DLCV HW3 Report

姓名：陳泓均
Collaborators：潘彥銘、林奕廷、詹書愷、賴繹文

Problem 1. GAN

1. Model architecture and implementation details:

Model 就如同 GAN 原始 paper 裏頭所寫，分為 generator 和 discriminator 兩部分：

a. Generator：

```
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        def batch1d(out_feat):
            layer = nn.BatchNorm1d(out_feat, 0.8)
            return layer
        def batch2d(out_feat):
            layer = nn.BatchNorm2d(out_feat)
            return layer
        self.hidden1 = nn.Linear(100, 128 * 8 * 8)
        self.batch1 = batch1d(128*8*8)
        self.conv1 = nn.ConvTranspose2d(128, 128, kernel_size = 4, stride = 2, padding = 1)
        self.batch2 = batch2d(128)
        self.conv2 = nn.ConvTranspose2d(128, 64, kernel_size = 4, stride = 2, padding = 1)
        self.batch3 = batch2d(64)
        self.conv3 = nn.ConvTranspose2d(64, 3, kernel_size = 4, stride = 2, padding = 1)
        self.batch4 = batch2d(3)
        self.tanh = nn.Tanh()

    def forward(self, x):
        x = F.relu(self.batch1(self.hidden1(x)))
        x = x.view(-1,128,8,8)
        x = F.relu(self.batch2(self.conv1(x)))
        x = F.relu(self.batch3(self.conv2(x)))
        x = self.tanh(self.batch4(self.conv3(x)))
        return x
```

基本上就是三層的 convolution，然後中間有加上 batchnorm 來穩定 training。Kernel_size 在經過嘗試之後發現 4 是最好的，並且剛好每次 convolution 都會讓 image 的 size 減半。其實有點像是 DCGAN 的架構，feature map 的數量都是不斷除以 2。

然後因為我的 loss 是傳統的 GAN loss，所以最後要加上 tanh 層。(如果是 WGAN 就不用加)

b. Discriminator：

```python
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size = 4, stride = 2, padding = 1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size = 4, stride = 2, padding = 1)
        self.conv3 = nn.Conv2d(64, 128, kernel_size = 4, stride = 2, padding = 1)
        self.conv4 = nn.Conv2d(128, 256, kernel_size = 4, stride = 2, padding = 1)
        self.hidden1 =  nn.Linear(4 * 4 * 256, 1)
        self.sig = nn.Sigmoid()

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = F.relu(self.conv3(x))
        x = F.relu(self.conv4(x))
        x = x.view(x.shape[0], -1)
        x = self.hidden1(x)
        x = self.sig(x)
        return x
```

如上圖所示，基本上就是和 Generator 反過來，但比起 Generator 多了一層，因此也沒有完全對稱，feature map 的數量也是不斷乘以 2。
最後要加上 Sigmoid，因為最後 output 之數值必須介於 0-1 之間。

2. Fig1_2.jpg:

3. Discussion:

GAN 不太穩定，不容易 train，因此 model 架構、discriminator 還有 generator 的訓練次數等等都很重要。我用的 loss function 是一般的 GAN(WAGN 沒有 train 起來)，所以 Generator 需要訓練比較多次，在經過多次嘗試之後，我發現 generator 訓練兩次效果是最好的，這樣 discriminator 和 generator 可以互相抗衡，否則如果一方太過強大，很容易 loss function 直接炸掉，訓練也會失敗。另外 model 架構的部分，加上 batchnorm 在 GAN 裡面就顯得很重要，可以穩定 CNN 的 training。

由於 CelebA 的 dataset 算是比較沒那麼 noisy，並且是有經過 alignment，所以在 training data 的 preprocessing 我就沒有先用 facial recognition 的 code 來篩選訓練資料。(在過去的經驗裡，篩選掉比較 noisy 的人臉會對訓練有很大的幫助)

Problem 2. ACGAN

1. Model architecture and implementation details:

a. Generator: 稍加修改了原本(上一題)的架構，把 feature map 的數量加多了，詳細原因會在下方的討論說明。然後 input 的 dimension 加上了 class label 的數量，這裡只有 smiling 與否所以是 1。

```
class Generator(nn.Module):
    def __init__(self, num_classes):
        super(Generator, self).__init__()
        def batch1d(out_feat):
            layer = nn.BatchNorm1d(out_feat, 0.8)
            return layer
        def batch2d(out_feat):
            layer = nn.BatchNorm2d(out_feat)
            return layer
        self.hidden1 = nn.Linear(100 + num_classes, 512 * 4 * 4)
        self.conv1 = nn.ConvTranspose2d(512, 256, kernel_size = 4, stride = 2, padding = 1)
        self.batch1 = batch2d(256)
        self.conv2 = nn.ConvTranspose2d(256, 128, kernel_size = 4, stride = 2, padding = 1)
        self.batch2 = batch2d(128)
        self.conv3 = nn.ConvTranspose2d(128, 64, kernel_size = 4, stride = 2, padding = 1)
        self.batch3 = batch2d(64)
        self.conv4 = nn.ConvTranspose2d(64, 3, kernel_size = 4, stride = 2, padding = 1)
        self.tanh = nn.Tanh()

    def forward(self, x):
        x = F.relu(self.hidden1(x))
        x = x.view(-1,512,4,4)
        x = F.relu(self.batch1(self.conv1(x)))
        x = F.relu(self.batch2(self.conv2(x)))
        x = F.relu(self.batch3(self.conv3(x)))
        x = self.tanh(self.conv4(x))
        return x
```

