Deep Learning for Computer Vision HW#3

B05901182 電機四 潘彥銘

Problem 1: (20%)

Collaboration: None

Reference:

https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html

1. (5%)

The structure of the generator:

```
G:  Generator(
  (layer1): Sequential(
    (0): ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace)
  )
  (layer2): Sequential(
    (0): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace)
  )
  (layer3): Sequential(
    (0): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace)
  )
  (layer4): Sequential(
    (0): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace)
  )
  (layer5): Sequential(
    (0): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): Tanh()
  )
)
```

The structure of the discriminator:

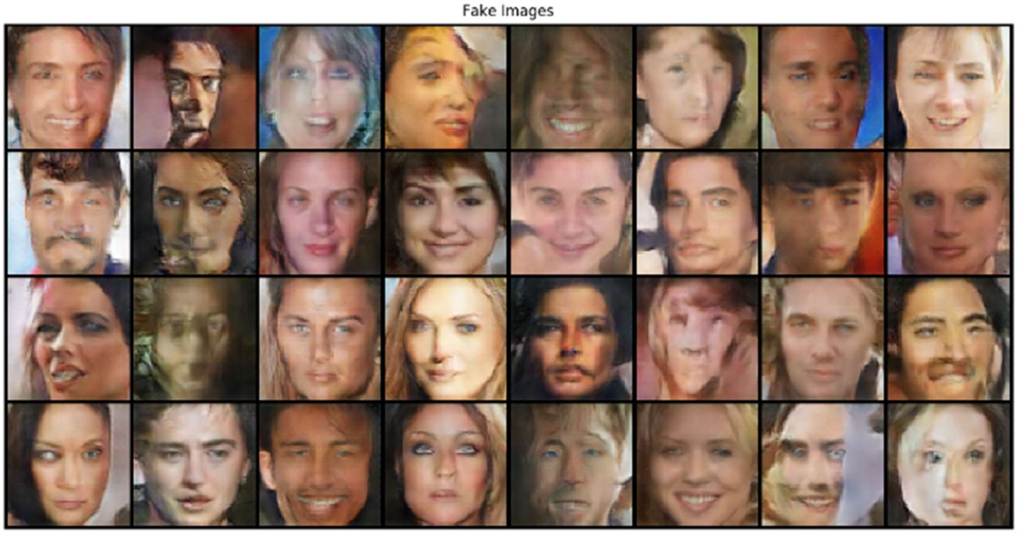
```
D:  Discriminator(
  (layer1): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): LeakyReLU(negative_slope=0.2, inplace)
  )
  (layer2): Sequential(
    (0): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): LeakyReLU(negative_slope=0.2, inplace)
  )
  (layer3): Sequential(
    (0): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): LeakyReLU(negative_slope=0.2, inplace)
  )
  (layer4): Sequential(
    (0): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): LeakyReLU(negative_slope=0.2, inplace)
  )
  (layer5): Sequential(
    (0): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): Sigmoid()
  )
)
```

The other hyper-parameters of the my model:

```
n_workers = 4
batch_size = 128
image_size = 64
num_channels = 3
z_size = 100  # Size of z latent vector (i.e. size of generator input)
gf_size = 64  # Size of feature maps in generator
df_size = 64  # Size of feature maps in discriminator
num_epochs = 50
lr = 0.0002
beta1 = 0.5
beta2 = 0.999 # (beta1, beta2) are the hyper-parameters of Adam optim
num_gpus = 1
```

2. (10%)

Fake Images



3. (5%)

While I implemented my GAN model, I was confused that why I should add .detach() while training discriminator(like the figure below).

```
output = netD(fake.detach()).view(-1)
```

After looking up it on google, I found that this is really important. While training discriminator, we don't want to update generator. So the .detach() is necessary. If we do not detach it, then the backward process will clear all the variable on the graph, including fake. Thus generator won't be update next time when we call fake.

