CAVEDU教育團隊部落格

[ 技術教學文-格式 ]

|  |  |  |  |
| --- | --- | --- | --- |
| **標題** | 有條件的生成手寫數字 | | |
|  | | | |
| **分類** |  | **標籤** |  |
| **撰寫/攝影** |  | | |
| **前情提要** | 文字＋文章連結........ | | |
| **時間** |  | **材料表** |  |
| **成本** |  |
| **難度** |  |
| **前言** | | | |
| 上次已經教大家DCGAN的建構方法並利用DCGAN來生成手寫數字，但你可以發現我們生成的數字都是隨機跳動的，訓練的時候也沒將數據集中的標籤取出來用；所以，如果想要生成出特定的數字則需要在訓練的時候加入標籤，也就是我們這次想要探討的cDCGAN (Conditional Deep Convolution Adversial Network)，這次我們將延續上次的Mnist數據以及程式來做改動，用最簡單的程式碼來完成cDCGAN。 | | | |
| **內文** | | | |
| |  |  | | --- | --- | | **作者** | **張嘉鈞** | | **難度** | 中偏難 | | **材料表** | 1. Jetson Nano 2. PC or Notebook |  DCGAN 到 cDCGAN 先來稍微複習一下DCGAN (深度捲積生成對抗網路)，字面上就是利用捲積網路的架構來做生成對抗，主要由生成器與鑑別器所構成，如下圖所示：    生成器會將一組雜訊或稱做潛在空間的張量轉換成一張照片，這張照片再經由鑑別器去判斷圖片是否夠真實，越接近0越假；越接近1越真。  由於我們在訓練的時候其實是沒有載入標籤的！所以他生成的時候都是隨機生成，為了能限制特定的輸出我們必須載入標籤，概念圖就會變成下面這張：    透過標籤的導入，讓生成器知道要生成的對象是哪一個數字，並且鑑別器訓練的目標變成「圖像是否真實」加上「是否符合該類別」，cDCGAN跟DCGAN相比，訓練的結果通常會比較好，因為DCGAN神經網路是盲目的去生成，而cDCGAN則是會將生成的範圍縮小，整體而言會收斂更快且更好。 將標籤合併於資料中 首先我們要先了解如何加入標籤，對於DCGAN來說有兩種加入標籤的方法，第一個是一開始就將圖片或雜訊跟標籤合併；另一個方法則是在深層做合併，讀者們在實作的時候可以自行調整看看差異，那較常見的做法是深層合併，而我寫的也是！   |  |  | | --- | --- | |  |  | | 潛層合併，先合併再輸入網路 | 深層合併：各別輸入後再合併 |   其中詳細的差別我還沒涉略到，不過選定了深層合併接著就可以先來實作生成器跟鑑別器了。首先先來建構生成器，可以參考上一篇DCGAN的程式碼，這邊幫大家整理了一張概念圖：    輸入的z是維度為 ( 100, 1, 1) 的雜訊，為了將標籤跟雜訊能合併，必須轉換到相同大小也就是 (1, 1)，可以看到這邊 y 的維度是 ( 10, 1, 1 ) 原因在於我們將原先阿拉伯數字的標籤轉成 onehot 編碼格式，如下圖所示。    OneHot編碼主要在於讓標籤離散，如果將標籤都用阿拉伯數字表示，對於神經網路而言他們屬於連續性的數值或許會將前後順序、距離給考慮進去，但是用onehot之後將可以將各類標籤單獨隔開並且對於彼此的距離也會相同。  **建立Generator**  接下來是程式的部分拉～如何在神經網路中做分流又合併，其實對於PyTorch而言非常的簡單只要在forward的地方做torch.cat就可以了。首先一樣要先定義網路層，我們定義了三個 Sequential，其中input\_x是給圖像用的所以第一層deconv的輸入維度是z\_dim；而input\_y則是標籤用所以deconv的輸入是label\_dim，可以對照上面的圖片看看：   |  | | --- | | def \_\_init\_\_(self, z\_dim, label\_dim):  super(Generator, self).\_\_init\_\_()  self.input\_x = nn.Sequential(  # input is Z, going into a convolution  nn.ConvTranspose2d(z\_dim, 256, 4, 1, 0, bias=False),  nn.BatchNorm2d(256),  nn.ReLU(True),  # image size = (1-1)\*1 - 2\*0 + 4 = 4  )  self.input\_y = nn.Sequential(  # input is Z, going into a convolution  nn.ConvTranspose2d( label\_dim, 256, 4, 1, 0, bias=False),  nn.BatchNorm2d(256),  nn.ReLU(True),  # image size = (1-1)\*1 - 2\*0 + 4 = 4  )  self.concat = nn.Sequential(    # 因為 x 跟 y 水平合併所以要再乘以 2  nn.ConvTranspose2d(256\*2, 128, 4, 2, 1, bias=False),  nn.BatchNorm2d(128),  nn.ReLU(True),  # image size = (4-1)\*2 - 2\*1 + 4 = 8  nn.ConvTranspose2d( 128, 64, 4, 2, 1, bias=False),  nn.BatchNorm2d(64),  nn.ReLU(True),  # image size = (8-1)\*2 - 2\*1 + 4 = 16    nn.ConvTranspose2d( 64, 1, 4, 2, 3, bias=False),  nn.Tanh()  # image size = (16-1)\*2 - 2\*3 + 4 = 28  ) |   接下來看 forward的部分，可以看到我們在向前傳遞的時候要丟入兩個數值，雜訊跟標籤，將x跟y丟進各自的Sequential中，接著我們使用torch.cat將x, y從橫向 ( dim=1 ) 合併後再進到concat中。   |  | | --- | | def forward(self, x, y):  x = self.input\_x(x)  y = self.input\_y(y)  out = torch.cat([x, y] , dim=1)  out = self.concat(out)  return out |   接下來可以試著將網路架構顯示出來，我們直接使用print也使用torchsummary來顯示，你可以發現其實你沒辦法看出網路分支再合併的狀況   |  | | --- | | def print\_div(text):    div='\n'  for i in range(60): div += "="  div+='\n'  print("{} {:^60} {}".format(div, text, div))    """ Define Generator """  G = Generator(100, 10)    """ Use Print"""  print\_div('Print Model Directly')  print(G)    """ Use Torchsummary """  print\_div('Print Model With Torchsummary')  test\_x = (100, 1, 1)  test\_y = (10, 1, 1)  summary(G, [test\_x, test\_y], batch\_size=64) |     所以我決定使用更圖像化一點的方式來視覺化我們的網路架構，現在有不下10種的圖形化方式，我舉兩個例子：Tensorboard、hiddenlayer。  **視覺化模型**  Tensorboard 是Google 出的強大視覺化工具，一般的文字、數值、影像、聲音都可以動態的紀錄在上面，一開始只支援Tensorflow 但是 PyTorch 1.2 之後都包含在其中 ( 但是要使用的話還是要先安裝tensorboard ) ，你可以直接從 torch.utils.tensorboard 中呼叫 Tensorboard，首先需要先實體化 SummaryWritter，接著直接使用add\_graph即可將圖片存到伺服器上   |  | | --- | | """ Initial Parameters"""  batch\_size = 1  test\_x = torch.rand(batch\_size, 100, 1, 1)  test\_y = torch.rand(batch\_size, 10, 1, 1)    print\_div('Print Model With Tensorboard')  print('open terminal and input "tensorboard --logdir=runs"')  print('open browser and key http://localhost:6006')  writer = SummaryWriter()  writer.add\_graph(G, (test\_x, test\_y))  writer.close() |   接下來要開啟伺服器，在終端機中移動到與程式碼同一層級的位置並且輸入：   |  | | --- | | > tensorboard –logdir=./runs |     一開始就可以看到 input > Generator 的箭頭有寫 2 tensor，而這些方塊都可以打開：    開啟後你可以看到更細部的資訊，也很清楚就可以看到支線合併的狀況。    每一次捲積後的形狀大小也都有顯示出來：    接下來簡單介紹一下hiddenlayer ，它不能用來取代高級API像是tensorboard之類的，它僅僅就是用來顯示神經網路模型，但是非常的輕巧所以我個人蠻愛使用它的，首先要先透過pip安裝hiddenlayer、graphviz：   |  | | --- | | > pip install hiddenlayer  > Pip install graphviz |   如果是用Jetson Nano的話，建議用 apt去裝 graphviz   |  | | --- | | $ sudo apt-get install graphviz |   接著用 build\_graph就能產生圖像也能直接儲存：   |  | | --- | | """ Initial Parameters"""  batch\_size = 1  test\_x = torch.rand(batch\_size, 100, 1, 1)  test\_y = torch.rand(batch\_size, 10, 1, 1)    print\_div('Print Model With HiddenLayer')  g\_graph = hl.build\_graph(G, (test\_x, test\_y))  g\_graph.save('./images/G\_grpah', format="jpg")  g\_graph |   因為太長了所以我截成兩半方便觀察，這邊就可以注意到前面的ConvTranspose、BatchNorm、ReLU是分開的，之後才合併這邊還特別給了一個Concat的方塊，我喜歡使用它的原因是簡單明瞭，捲積後的維度也都有寫下來，並且直接執行就可以看到結果，不用像Tensorboard還要再開啟服務 ( 小抱怨 )。      **建立Discriminator**    跟建立Generator的概念相似，我們要個別處理輸入的圖片跟標籤，所以依樣宣告兩個 Sequential 個別處理接著再將輸出 concate 在一起，主要要注意的是 y 的輸入為度為 (10, 28, 28)：   |  | | --- | | import torch  import torch.nn as nn  from torchsummary import summary    class Discriminator(nn.Module):    def \_\_init\_\_(self, c\_dim=1, label\_dim=10):    super(Discriminator, self).\_\_init\_\_()    self.input\_x = nn.Sequential(    # Input size = 1 ,28 , 28  nn.Conv2d(c\_dim, 64, (4,4), 2, 1),  nn.LeakyReLU(),  )  self.input\_y = nn.Sequential(    # Input size = 10 ,28 , 28  nn.Conv2d(label\_dim, 64, (4,4), 2, 1),  nn.LeakyReLU(),  )    self.concate = nn.Sequential(    # Input size = 64+64 ,14 , 14  nn.Conv2d(64\*2 , 64, (4,4), 2, 1),  nn.LeakyReLU(),    # Input size = (14-4+2)/2 +1 = 7  nn.Conv2d(64, 128, 3, 2, 1),  nn.LeakyReLU(),    # Input size = (7-3+2)/2 +1 = 4  nn.Conv2d(128, 1, 4, 2, 0),  nn.Sigmoid(),    # output size = (4-4)/2 +1 = 1  )    def forward(self, x, y):    x = self.input\_x(x)  y = self.input\_y(y)  out = torch.cat([x, y] , dim=1)  out = self.concate(out)  return out    D = Discriminator(1, 10)  test\_x = torch.rand(64, 1,28,28)  test\_y = torch.rand(64, 10,28,28)    writer = SummaryWriter()  writer.add\_graph(D, (test\_x, test\_y))  writer.close()    hl.build\_graph(D, (test\_x, test\_y)) |   視覺化的結果如下：    **數據處理**  神經網路都建置好就可以準備來訓練啦！當然第一步要先將數據處理好，那我個人自學神經網路的過程我覺得最難的就是數據處理了，這次數據處理有2個部分：   1. 宣告固定的雜訊跟標籤用來預測用 2. 將標籤轉換成onehot格式 ( scatter )   Onehot數據處理，在torch中可以直接使用scatter的方式，我在程式註解的地方有推薦一篇文章大家可以去了解scatter的概念，至於這邊我先附上實驗的程式碼：   |  | | --- | | """ OneHot 格式 之 scatter 應用"""  """ 超好理解的圖形化教學 https://medium.com/@yang6367/understand-torch-scatter-b0fd6275331c """    label =torch.tensor([1,5,6,9])  print(label, label.shape)      a = torch.zeros(10).scatter\_(0, label, 1)  print(a)    print('\n\n')  label\_=label.unsqueeze(1)  print(label\_, label\_.shape)  b = torch.zeros(4,10).scatter\_(1, label\_, 1)  print(b) |     接下來我們將兩個部分分開處理，先來處理測試用的雜訊跟標籤，測試用圖片為美個類別各10張，所以總共有100張圖片代表是100組雜訊及對應label：   |  | | --- | | """ 產生固定資料，每個類別10張圖(雜訊) 以及 對應的標籤，用於視覺化結果 """  temp\_noise = torch.randn(label\_dim, z\_dim) # (10, 100) 10張圖  fixed\_noise = temp\_noise  fixed\_c = torch.zeros(label\_dim, 1) # (10, 1 ) 10個標籤    for i in range(9):  fixed\_noise = torch.cat([fixed\_noise, temp\_noise], 0) #將每個類別的十張雜訊依序合併，維度1會自動boardcast  temp = torch.ones(label\_dim, 1) + i #依序將標籤對應上 0~9  fixed\_c = torch.cat([fixed\_c, temp], 0) #將標籤也依序合併    fixed\_noise = fixed\_noise.view(-1, z\_dim, 1, 1) #由於是捲積所以我們要將形狀轉換成二維的  print('Predict Noise: ', fixed\_noise.shape)  print('Predict Label (before): ', fixed\_c.shape, '\t\t\t', fixed\_c[50])    """ 針對 lael 做 onehot """  fixed\_label = torch.zeros(100, label\_dim) #先產生 [100,10] 的全0張量，100個標籤，每個標籤維度是 10  fixed\_label.scatter\_(1, fixed\_c.type(torch.LongTensor), 1) #轉成 onehot編碼 (1, ) -> (10, )  fixed\_label = fixed\_label.view(-1, label\_dim, 1, 1) #轉換形狀 (10, 1, 1 )  print('Predict Label (onehot): ',fixed\_label.shape, '\t\t', fixed\_label[50].view(1,-1), '\n') |   我在顯示的時候有將形狀從 (10,1)變成(1,10) 來方便做觀察：    接下來要幫訓練的數據做前處理，處理方式跟前面雷同，主要差別在要餵給鑑別器的標籤 ( fill ) 處理方式比較不同，從結果圖就能看的出來彼此不同的地方：   |  |  | | --- | --- | | """ 幫標籤做前處理，onehot for g, fill for d """  onehot = torch.zeros(label\_dim, label\_dim) # 產生 (10,10) 10個標籤，維度為10 (onehot)  onehot = onehot.scatter\_(1, torch.LongTensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]).view(label\_dim, 1), 1).view(label\_dim, label\_dim, 1, 1)  print('Train G label:',onehot[1].shape, '\n', onehot[1], '\n') # 假設我們要取得標籤 1 的 onehot (10,1,1)，直接輸入索引 1    fill = torch.zeros([label\_dim, label\_dim, image\_size, image\_size]) # 產生 (10, 10, 28, 28) 意即 10個標籤 維度都是 (10,28,28)  for i in range(label\_dim):  fill[i, i, :, :] = 1 # 舉例 標籤 5，第一個[]代表標籤5，第二個[]代表onehot為1的位置  print('Train D Label: ', fill.shape)  print('\n', fill[1].shape, '\n', fill[1]) # 假設我們要取得標籤 1 的 onehot (10,28,28) | | |  |  |   **開始訓練－起手式**  一樣從基本的參數開始宣告起，流程個別是：基本參數、數據載入、建立訓練相關的東西（模型、優化器、損失）、開始訓練。   |  | | --- | | """ 基本參數 """  epoch = 10  lr = 1e-5  batch = 4  device = torch.device('cuda:0' if torch.cuda.is\_available() else 'cpu')  z\_dim = 100 # latent Space  c\_dim = 1 # Image Channel  label\_dim = 10 # label      """ 取得數據集以及DataLoader """  transform = trans.Compose([  trans.ToTensor(),  trans.Normalize((0.5,),(0.5,)),  ])    train\_set = dset.MNIST(root='./mnist\_data/',  train=True, transform=transform,  download=True)    test\_set = dset.MNIST(root='./mnist\_data/',  train=False,  transform=transform,  download=False)    train\_loader = torch.utils.data.DataLoader(  dataset = train\_set,  batch\_size = batch,  shuffle=True,  drop\_last=True  )    test\_loader = torch.utils.data.DataLoader(  dataset = test\_set,  batch\_size = batch,  shuffle=False  )    """ 訓練相關 """    D = Discriminator(c\_dim, label\_dim).to(device)  G = Generator(z\_dim, label\_dim).to(device)  loss\_fn = nn.BCELoss()  D\_opt = optim.Adam(D.parameters(), lr= lr)  G\_opt = optim.Adam(G.parameters(), lr= lr)  D\_avg\_loss = []  G\_avg\_loss = []    img = []  ls\_time = [] |   **開始訓練 - 手動更新學習率**  會手動更新主要原因在於其實GAN的訓練並不是那麼的順利，如果速度太快會導致震盪嚴重訓練生成效果極差，所以GAN普遍的學習率都會更新並且都蠻低的，這邊我也稍微調整一下：   |  | | --- | | """ 看到很多範例都有手動調整學習率 """  if epoch == 8:  G\_opt.param\_groups[0]['lr'] /= 5  D\_opt.param\_groups[0]['lr'] /= 5 |   **開始訓練 - 訓練Ｄ、Ｇ**  一樣參考上一篇的DCGAN來改良，主要差別在於需要引入label，並且需要將label轉換成onehot格式，其中  鑑別器 (D) 的訓練步驟一樣先學真實圖片給予標籤1 再學生成圖片給予標籤 0，生成圖片的部分要產生對應的亂數label，丟入G的時候是從先前寫的 onehot 中提取對應的onehot格式標籤而丟入D的時候是從 fill 中提取～  生成器 (G) 的訓練方式就是把Ｄ的後半段拿出來用，但是標籤需要改成 1，因為它的目的是要騙過D！   |  | | --- | | """ 訓練 D """    D\_opt.zero\_grad()    x\_real = data.to(device)  y\_real = torch.ones(batch, ).to(device)  c\_real = fill[label].to(device)    y\_real\_predict = D(x\_real, c\_real).squeeze() # (-1, 1, 1, 1) -> (-1, )  d\_real\_loss = loss\_fn(y\_real\_predict, y\_real)  d\_real\_loss.backward()    noise = torch.randn(batch, z\_dim, 1, 1, device = device)  noise\_label = (torch.rand(batch, 1) \* label\_dim).type(torch.LongTensor).squeeze()  noise\_label\_onehot = onehot[noise\_label].to(device) #隨機產生label (-1, )    x\_fake = G(noise, noise\_label\_onehot) # 生成假圖  y\_fake = torch.zeros(batch, ).to(device) # 給予標籤 0  c\_fake = fill[noise\_label].to(device) # 轉換成對應的 10,28,28 的標籤    y\_fake\_predict = D(x\_fake, c\_fake).squeeze()  d\_fake\_loss = loss\_fn(y\_fake\_predict, y\_fake)  d\_fake\_loss.backward()  D\_opt.step()    """ 訓練 G """    G\_opt.zero\_grad()    noise = torch.randn(batch, z\_dim, 1, 1, device = device)  noise\_label = (torch.rand(batch, 1) \* label\_dim).type(torch.LongTensor).squeeze()  noise\_label\_onehot = onehot[noise\_label].to(device) #隨機產生label (-1, )    x\_fake = G(noise, noise\_label\_onehot)  #y\_fake = torch.ones(batch, ).to(device) #這邊的 y\_fake 跟上述的 y\_real 一樣，都是 1  c\_fake = fill[noise\_label].to(device)    y\_fake\_predict = D(x\_fake, c\_fake).squeeze()  g\_loss = loss\_fn(y\_fake\_predict, y\_real) #直接用 y\_real 更直觀  g\_loss.backward()  G\_opt.step()    D\_loss.append(d\_fake\_loss.item() + d\_real\_loss.item())  G\_loss.append(g\_loss.item()) |   **成果**  起初我在第五次迭代的時候調整了學習率結果原本 1 到 5 學習的都不錯，到第 6次的時候開始有了偏差，所以真的不能亂調學習率阿～   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 1 | 2 | 3 | 4 | 5 | |  |  |  |  |  | | 6 | 7 | 8 | 9 | 10 | |  |  |  |  |  |   下面是迭代15次的成果，感覺上比參考的gihub還要差了一些，仔細看了一下應該是D的結構跟learning rate的調整有差，大家可以在自己調整看看。    **訓練時間比較**  一樣都是 10 個 epoch ，Jetson Nano所需要的時間大約是 1 小時 40 分鐘，其實還算是蠻快的，大家可以試試看 CPU 去跑跑看就可以知道差異了。   |  |  | | --- | --- | |  |  |   **結語**  最後相信大家到看完這篇以及上一篇DCGAN已經對生成對抗網路有一定的熟悉度了，接下來我們可以找些GAN的github的範例來玩玩看並且增加應用～  **完整程式碼**   |  | | --- | | import torch  import torch.nn as nn  import torch.optim as optim  from torchvision.utils import make\_grid  import torchvision.datasets as dset  import torchvision.transforms as trans  import time  from torchsummary import summary  import hiddenlayer as hl  from torch.utils.tensorboard import SummaryWriter    class Generator(nn.Module):    def \_\_init\_\_(self, z\_dim, label\_dim):    super(Generator, self).