b. Discriminator：也是小做修改，並且使得 Discriminator 和 Generator 的架構對稱。另外，因為是 ACGAN，因此加上了 auxiliary classifier，同樣寫在 discriminator 的 module 裡面。

```python
class Discriminator(nn.Module):
    def __init__(self, num_classes):
        super(Discriminator, self).__init__()
        self.conv1 = nn.Conv2d(3, 64, kernel_size = 4, stride = 2, padding = 1)
        self.conv2 = nn.Conv2d(64, 128, kernel_size = 4, stride = 2, padding = 1)
        self.conv3 = nn.Conv2d(128, 256, kernel_size = 4, stride = 2, padding = 1)
        self.conv4 = nn.Conv2d(256, 512, kernel_size = 4, stride = 2, padding = 1)
        self.fc_dis =  nn.Linear(4 * 4 * 512, 1)
        self.fc_class = nn.Linear(4*4*512, num_classes)
        # self.sofxmax = nn.Softmax(1)
        self.sig = nn.Sigmoid()

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = F.relu(self.conv3(x))
        x = F.relu(self.conv4(x))
        x = x.view(x.shape[0], -1)
        dis = self.sig(self.fc_dis(x))
        cl = self.sig(self.fc_class(x))
        return dis, cl
```

2. Generated image: [fig2_2.jpg]

左邊為 not_smiling，右邊為 smiling，可以看到右邊明顯比起左邊有微笑。

3. Discussion:

最初直接把前一小題的 GAN model 直接拿來用，以為加上 ACGAN 的 auxiliary classifier 就可以成功，但後來發現他會直接忽略加入的 condition。我一開始有試著調整一些參數，但效用都不大。最後才發現可能是 model 架構的問題，於是我調整到跟 DCGAN 更相像的架構(忘記 DCGAN 的細節，所以不確定有沒有一樣)，基本上就是多加了 feature map 的數量，因為我認為有可能是 feature map 的數量過少，可能 condition 的資訊在過程中丟失。因此，結論就是如果要訓練

ACGAN，必須確保 feature map 的 channel 足夠多，以便保存 condition 的資訊。

Problem 3.　DANN

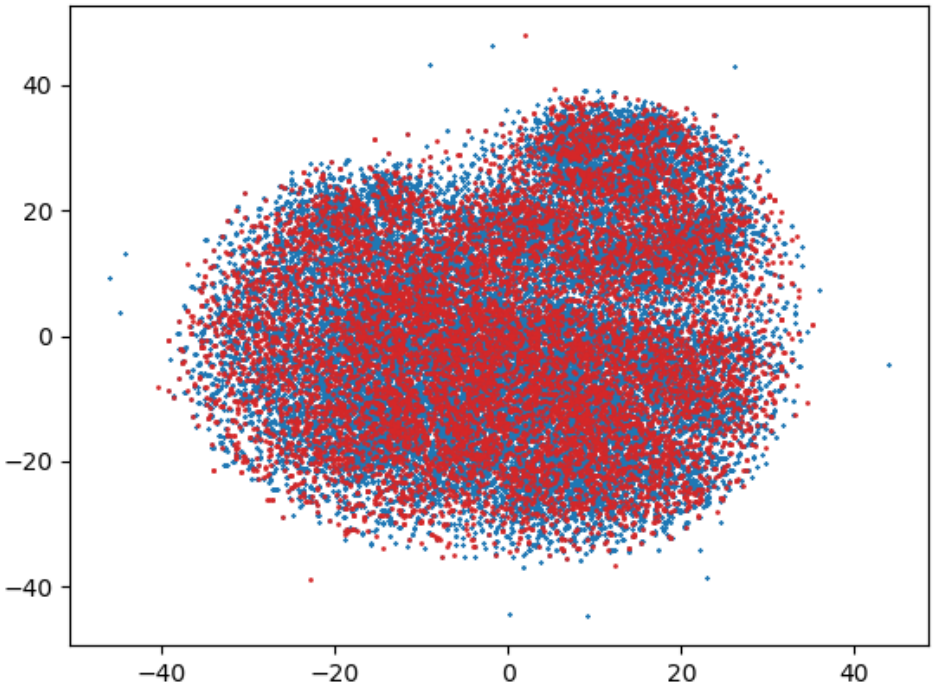
1.2.3.用表格來說明：

	Svhn -> mnistm	Mnistm -> svhn
Train on source	acc = 0.44399]	acc = 0.28180700676090964
Adaptation	acc = 0.49019	acc = 0.4873232944068838
Train on target	acc = 0.97439]	acc = 0.8970881991395205

