0.5972)
epoch:56 -- iter:500 complete! (test_acc=0.5833)
[56/60] - ptime: 198.75, loss_d: 1.188, loss_g: 0.729
Classification accuracy: 0.5833
```



左上圖為Generator和Discriminator的Training loss history
左中圖為test generation accuracy的history(by epochs)。
左下圖為epoch=56時的第100個iteration的test_acc達到最高的0.625
然後我用這個model generator參數來生成上圖的grid images，並且幾次不同noise的forward後，這個generator的test_acc有達到0.7222

Results and Discussion-Training Setting

```
###### training parameters
z_dim = 100
batch_size = 32
lr = 0.0002
train_epoch = 60
# clip_value = 0.03
dis_loss = L.DisLoss(loss_type='hinge')
gen_loss = L.GenLoss(loss_type='hinge')
G_optimizer = optim.Adam(G.parameters(), lr=lr, betas=(0.3, 0.9))
D_optimizer = optim.Adam(D.parameters(), lr=lr, betas=(0.3, 0.9))
lr_decay = np.linspace(lr, lr/500, train_epoch)
n_critic = 2
```

Loss採用paper的hinge loss。而我也試著加過WGAN-GP的Gradient penalty進training loss，是有讓training趨於穩定，但是test_acc還是沒加penalty較高，所以最後還是把penalty拿掉。

優化器用adam，並且beta1=0.3，beta2=0.9。Paper的beta1=0，但經實驗後，在iclevr的data，beta1=0.3最佳，太大太小都不行。

```
'''######### train Generator G #######'''
if (epoch+1) >= 20:
    n_critic=3
if (epoch+1) >= 40:
    n_critic=4

if i_batch % n_critic == 0:
    G.zero_grad()
```

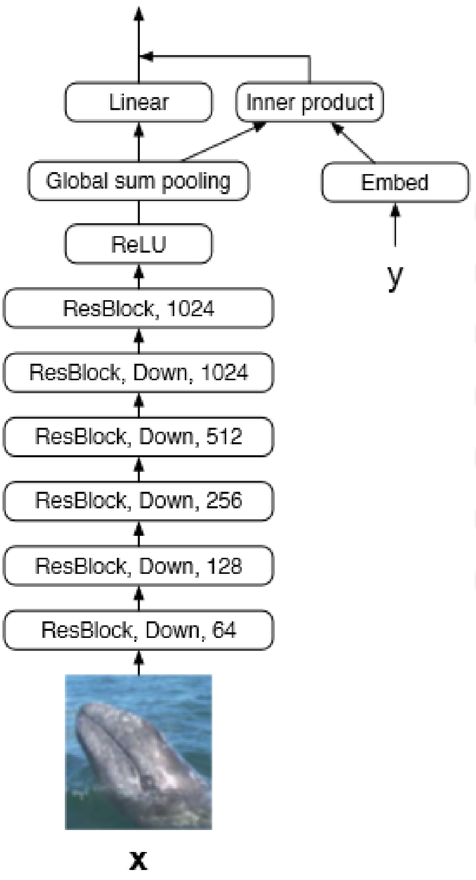
左中圖為我這次lab的GAN training的相關參數

關於batch_size，我的心得是，若n_critic都等於1的話，其實64和32沒多大差異。但為了先train D的若干次再train G，則用batch=32的效果最佳，也較快收斂。

值得一提的是，我參考projection paper方法採linear decay的learning rate，確實比原本用階梯式decay來的好，經幾次觀察training histroy後，推測是每次training都相當不一樣，若用階梯式decay，很難設定在好的decay時機。

n_critic即為每update D的若干次後，才update G一次。並且我將n_critic依每20個epoch共分成三個階段增加。n_critic最前面20epoch為2，中間為3，最後20個epochs時為4。在batch=32時，這樣做可以達到最好的效果。而在batch=64時，這樣做完全沒有improve的效果，可能是update的頻率太低。

Results and Discussion-Best Model Setting1




```
self.l_y_embed = utils.spectral_norm(nn.Embedding(num_classes, num_features * 16))
```