Problem 2: (20%)
Collaboration: B05901027 詹書愷
Reference:
1. https://arxiv.org/abs/1610.09585
2. https://github.com/clvrai/ACGAN-PyTorch

1. (5%)
The structure of the generator:

```
G:  Generator(
  (layer0): Sequential(
    (0): Linear(in_features=101, out_features=384, bias=True)
  )
  (layer1): Sequential(
    (0): ConvTranspose2d(384, 192, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace)
  )
  (layer2): Sequential(
    (0): ConvTranspose2d(192, 96, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): BatchNorm2d(96, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace)
  )
  (layer3): Sequential(
    (0): ConvTranspose2d(96, 48, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): BatchNorm2d(48, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace)
  )
  (layer4): Sequential(
    (0): ConvTranspose2d(48, 24, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): BatchNorm2d(24, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace)
  )
  (layer5): Sequential(
    (0): ConvTranspose2d(24, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): Tanh()
  )
)
```

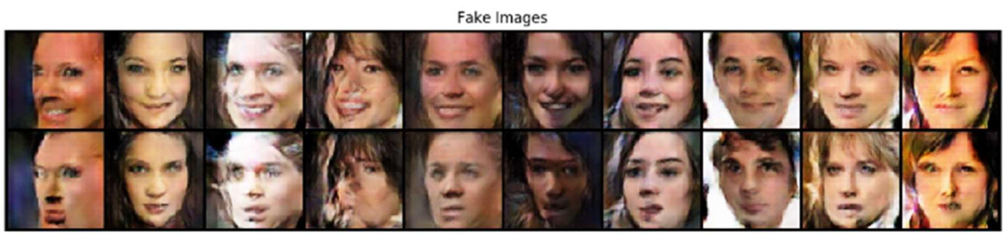
The structure of the discriminator:

```
D:  Discriminator(
  (layer1): Sequential(
    (0): Conv2d(3, 16, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (1): LeakyReLU(negative_slope=0.2, inplace)
    (2): Dropout(p=0.5)
  )
  (layer2): Sequential(
    (0): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), bias=False)
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): LeakyReLU(negative_slope=0.2, inplace)
    (3): Dropout(p=0.5)
  )
  (layer3): Sequential(
    (0): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): LeakyReLU(negative_slope=0.2, inplace)
    (3): Dropout(p=0.5)
  )
  (layer4): Sequential(
    (0): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), bias=False)
    (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): LeakyReLU(negative_slope=0.2, inplace)
    (3): Dropout(p=0.5)
  )
  (layer5): Sequential(
    (0): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): LeakyReLU(negative_slope=0.2, inplace)
    (3): Dropout(p=0.5)
  )
  (layer6): Sequential(
    (0): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), bias=False)
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): LeakyReLU(negative_slope=0.2, inplace)
    (3): Dropout(p=0.5)
  )
  (dis_fc): Linear(in_features=12800, out_features=1, bias=True)
  (ac_fc): Linear(in_features=12800, out_features=2, bias=True)
  (sigmoid): Sigmoid()
)
```

The other hyper-parameters of the my model:

```
n_workers = 4
batch_size = 100
image_size = 64
num_channels = 3
num_classes = 2
z_size = 100  # Size of z latent vector (i.e. size of generator input)
num_epochs = 100
lr = 0.0002
beta1 = 0.5
beta2 = 0.999  # (beta1, beta2) are the hyper-parameters of Adam optim
num_gpus = 1
```

2. (10%)



3. (5%)

At first, my generator's output image doesn't smile when I feeded a noise vector concatenated with a smile label. After I checked my code several times, I found that I made a mistake.

When I'm training the auxiliary classifier along with generator, I feed the true label to classifier, which is not right. I should feed the fake label, which is randomly generated, to the auxiliary classifier. Because now the ac_output(like the figure below) is generated by discriminator, which is feeded by the fake image. So the acxiliary classifier should be feeded the fake label to learn correctly.

```
dis_output, ac_output = netD(fake)
Gloss = dis_criterion(dis_output, dis_label) + ac_criterion(ac_output, fake_smile_label)
```

Problem 3: (35%)

Collaboration:

B05901027 詹書愷、B05602042 林奕廷、B05901074 陳泓均

Reference:

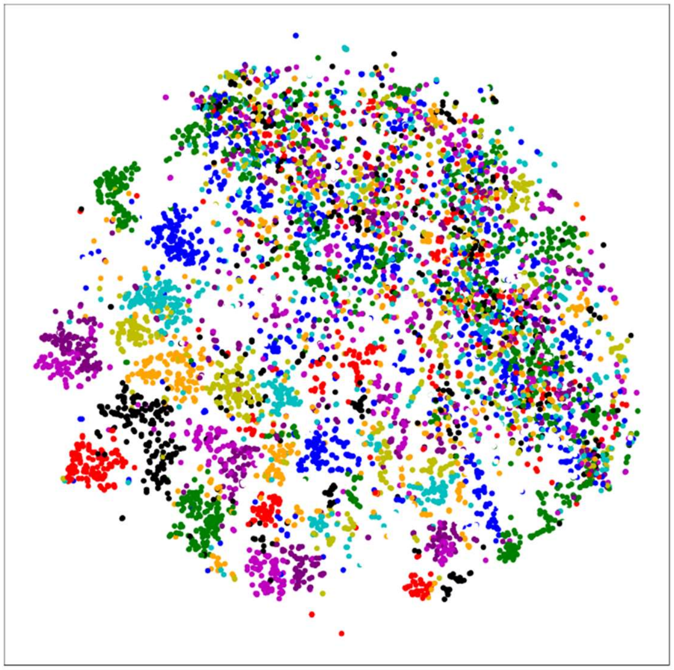
1. http://sites.skoltech.ru/compvision/projects/grl/files/paper.pdf
2. https://github.com/fungtion/DANN_py3

1.(3%)、 2.(10%)、 3. (3%)

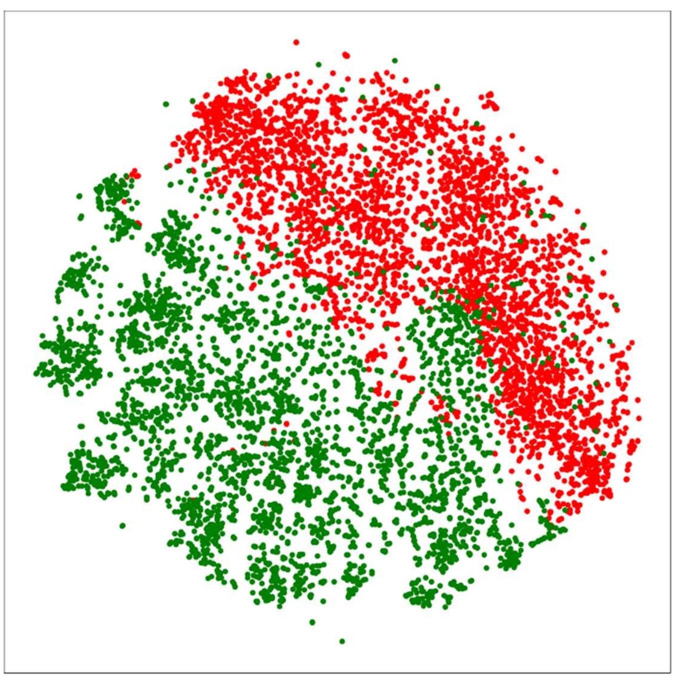
	SVHN -> MNIST-M	MNIST-M -> SVHN
Trained on source	0.4523(45.23%)	0.3891(38.91%)
Adaptation(DANN)	0.4538(45.38%)	0.4484(44.84%)
Trained on target	0.9632(96.32%)	0.8842(88.42%)

4. (6%)