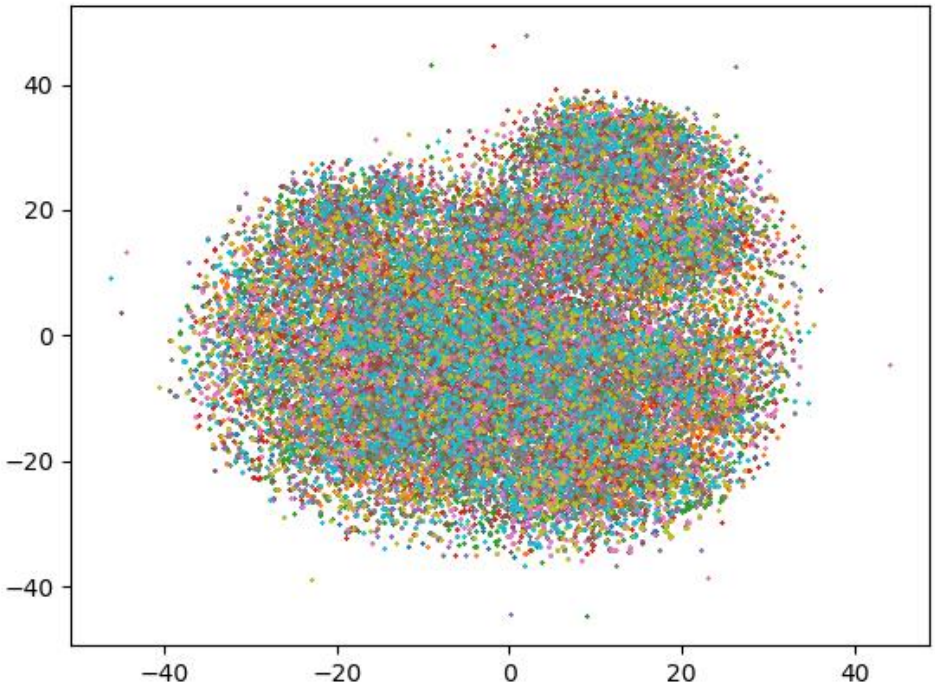
可以發現實驗的結果，都算是符合預期，Adaptation 的結果有比 only train on source 好一點。不過值得注意的是，似乎在 mnistm-> svhn 的這個 setting 上，adaptation 進步的幅度比較大，且由 train on source 和 target 上來看的話，svhn 可能是比 mnistm 難 train 一些，但 mnistm-> svhn 的 adaptation 結果並沒有和反向的 setting 差太多。