```
def forward(self, x, y_emb=None, y_lin=None):
    y_cond = self.cond_emb_in(y_emb).view(y_emb.size(0), -1,
                                          self.num_features, self.num_features)
    h = x
    h = torch.cat([x, y_cond],1)
    h = self.block1(h)
    h = self.block2(h)
    # h = torch.cat([h, y_cond_conv],1)
    h = torch.cat([h, y_cond_2],1)
    h = self.block3(h)
    h = self.block4(h)
    h = self.block5(h)
    h = self.activation(h)
    ##### Global pooling #####
    h = torch.sum(h, dim=(2, 3))
    output = self.l6(h)
    if y_emb is not None:
        l_y_sum = self.l_y_embed(y_emb)

    ##### Inner products for Embedding projections of 1 to 3 objects #####
    output1 = torch.sum(torch.mul(l_y_sum[:,0,:] ,h), dim=1, keepdim=True)
    output2 = torch.sum(torch.mul(l_y_sum[:,1,:] ,h), dim=1, keepdim=True)
    output3 = torch.sum(torch.mul(l_y_sum[:,2,:] ,h), dim=1, keepdim=True)
    output = output + (output1 + output2 +output3)/3

    return output
```

Discriminator一個對於test_acc有顯著影響的是，我把三個object label做embedding後，改成分別跟global sum pooling的vector內積再相加。

Batch=32的input之下，原本object label在embedding後的tensor size為[32,3,1024]，而我把dim=1先做平均後，再squeeze變成[32,1024]後才跟global pooling vector內積，這樣效果不好。

後來改成如紅框中的dim=1分別slice出來內積，三個內積值平均後，再跟linear的output相加，有讓test_acc顯著提升。

Results and Discussion-Best Model Setting2

```python
class ConditionalBatchNorm2d(nn.BatchNorm2d):
    """Conditional Batch Normalization"""
    def __init__(self, num_features, eps=1e-05, momentum=0.1,
                 affine=False, track_running_stats=True):
        super(ConditionalBatchNorm2d, self).__init__(
            num_features, eps, momentum, affine, track_running_stats)

    def forward(self, input, weight, bias, **kwargs):
        self._check_input_dim(input)

        exponential_average_factor = 0.0
        if self.training and self.track_running_stats:
            self.num_batches_tracked += 1
            if self.momentum is None:  # use cumulative moving average
                exponential_average_factor = 1.0 / self.num_batches_tracked.item()
            else:  # use exponential moving average
                exponential_average_factor = self.momentum
        output = F.batch_norm(input, self.running_mean, self.running_var,
                              self.weight, self.bias,
                              self.training or not self.track_running_stats,
                              exponential_average_factor, self.eps)

        weight1, weight2, weight3 = weight[:,0,:], weight[:,1,:], weight[:,2,:]
        bias1, bias2, bias3 = bias[:,0,:], bias[:,1,:], bias[:,2,:]
        size = output.size()
        weight1 = weight1.unsqueeze(-1).unsqueeze(-1).expand(size)
        bias1 = bias1.unsqueeze(-1).unsqueeze(-1).expand(size)
        weight2 = weight2.unsqueeze(-1).unsqueeze(-1).expand(size)
        bias2 = bias2.unsqueeze(-1).unsqueeze(-1).expand(size)
        weight3 = weight3.unsqueeze(-1).unsqueeze(-1).expand(size)
        bias3 = bias3.unsqueeze(-1).unsqueeze(-1).expand(size)

        out1 = weight1 * output + bias1
        out2 = weight2 * output + bias2
        out3 = weight3 * output + bias3
        output = (out1 + out2 + out3)/3
        return output
```

跟上一張slide的想法類似，我把object label做embedding後的三組weight & bias，不取平均，先分別跟batch_norm的output做乘加，然後才取平均。

註：CategoricalConditionalBatchNorm是project discriminator這篇paper的另一個亮點，有別於concate的方式來做condition。