\_\_init\_\_()    self.input\_x = nn.Sequential(  # input is Z, going into a convolution  nn.ConvTranspose2d(z\_dim, 256, 4, 1, 0, bias=False),  nn.BatchNorm2d(256),  nn.ReLU(True),  # image size = (1-1)\*1 - 2\*0 + 4 = 4  )    self.input\_y = nn.Sequential(  # input is Z, going into a convolution  nn.ConvTranspose2d( label\_dim, 256, 4, 1, 0, bias=False),  nn.BatchNorm2d(256),  nn.ReLU(True),  # image size = (1-1)\*1 - 2\*0 + 4 = 4  )    self.concat = nn.Sequential(    # 因為 x 跟 y 水平合併所以要再乘以 2  nn.ConvTranspose2d(256\*2, 128, 4, 2, 1, bias=False),  nn.BatchNorm2d(128),  nn.ReLU(True),  # image size = (4-1)\*2 - 2\*1 + 4 = 8    nn.ConvTranspose2d( 128, 64, 4, 2, 1, bias=False),  nn.BatchNorm2d(64),  nn.ReLU(True),  # image size = (8-1)\*2 - 2\*1 + 4 = 16    nn.ConvTranspose2d( 64, 1, 4, 2, 3, bias=False),  nn.Tanh()  # image size = (16-1)\*2 - 2\*3 + 4 = 28  )    def forward(self, x, y):    x = self.input\_x(x)  y = self.input\_y(y)  out = torch.cat([x, y] , dim=1)  out = self.concat(out)  return out  class Discriminator(nn.Module):    def \_\_init\_\_(self, c\_dim=1, label\_dim=10):    super(Discriminator, self).\_\_init\_\_()    self.input\_x = nn.Sequential(    # Input size = 1 ,28 , 28  nn.Conv2d(c\_dim, 64, (4,4), 2, 1),  nn.LeakyReLU(),  )  self.input\_y = nn.Sequential(    # Input size = 10 ,28 , 28  nn.Conv2d(label\_dim, 64, (4,4), 2, 1),  nn.LeakyReLU(),  )    self.concate = nn.Sequential(    # Input size = 64+64 ,14 , 14  nn.Conv2d(64\*2 , 64, (4,4), 2, 1),  nn.LeakyReLU(),    # Input size = (14-4+2)/2 +1 = 7  nn.Conv2d(64, 128, 3, 2, 1),  nn.LeakyReLU(),    # Input size = (7-3+2)/2 +1 = 4  nn.Conv2d(128, 1, 4, 2, 0),  nn.Sigmoid(),    # output size = (4-4)/2 +1 = 1  )    def forward(self, x, y):    x = self.input\_x(x)  y = self.input\_y(y)  out = torch.cat([x, y] , dim=1)  out = self.concate(out)  return out  label\_dim=10  z\_dim = 100  image\_size=28    """ 產生固定資料，每個類別10張圖(雜訊) 以及 對應的標籤，用於視覺化結果 """  temp\_noise = torch.randn(label\_dim, z\_dim) # (10, 100) 10張圖  fixed\_noise = temp\_noise  fixed\_c = torch.zeros(label\_dim, 1) # (10, 1 ) 10個標籤    for i in range(9):  fixed\_noise = torch.cat([fixed\_noise, temp\_noise], 0) #將每個類別的十張雜訊依序合併，維度1會自動boardcast  temp = torch.ones(label\_dim, 1) + i #依序將標籤對應上 0~9  fixed\_c = torch.cat([fixed\_c, temp], 0) #將標籤也依序合併    fixed\_noise = fixed\_noise.view(-1, z\_dim, 1, 1) #由於是捲積所以我們要將形狀轉換成二維的  print('Predict Noise: ', fixed\_noise.shape)  print('Predict Label (before): ', fixed\_c.shape, '\t\t\t', fixed\_c[50])    """ 針對 lael 做 onehot """  fixed\_label = torch.zeros(100, label\_dim) #先產生 [100,10] 的全0張量，100個標籤，每個標籤維度是 10  fixed\_label.scatter\_(1, fixed\_c.type(torch.LongTensor), 1) #轉成 onehot編碼 (1, ) -> (10, )  fixed\_label = fixed\_label.view(-1, label\_dim, 1, 1) #轉換形狀 (10, 1, 1 )  print('Predict Label (onehot): ',fixed\_label.shape, '\t\t', fixed\_label[50].view(1,-1), '\n')    """ 幫標籤做前處理，onehot for g, fill for d """  onehot = torch.zeros(label\_dim, label\_dim) # 產生 (10,10) 10個標籤，維度為10 (onehot)  onehot = onehot.scatter\_(1, torch.LongTensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]).view(label\_dim, 1), 1).view(label\_dim, label\_dim, 1, 1)  print('Train G label:',onehot[1].shape, '\n', onehot[1], '\n') # 假設我們要取得標籤 1 的 onehot (10,1,1)，直接輸入索引 1    fill = torch.zeros([label\_dim, label\_dim, image\_size, image\_size]) # 產生 (10, 10, 28, 28) 意即 10個標籤 維度都是 (10,28,28)  for i in range(label\_dim):  fill[i, i, :, :] = 1 # 舉例 標籤 5，第一個[]代表標籤5，第二個[]代表onehot為1的位置  print('Train D Label: ', fill.shape)  print('\n', fill[1].shape, '\n', fill[1]) # 假設我們要取得標籤 1 的 onehot (10,28,28)  """ 基本參數 """  epoch = 10  lr = 1e-5  batch = 4  device = torch.device('cuda:0' if torch.cuda.is\_available() else 'cpu')  z\_dim = 100 # latent Space  c\_dim = 1 # Image Channel  label\_dim = 10 # label      """ 取得數據集以及DataLoader """  transform = trans.Compose([  trans.ToTensor(),  trans.Normalize((0.5,),(0.5,)),  ])    train\_set = dset.MNIST(root='./mnist\_data/',  train=True, transform=transform,  download=True)    test\_set = dset.MNIST(root='./mnist\_data/',  train=False,  transform=transform,  download=False)    train\_loader = torch.utils.data.DataLoader(  dataset = train\_set,  batch\_size = batch,  shuffle=True,  drop\_last=True  )    test\_loader = torch.utils.data.DataLoader(  dataset = test\_set,  batch\_size = batch,  shuffle=False  )    """ 訓練相關 """    D = Discriminator(c\_dim, label\_dim).to(device)  G = Generator(z\_dim, label\_dim).to(device)  loss\_fn = nn.BCELoss()  D\_opt = optim.Adam(D.parameters(), lr= lr)  G\_opt = optim.Adam(G.parameters(), lr= lr)  D\_avg\_loss = []  G\_avg\_loss = []    img = []  ls\_time = []    for i in range(epoch):    start\_time = time.time()  D\_loss = []  G\_loss = []    """ 看到很多範例都有手動調整學習率 """  if epoch == 2 :  G\_opt.param\_groups[0]['lr'] /= 2  D\_opt.param\_groups[0]['lr'] /= 2  elif epoch == 5:  G\_opt.param\_groups[0]['lr'] /= 5  D\_opt.param\_groups[0]['lr'] /= 5  elif epoch == 15:  G\_opt.param\_groups[0]['lr'] /= 10  D\_opt.param\_groups[0]['lr'] /= 10      for