4. T-SNE visualization:

a. Svhn -> mnistm

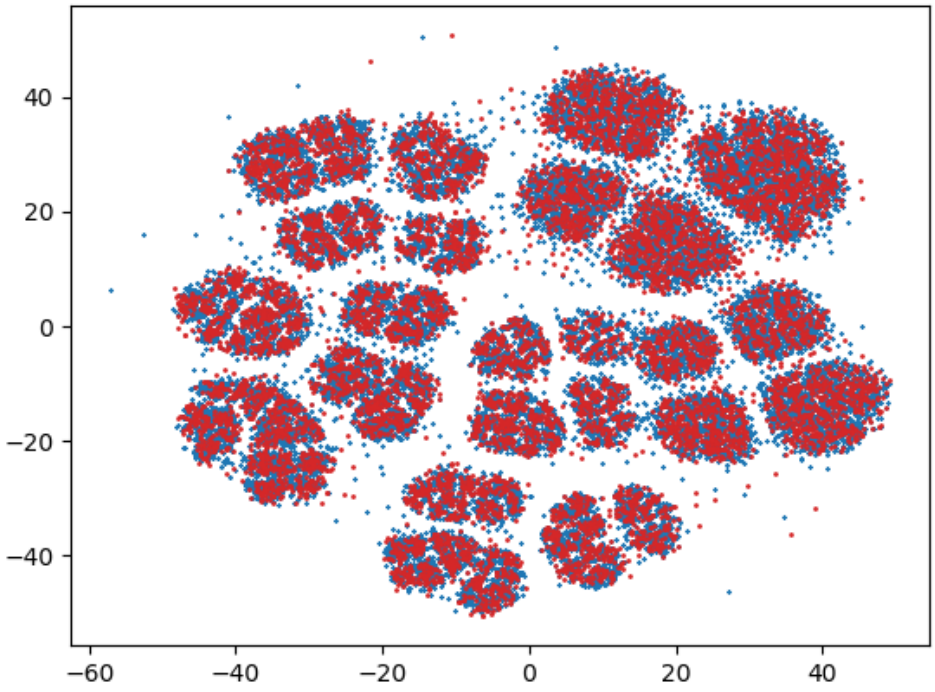


Different domains

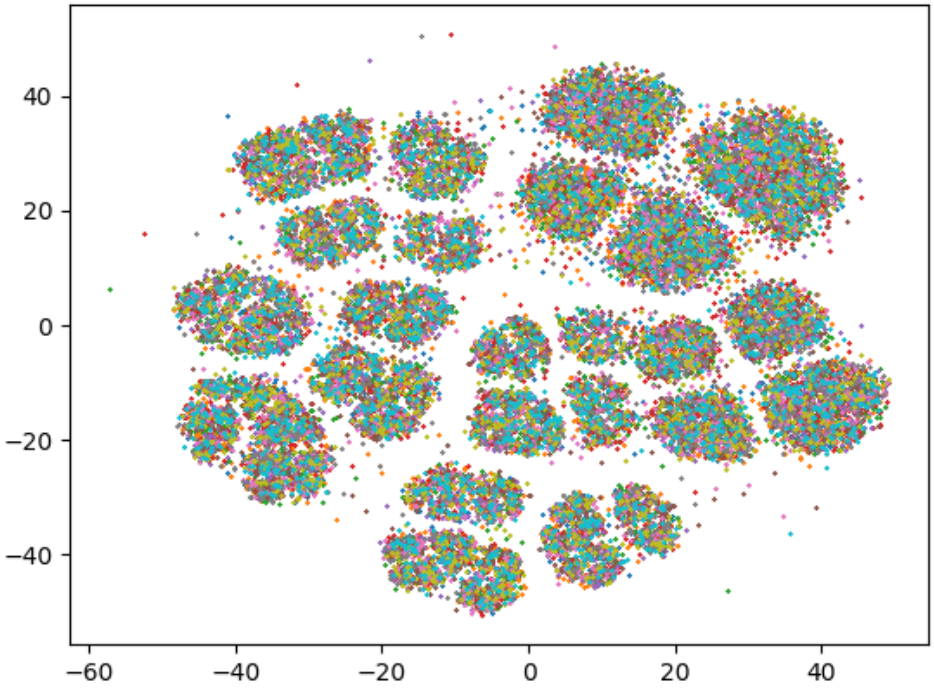


Different digits

b. Mnistm -> svhn



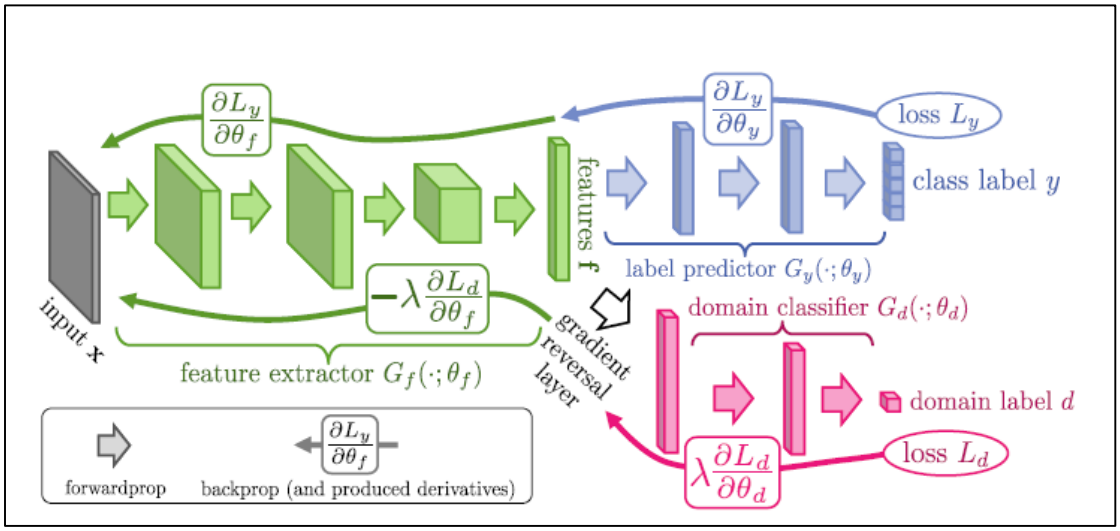
Different domains



Different digits

由以上圖形可以發現，其實 DANN 的 latent space，已經成功把不同 domain 的資料混淆了，在圖上兩個 domain 的 feature 投影到二維後幾乎是重疊的；但是，在分辨 label 上面，DANN 其實並沒有做得非常好，在圖上其實沒有很明顯地按照 label 散開。

