[Lab6] Let's Play GANs!

0616014 楊政道

May 26, 2020

1 Introduction

1.1 實驗目的

本次實驗目的為使用生成對抗網路 (Generative Adversarial Networks, 簡稱 GAN), 來完成給定條件下的圖片生成。欲生成的圖片由 8 種顏色和 3 種形狀的物體, 最多 3 個所陳列在某個全灰背景下的圖片。

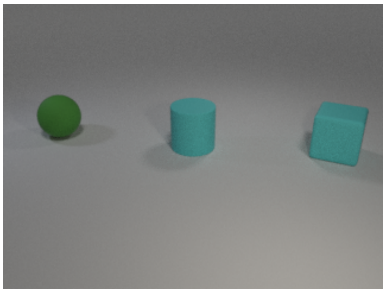

Figure 1: 訓練資料範例

2 Implementation details

2.1 DataLoader

由於這次的資料是圖片, 為了避免讀取圖片佔用到太多的計算資源倒置 GPU 使用率太低, 因此先對圖片預先存成 torch tensor 的二進制檔案, 並且預先做好圖片的轉換 (Resize, ToTensor 和 Normalize)。

```
class Dataset(Dataset):

    def __init__(self, path):
        self.labels = torch.load('dataset/labels.pth').long()
        self.images = torch.load('dataset/images.pth')

    def __getitem__(self, index):
        images, labels = [], []
        for i in range(3):
            images.append(self.images[index * 3 + i])
            labels.append(self.labels[index * 3 + i])
        images = torch.stack(images)
        labels = torch.stack(labels)
        return images, labels

    def __len__(self):
        return self.labels.size(0) // 3
```

2.2 Task

原本的問題是給一個不固定長度的陣列, 輸出一張有陣列中的物體的圖。考量到不固定長度訓練起來有難度, 因此將原本的問題改成, 給一張圖和一個要加上去的物體, 輸出原本的圖加上該物體的 RNN 問題。

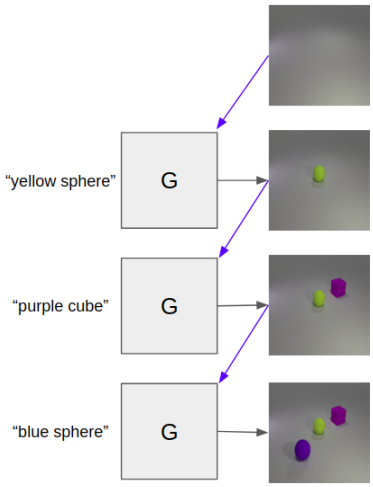

Figure 2: Task

2.3 GAN

GAN 的整體架構就是 Discriminator, Generator, 還有一個 RNN, RNN 的作用在於處理收到的 condition sequence, RNN 輸出的 output 則用於 GAN 的 condition, hidden 則繼續傳下去等待下一個 input。

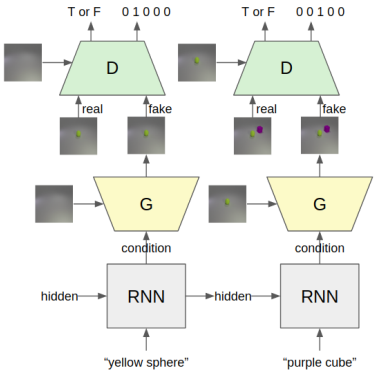

Figure 3: GAN 架構

2.4 Discriminator

Discriminator 目的要分辨輸入的圖為真還是假, 並且分辨上面的物件類別。輸入為上一張圖跟目標圖, 各做三次的 DownConv 後相減, 可以獲得兩張圖差異的 feature map, 後面再根據這個差異來判斷圖的真假以及要加上去的物件的類別。

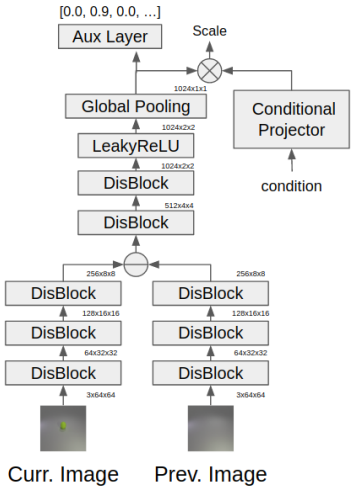

Figure 4: Discriminator 架構

DisBlock 參考 ResNet 的架構, 經過一層 DisBlock 解析度都會/2。

Loss function 有 4 個, 分別為 d_real, d_fake, aux_loss, gradient_penalty。

$$Loss = -d\_real + d\_fake + aux\_loss + gradient\_penalty$$

使用到了 Conditional Projector, WGAN-GP, spectral_norm。

2.5 Generator

Generator 目的要根據 condition 來生出對應的圖片。輸入為一個 noise 和一個 conditional vector，中間層時接上 prev_image，輸出一個 3x64x64 的圖片。