idx, (data, label) in enumerate(train\_loader):    """ 訓練 D """    D\_opt.zero\_grad()    x\_real = data.to(device)  y\_real = torch.ones(batch, ).to(device)  c\_real = fill[label].to(device)    y\_real\_predict = D(x\_real, c\_real).squeeze() # (-1, 1, 1, 1) -> (-1, )  d\_real\_loss = loss\_fn(y\_real\_predict, y\_real)  d\_real\_loss.backward()    noise = torch.randn(batch, z\_dim, 1, 1, device = device)  noise\_label = (torch.rand(batch, 1) \* label\_dim).type(torch.LongTensor).squeeze()  noise\_label\_onehot = onehot[noise\_label].to(device) #隨機產生label (-1, )    x\_fake = G(noise, noise\_label\_onehot) # 生成假圖  y\_fake = torch.zeros(batch, ).to(device) # 給予標籤 0  c\_fake = fill[noise\_label].to(device) # 轉換成對應的 10,28,28 的標籤    y\_fake\_predict = D(x\_fake, c\_fake).squeeze()  d\_fake\_loss = loss\_fn(y\_fake\_predict, y\_fake)  d\_fake\_loss.backward()  D\_opt.step()    """ 訓練 G """    G\_opt.zero\_grad()    noise = torch.randn(batch, z\_dim, 1, 1, device = device)  noise\_label = (torch.rand(batch, 1) \* label\_dim).type(torch.LongTensor).squeeze()  noise\_label\_onehot = onehot[noise\_label].to(device) #隨機產生label (-1, )    x\_fake = G(noise, noise\_label\_onehot)  #y\_fake = torch.ones(batch, ).to(device) #這邊的 y\_fake 跟上述的 y\_real 一樣，都是 1  c\_fake = fill[noise\_label].to(device)    y\_fake\_predict = D(x\_fake, c\_fake).squeeze()  g\_loss = loss\_fn(y\_fake\_predict, y\_real) #直接用 y\_real 更直觀  g\_loss.backward()  G\_opt.step()    D\_loss.append(d\_fake\_loss.item() + d\_real\_loss.item())  G\_loss.append(g\_loss.item())    if idx%(int(len(train\_loader)/10))==0:  with torch.no\_grad():  print("Epoch [{}/{}] \t Step[{}/{}] \t D\_LOSS:{:.6} \t G\_LOSS:{:.6}".format(i+1, epoch, idx+1, len(train\_loader) , D\_loss[idx], G\_loss[idx]))    res = G(fixed\_noise, fixed\_label)  img.append(make\_grid(res, padding=0, normalize=True))    D\_avg\_loss.append(torch.mean(torch.FloatTensor(D\_loss)))  G\_avg\_loss.append(torch.mean(torch.FloatTensor(G\_loss)))    end\_time = time.time()  cost\_time = end\_time - start\_time  ls\_time.append(cost\_time)  print('Each Epoch Cost :{} s '.format(cost\_time))    print('Total Cost Time : {}'.format(sum(ls\_time))) | | | | |
| **相關文章** | | | |
| Conditional DCGAN for MNIST  <https://github.com/znxlwm/pytorch-MNIST-CelebA-cGAN-cDCGAN>  pytorch-MNIST-CelebA-cGAN-cDCGAN  <https://github.com/sarahwolf32/conditional-DCGAN-for-MNIST>  Understand torch.scatter\_() <https://medium.com/@yang6367/understand-torch-scatter-b0fd6275331c>  PyTorch笔记之 scatter() 函数  <https://www.cnblogs.com/dogecheng/p/11938009.html> | | | |