5. Architecture and implementation detail of the model:



基本上架構按照 DANN 的 paper 上，也就是上圖所示，分為 feature extractor, domain label predictor，以及 class label predictor 三個部分，然後其中 feature extractor 接到 domain label predictor 之中間有一 gradient reversal layer。我的架構如下：

a. Feature extractor:

```
class f_ext(nn.Module):
    def __init__(self):
        super(f_ext, self).__init__()
        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(64, 64, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(64, 64, kernel_size=3, padding=1)
        self.batch = nn.BatchNorm2d(64)
        self.maxpool = nn.MaxPool2d(2)
        self.dropout = nn.Dropout2d(0.4)

    def forward(self, x):
        x = self.maxpool(F.relu(self.batch(self.conv1(x))))
        x = self.maxpool(F.relu(self.batch(self.dropout((self.conv2(x))))))
        x = self.maxpool(F.relu(self.batch(self.dropout((self.conv3(x))))))

        return x
```

如上圖所示，基本上由三層 convolution layer 組成，其中加了 batchnorm 為了穩定 CNN 的 training，還有 dropout 層為了防止 overfitting。基本上由於 task 不是太困難，network 架構不是太大。

b. Domain label predictor

```
class domain_cl(nn.Module):
    def __init__(self):
        super(domain_cl, self).__init__()
        self.hidden1 =  nn.Linear(64*3*3, 256)
        self.hidden2 = nn.Linear(256, 256)
        self.hidden3 = nn.Linear(256, 1)
        # self.batch = nn.BatchNorm1d(100)
        self.sig = nn.Sigmoid()
    def forward(self, x):
        x = F.relu(self.hidden1(x))
        x = F.relu(self.hidden2(x))
        x = self.sig(self.hidden3(x))
        return x
```

如上圖所示，基本上由簡單的 fully connected layer 組成，最後 predict 的是一個數值(0 代表 domain svhn，1 代表 domain mnist)，然後再用 logistic regression 的方式去做 training。加上 sigmoid 則是為了讓其數值在 0-1 之間。

c. Class label predictor

```
class label_cl(nn.Module):
    def __init__(self):
        super(label_cl, self).__init__()
        self.hidden1 =  nn.Linear(64 * 3 * 3, 256)
        self.hidden2 = nn.Linear(256, 256)
        self.hidden3 = nn.Linear(256, 10)
        # self.batch = nn.BatchNorm1d(100)
        # self.dropout = nn.Dropout2d()

    def forward(self, x):
        x = F.relu(self.hidden1(x))
        x = F.relu(self.hidden2(x))
        x = self.hidden3(x)
        return x
```

一樣是三層 fully connected layer，我盡量將其設計的和 domain predictor 相像，唯一不同只在於 output_dim 是 10。最後一層沒有加上 softmax 是因為我直接使用 nn.CrossEntropyLoss，而其已經將 softmax 與 NLLLoss 合併。

結論是其實這個 task 並不算太大，也因此 network 的設計也不需要太複雜，利用 CNN 抽出 feature 之後，後面的 prediction network 其實用幾層簡單的 fully connected layer 就可以了。

6. **Discussion:**

   Train 了 DANN 之後發現，根據 tSNE 的結果，的確 domain signal 在 feature space 裡面已經幾乎消失，但其實 t-SNE 的結果顯示似乎不同 class label 的 feature 並沒有分開的很明顯，且最後 adaptation 的結果其實也只比 train 在 source 好一點點而已。可能因為方法比較 naïve，所以進步的幅度並不大，也因此可能還需要更好的 adaptation 的方法來改進。

Problem 4.　Improved UDA -> ADDA

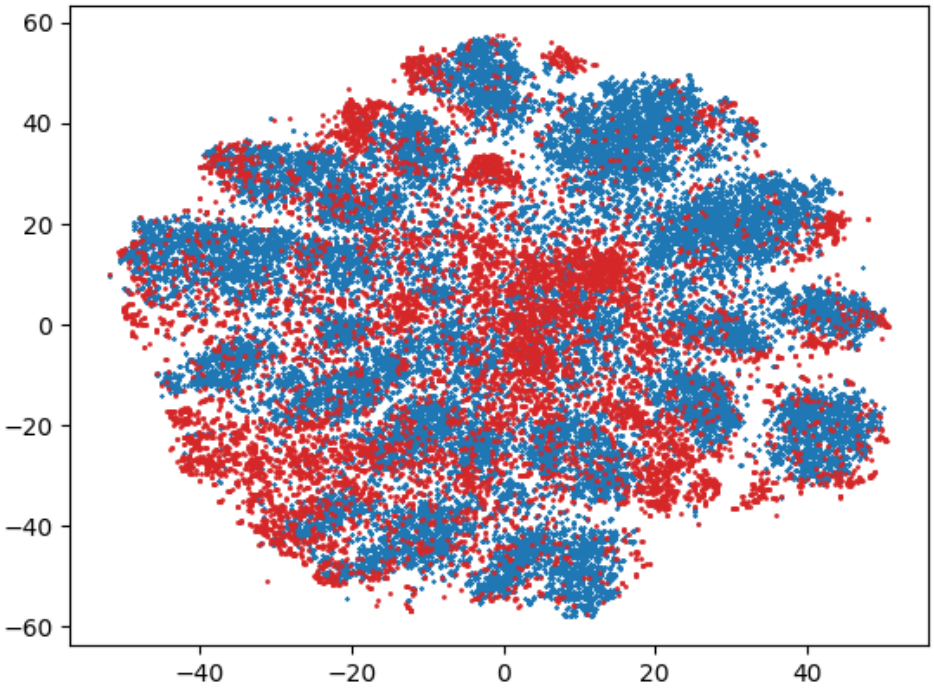
我所使用的 UDA 的架構為 ADDA，詳細架構與訓練方式等等在下方第三點說明。