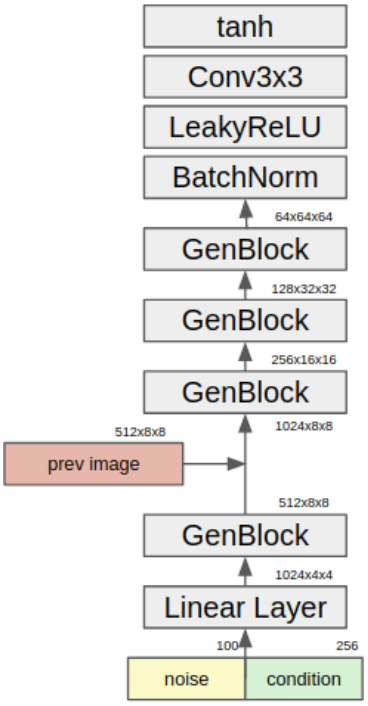

Figure 5: Generator 架構

Loss function 只有一項, 要讓生出來的圖被 discriminator 辨認為真。

$$Loss = -d\_fake$$

3 Discussion

3.1 GAN 學習不穩

原始的 GAN 學習上非常不穩, 同樣的參數跑不同次結果會有很大的不同。原因為當 Discriminator 學習太過快速讓 Generator 沒辦法跟上時, 對於 Generator 來說的 gradient 太小, 所以會訓練失敗。解決的方法為弱化 Discriminator。

3.2 Discriminator 後的 Sigmoid Layer

Sigmoid 會讓遠離 0 兩端的 gradient 太小, 導致 gradient desent 有困難, 將 sigmoid 層拔掉之後輸出一個 scale, 可以解決此問題。

3.3 gradient penalty

由於 discriminator 的 sigmoid layer 被拔掉了, 為了避免 discriminator 沒有下限的往下學習，因此需要加上 regurization 項來穩定 discriminator 的學習。下面為 gradient penalty 的實作 code。

```
def __gradient_penalty(self, d_real, real_images):
    batch_size = d_real.size(0)
    grad = autograd.grad(
        outputs = d_real.sum(),
        inputs = real_images,
        create_graph=True,
        retain_graph=True,
        only_inputs=True
```

```
        )[0]
        return (grad ** 2).view(batch_size, -1).sum(1)
```

3.4 Hyper parameters

Discriminator 的 optimzer 為 Adam(lr=4e-4, betas=(0, 0.9)), Generator 的 optimizer 為 Adam(lr=1e-4, betas=(0, 0.9)), RNN 的 optimizer 為 Adam(lr=1e-3), 訓練 140epoches。