```python
class CategoricalConditionalBatchNorm2d(ConditionalBatchNorm2d):
    def __init__(self, num_classes, num_features, eps=1e-5, momentum=0.1,
                 affine=False, track_running_stats=True):
        super(CategoricalConditionalBatchNorm2d, self).__init__(
            num_features, eps, momentum, affine, track_running_stats)

        self.weights_emb = nn.Embedding(num_classes, num_features)
        self.biases_emb = nn.Embedding(num_classes, num_features)

        self._initialize()

    def _initialize(self):
        init.ones_(self.weights_emb.weight.data)
        init.zeros_(self.biases_emb.weight.data)

    def forward(self, input, c_emb, **kwargs):
        weight_emb = self.weights_emb(c_emb)
        bias_emb = self.biases_emb(c_emb)

        return super(CategoricalConditionalBatchNorm2d, self).forward(
                                    input, weight_emb, bias_emb)
```

Results and Discussion-Best Model Setting3

雖然projection d這篇paper的方法，並沒有混用concat的方法，但我嘗試分別在generator(左圖)與discriminator(右圖)的input端，將conditional embedding reshape後concat進來，結果發現對model的收斂很有幫助。

```python
class ResNetGenerator(nn.Module):
    """Generator generates 64x64."""
    def __init__(self, num_features=64, dim_z=100, bottom_width=4,
                 activation=F.relu, num_classes=0, distribution='normal'):
        super(ResNetGenerator, self).__init__()
        self.num_features = num_features
        self.dim_z = dim_z
        self.bottom_width = bottom_width
        self.activation = activation
        self.num_classes = num_classes
        self.distribution = distribution
        self.cond_emb = nn.Embedding(num_classes, num_features ** 2)

        self.l1 = nn.Linear(dim_z, (16-4) * num_features * bottom_width ** 2)
        self.block2 = Block(num_features * (16-4) * 2, num_features * 8,
                            activation=activation, upsample=True, num_classes=num_classes)
        self.block3 = Block(num_features * 8, num_features * 4,
                            activation=activation, upsample=True, num_classes=num_classes)
        self.block4 = Block(num_features * 4, num_features * 2,
                             activation=activation, upsample=True, num_classes=num_classes)
        self.block5 = Block(num_features * 2, num_features,
                             activation=activation, upsample=True, num_classes=num_classes)
        self.b6 = nn.BatchNorm2d(num_features)
        self.conv6 = nn.Conv2d(num_features, 3, 1, 1)

    def forward(self, z, y=None, **kwargs):
        y_cond = self.cond_emb(y).view(y.size(0), -1, self.bottom_width, self.bottom_width)
        h = self.l1(z).view(z.size(0), -1, self.bottom_width, self.bottom_width)
        h = torch.cat([h, y_cond], 1)
        for i in range(2, 6):
            h = getattr(self, 'block{}'.format(i))(h, y, **kwargs)
        h = self.activation(self.b6(h))
        return torch.tanh(self.conv6(h))
```

```python
        self.block1 = OptimizedBlock(3+3, num_features)
        self.block2 = Block(num_features, num_features * 2,
                            activation=activation, downsample=True)
        self.block3 = Block(num_features * 2, num_features * 4,
                            activation=activation, downsample=True)
        self.block4 = Block(num_features * 4, num_features * 8,
                            activation=activation, downsample=True)
        self.block5 = Block(num_features * 8, num_features * 16,
                            activation=activation, downsample=True)
        self.l6 = utils.spectral_norm(nn.Linear(num_features * 16, 1))
        if num_classes > 0:
            self.cond_emb_in = utils.spectral_norm(
                nn.Embedding(num_classes, num_features ** 2))
            self.l_y_embed = utils.spectral_norm(
                nn.Embedding(num_classes, num_features * 16))
        self._initialize()

    def forward(self, x, y_emb=None, y_lin=None):
        y_cond = self.cond_emb_in(y_emb).view(y_emb.size(0), -1,
                                               self.num_features, self.num_features)
        h = x
        h = torch.cat([x, y_cond],1)
        h = self.block1(h)
        h = self.block2(h)
        h = self.block3(h)
        h = self.block4(h)
        h = self.block5(h)
        h = self.activation(h)
        ##### Global pooling #####
        h = torch.sum(h, dim=(2, 3))
        output = self.l6(h)
        if y_emb is not None:
            l_y_sum = self.l_y_embed(y_emb)
```

The End