1. 用表格來表示：

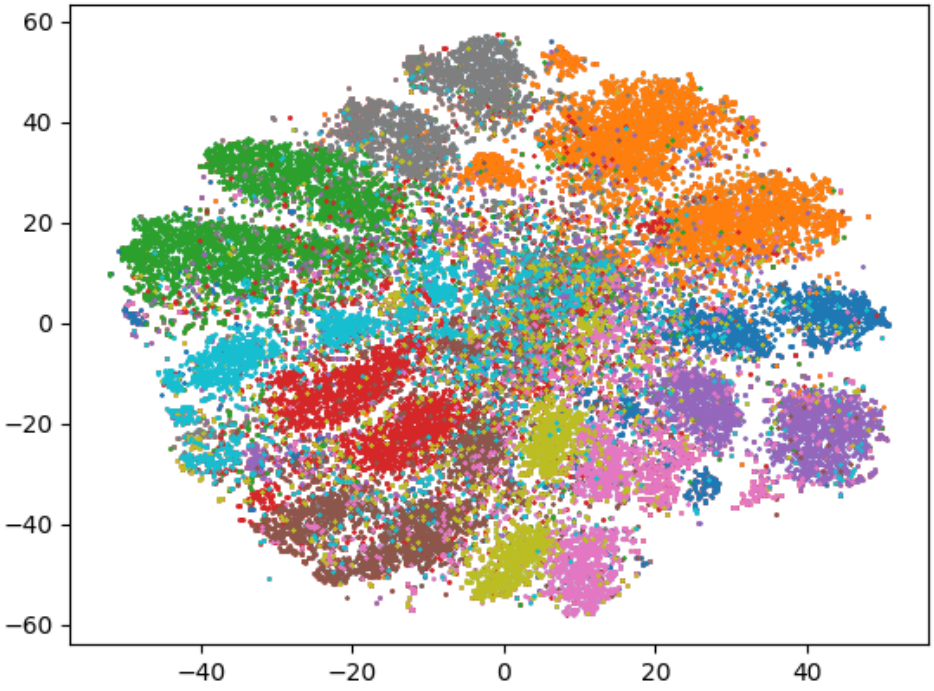
	Svhn -> mnistm	Mnistm -> svhn
Adaptation	acc = 0.50149]	acc = 0.590119852489244