4 Result

如下為 test.json 的結果

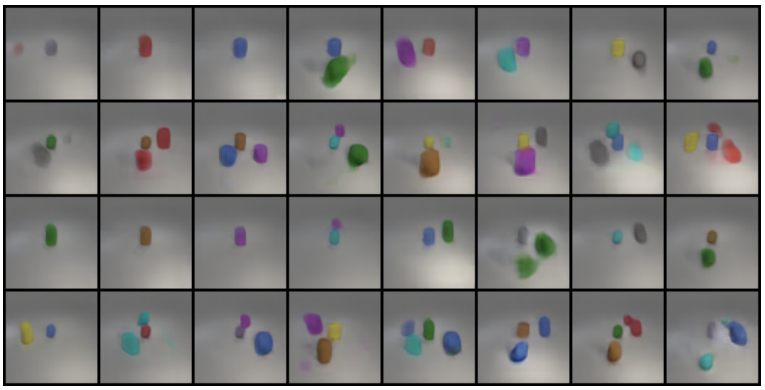

Figure 6: test.json 的結果, accuracy=0.89

隨意生一筆測資的結果

Figure 7: custom.json 的結果, accuracy=0.83

5 Appendix

5.1 Generator Structure

```
(gen): GenNet(
    (c_proj): Linear(in_features=256, out_features=256, bias=True)
    (l0): Linear(in_features=356, out_features=16384, bias=True)
    (block1): GenBlock(
      (ac): LeakyReLU(negative_slope=0.01)
      (upsample): Upsample(scale_factor=2.0, mode=nearest)
      (cbn1): ConditionalBatchNorm2d(
        (bn): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=False, track_running_stats=
            True)
        (fc): Linear(in_features=256, out_features=2048, bias=True)
```

```
    )
    (cbn2): ConditionalBatchNorm2d(
      (bn): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=False, track_running_stats=True
          )
      (fc): Linear(in_features=256, out_features=1024, bias=True)
    )
    (c0): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1))
    (c1): Conv2d(1024, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (c2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  )
  (block2): GenBlock(
    (ac): LeakyReLU(negative_slope=0.01)
    (upsample): Upsample(scale_factor=2.0, mode=nearest)
    (cbn1): ConditionalBatchNorm2d(
      (bn): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=False, track_running_stats=
          True)
      (fc): Linear(in_features=256, out_features=2048, bias=True)
    )
    (cbn2): ConditionalBatchNorm2d(
      (bn): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=False, track_running_stats=True
          )
      (fc): Linear(in_features=256, out_features=512, bias=True)
    )
    (c0): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1))
    (c1): Conv2d(1024, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (c2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  )
  (block3): GenBlock(
    (ac): LeakyReLU(negative_slope=0.01)
    (upsample): Upsample(scale_factor=2.0, mode=nearest)
    (cbn1): ConditionalBatchNorm2d(
      (bn): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=False, track_running_stats=True
          )
      (fc): Linear(in_features=256, out_features=512, bias=True)
    )
    (cbn2): ConditionalBatchNorm2d(
      (bn): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=False, track_running_stats=True
          )
      (fc): Linear(in_features=256, out_features=256, bias=True)
    )
    (c0): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1))
    (c1): Conv2d(256, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (c2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  )
  (block4): GenBlock(
    (ac): LeakyReLU(negative_slope=0.01)
    (upsample): Upsample(scale_factor=2.0, mode=nearest)
    (cbn1): ConditionalBatchNorm2d(
      (bn): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=False, track_running_stats=True
          )
      (fc): Linear(in_features=256, out_features=256, bias=True)
    )
    (cbn2): ConditionalBatchNorm2d(
      (bn): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=False, track_running_stats=True)
      (fc): Linear(in_features=256, out_features=128, bias=True)
    )
    (c0): Conv2d(128, 64, kernel_size=(1, 1), stride=(1, 1))
    (c1): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (c2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  )
  (bn): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (ac): LeakyReLU(negative_slope=0.01)
  (conv): Conv2d(64, 3, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (tanh): Tanh()
)
```

5.2 Discriminator Structure

```
(dis): DisNet(
    (ac): LeakyReLU(negative_slope=0.01)
    (block1): DisBlock(
      (c0): Conv2d(3, 64, kernel_size=(1, 1), stride=(1, 1))
      (c1): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (c2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (ac): LeakyReLU(negative_slope=0.01)
      (downsample): AvgPool2d(kernel_size=2, stride=2, padding=0)
    )
    (block2): DisBlock(
      (c0): Conv2d(64, 128, kernel_size=(1, 1), stride=(1, 1))
      (c1): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (c2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (ac): LeakyReLU(negative_slope=0.01)
      (downsample): AvgPool2d(kernel_size=2, stride=2, padding=0)
    )
    (block3): DisBlock(
      (c0): Conv2d(128, 256, kernel_size=(1, 1), stride=(1, 1))
      (c1): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (c2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (ac): LeakyReLU(negative_slope=0.01)
      (downsample): AvgPool2d(kernel_size=2, stride=2, padding=0)
    )
    (block4): DisBlock(
      (c0): Conv2d(256, 512, kernel_size=(1, 1), stride=(1, 1))
      (c1): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (c2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (ac): LeakyReLU(negative_slope=0.01)
      (downsample): AvgPool2d(kernel_size=2, stride=2, padding=0)
    )
    (block5): DisBlock(
      (c0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(1, 1))
      (c1): Conv2d(512, 1024, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (c2): Conv2d(1024, 1024, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (ac): LeakyReLU(negative_slope=0.01)
      (downsample): AvgPool2d(kernel_size=2, stride=2, padding=0)
    )
    (l6): Linear(in_features=1024, out_features=1, bias=True)
    (condition_projector): Sequential(
      (0): Linear(in_features=256, out_features=1024, bias=True)
      (1): ReLU()
      (2): Linear(in_features=1024, out_features=1024, bias=True)
    )
    (aux_objective): Sequential(
      (0): Linear(in_features=1024, out_features=256, bias=True)
      (1): ReLU()
      (2): Linear(in_features=256, out_features=24, bias=True)
    )
  )
```