2. **tSNE visualization:**

(1) svhn -> mnistm

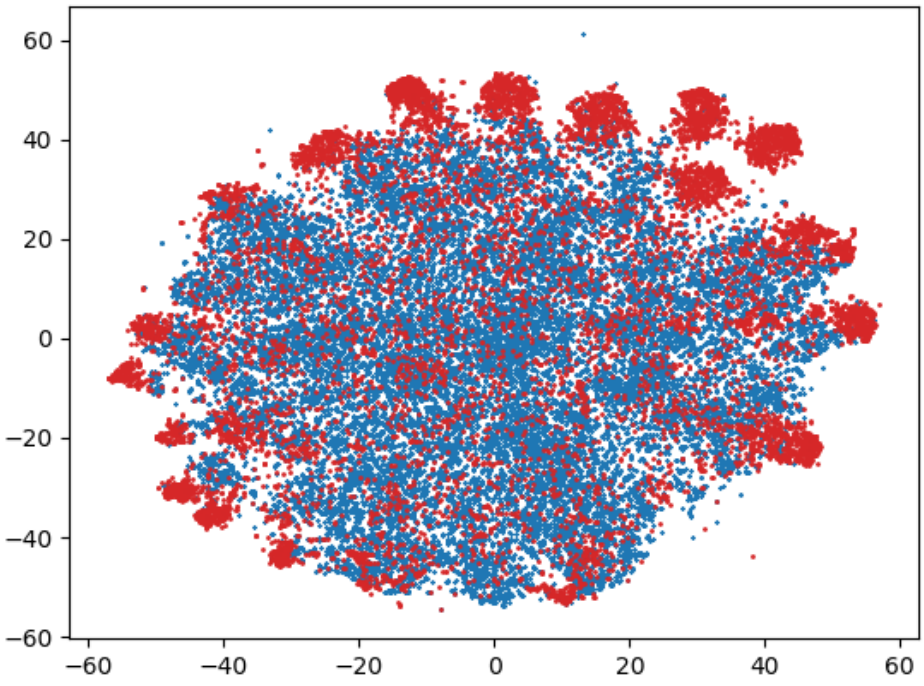


Different domains

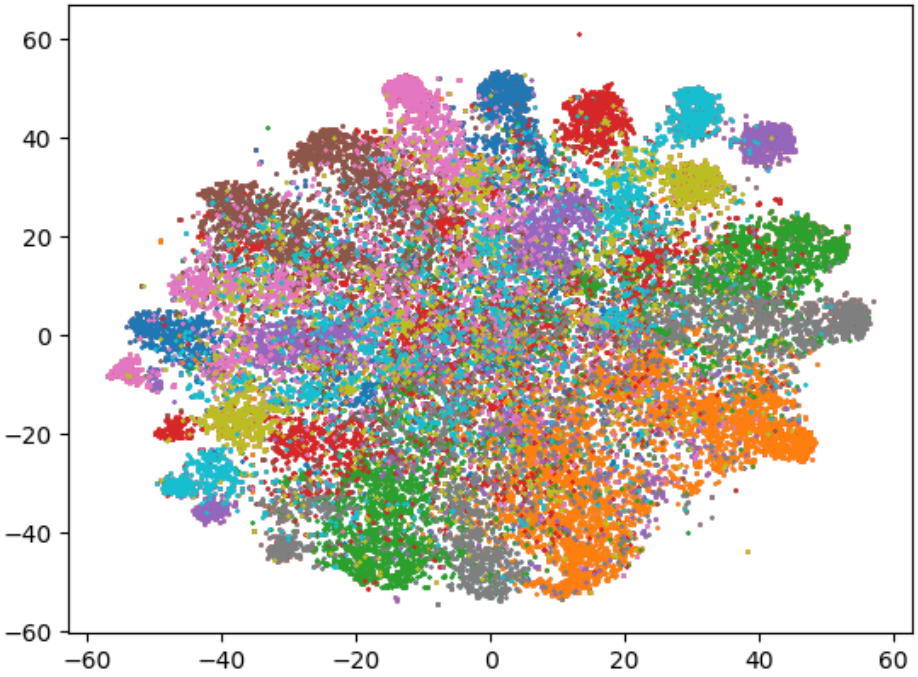


Different digits

(2) mnistm -> svhn



Different domains



Different digits

基本上不論是哪個方向的 adaptation，不同 domain 的 feature space 都沒有相差太多，還算是大致上有達到預期的效果，讓不同 domain 的 feature space 重合。而不同 digits 的部分，有分的比較開，跟 DANN 的結果比起來還好滿多的，尤其是 svhn -> mnistm 的方向，其實還分得滿開的，不知道為甚麼結果做起來反而準確率比 mnistm -> svhn 的差。

3. Architecture and implementation detail:

基本上架構分成幾個部分：

a. Source and target CNN:

這個 work 裡面她把 source 和 target domain 的 CNN 分開，兩者共用架構但不共用參數，可以把他想成 encoder，各自抽取 source 和 target domain 的 feature。

```
class source_cnn(nn.Module):
    def __init__(self):
        super(source_cnn, self).__init__()
        self.conv1 = nn.Conv2d(1, 20, kernel_size=(5, 5), stride=(1, 1))
        self.conv2 = nn.Conv2d(20, 50, kernel_size=(5, 5), stride=(1, 1))
        self.maxpool = nn.MaxPool2d (2)
        self.dropout = nn.Dropout2d(0.5)
        self.linear = nn.Linear(800, 500)

    def forward(self, x):
        x = F.relu(self.maxpool(self.conv1(x)))
        x = F.relu(self.maxpool(self.dropout((self.conv2(x)))))
        x = x.view(x.shape[0], -1)
        x = self.linear(x)

        return x
```

架構細節如上圖，target_cnn 和 source 一樣因此就不再放。用的是 LeNet 的架構，因為 task 不算大，因此也用比較少層且 feature map 數量較少的 CNN。

b. Domain discriminator:

Domain discriminator 的作用是根據 feature 去 predict 他是來自哪一個 domain 的。會給出一個介於 0-1 之間的分數，高代表可能來自 source，低則代表可能來自 target。

```python
class domain_cl(nn.Module):
    def __init__(self):
        super(domain_cl, self).__init__()
        self.hidden1 = nn.Linear(500,500)
        self.hidden2 = nn.Linear(500,500)
        self.hidden3 = nn.Linear(500,1)
        # self.hidden1 =  nn.Linear(64*3*3, 256)
        # self.hidden2 = nn.Linear(256, 256)
        # self.hidden3 = nn.Linear(256, 1)
        # self.batch = nn.BatchNorm1d(100)
        self.sig = nn.Sigmoid()
    def forward(self, x):
        x = F.relu(self.hidden1(x))
        x = F.relu(self.hidden2(x))
        x = self.sig(self.hidden3(x))
        return x
```

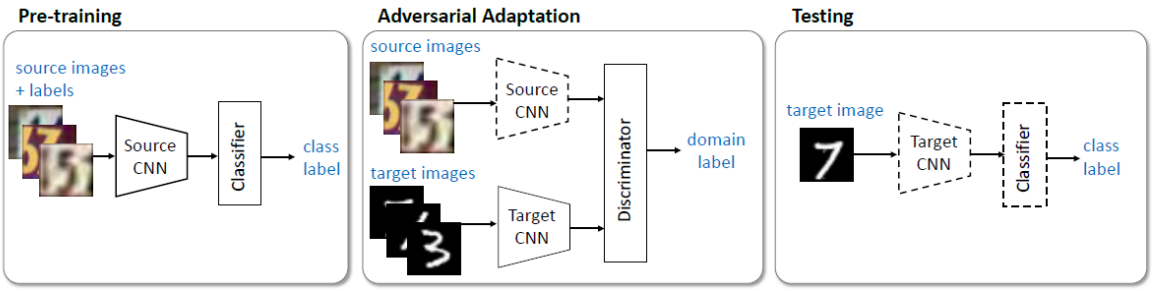
架構如上圖，基本上也是幾層的 fully-connected layer。

c. Label Classifier:

架構如下圖所示。

```python
class label_cl(nn.Module):
    def __init__(self):
        super(label_cl, self).__init__()
        # self.hidden1 =  nn.Linear(64 * 3 * 3, 256)
        # self.hidden2 = nn.Linear(256, 256)
        self.hidden = nn.Linear(500, 10)
        # self.batch = nn.BatchNorm1d(100)
        self.dropout = nn.Dropout2d()

    def forward(self, x):
        # x = F.relu(self.hidden1(x))
        # x = F.relu(self.hidden2(x))
        x = self.dropout(F.relu(x))
        x = self.hidden(x)
        return x
```



介紹完幾個 module，接下來說明訓練方式。訓練分為兩個階段。

一、 先 pre-train source_cnn 以及 classifier 在 source domain data 上。

二、 Fix 住 source_cnn，訓練 target CNN 以及 domain discriminator。這步驟的目的是為了讓 source 和 target domain 有相像的 feature space，因此使用了 adversarial training 的方法。

Testing 的時候則直接拿 target_CNN 和 classifier 去做 digit classification。

ADDA 主要的精神應該是讓 target 和 source domain 的 latent space 有相像的 distribution，但是因為是利用 adversarial training 的方式，因此不太好訓練。關於訓練的細節，底下的討論會詳細說明。

4. Discussion:

首先在訓練過程方面，由於使用的是類似 GAN 的 adversarial training 的架構，因此對於各種參數相當敏感，非常難調整。首先有幾個重要的訣竅：

a. 在第一步 pre-train 完 source_CNN 之後，第二步 adversarial training 的時候要用 source_CNN 的參數去 initialize target_CNN，這麼做是為了讓 source 和 target 的 feature space 不要相差太大，如果兩者完全沒有重疊，那麼 adversarial training 會完全沒有效用(和 GAN 原理相似)。
b. 在開始 adversarial training 之前，先 update discriminator 幾個 step，原因是 target_cnn 因為有經過 initialize，已經某種程度上可以產生出和 source，也就是真的 feature 有相像的分布，因此必須先讓 discriminator 多學一下，趕上 generator(target_cnn)的水準。至於要 update 幾個 step，根據 learning rate 以及 domain 不同也有差別，在這裡不贅述，總之非常難調。
c. 再來是 discriminator 和 generator 的強度平衡，在這個 task 裡面 discriminator 必須比 generator 稍強一點，我一開始沒發現這點所以一直往錯的方向調，因此 d update 的次數必須比 g 多，在 mnistm->svhn 是兩次最好，svhn->mnistm 則是 4 次。
d. 最後，不同方向 training 的參數竟然很不一樣，其中我認為原因是因為 svhn 應該是比 mnistm 難訓練的 dataset，從第三題的第一小題就可看出，也因為這樣訓練的方式也會有所不同。Svhn->mnistm 的參數比較難調，非常難調，我認為可能是因為 svhn 訓練完過後，已經某種程度上有能力去做 mnistm 的 task，因此如果參數調不好，反而結果會更差；最後，經過大約兩天的嘗試，我最後只能調出比 DANN 好一點點的結果。而 mn->sv 的，其實沒怎麼調參數就成功了，且 learning rate 也不需要調得很小。

Reference：

1. My MLDS github repository
   https://github.com/jackchen03/MLDS_2019_Spring/tree/master/hw3/hw3-1/hw3_1
2. DANN pytorch implementation

https://github.com/CuthbertCai/pytorch_DANN

https://github.com/fungtion/DANN

3. ADDA pytorch implementation

https://github.com/corenel/pytorch-adda