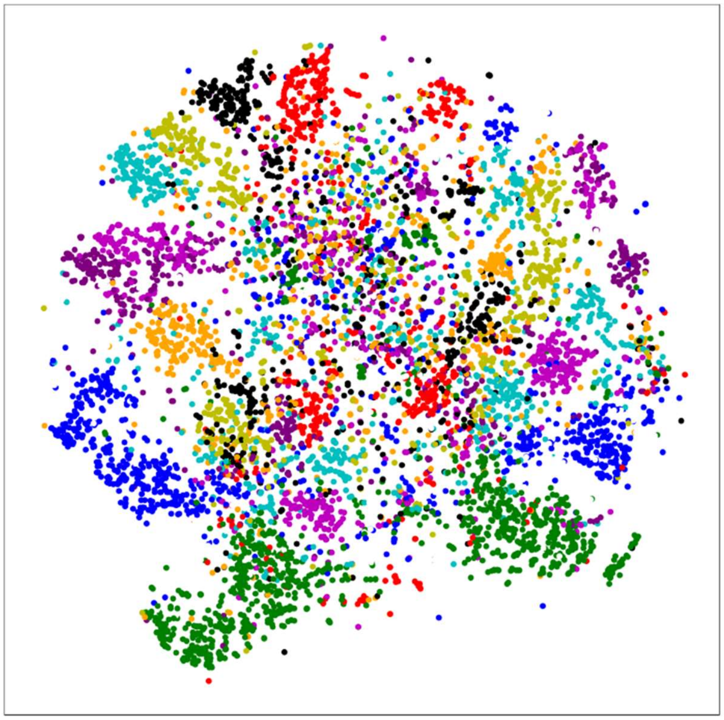
MNIST-M -> SVHN (a) different digit classes 0-9 (I use all test data)



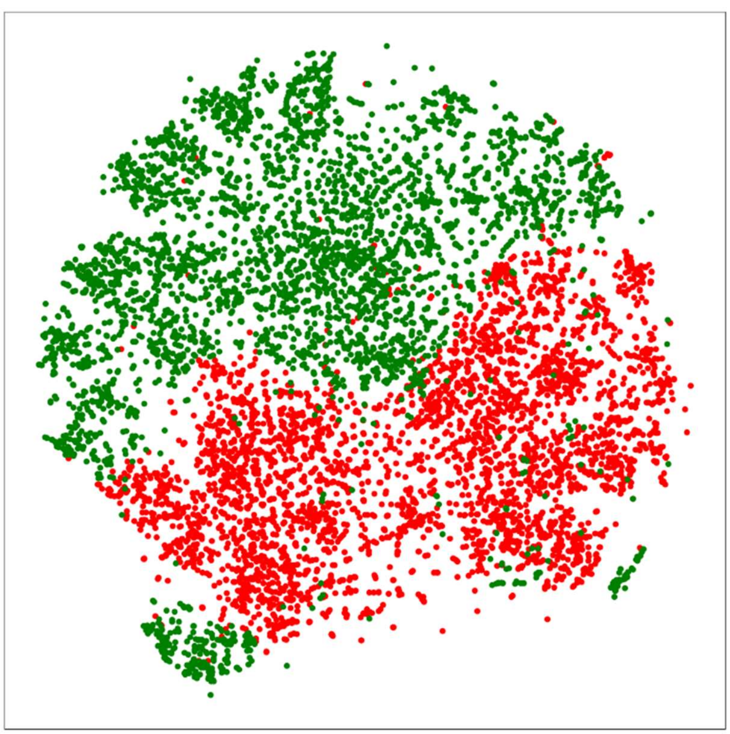
MNIST-M -> SVHN (b) different domains (I use all test data)



SVHN -> MNIST-M (a) different digit classes 0-9 (I use all test data)



SVHN -> MNIST-M (b) different domains (I use part of the data)



5. (6%)

The architecture of the two domain adaptation DANN model:

```
DANN(
  (feature_extractor): Sequential(
    (0): Conv2d(3, 32, kernel_size=(5, 5), stride=(1, 1))
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace)
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (4): Conv2d(32, 48, kernel_size=(5, 5), stride=(1, 1))
    (5): BatchNorm2d(48, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): ReLU(inplace)
    (7): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (class_clf): Sequential(
    (0): Linear(in_features=768, out_features=100, bias=True)
    (1): ReLU(inplace)
    (2): Dropout(p=0.5)
    (3): Linear(in_features=100, out_features=100, bias=True)
    (4): BatchNorm1d(100, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace)
    (6): Linear(in_features=100, out_features=10, bias=True)
  )
  (domain_clf): Sequential(
    (0): Linear(in_features=768, out_features=100, bias=True)
    (1): ReLU(inplace)
    (2): Linear(in_features=100, out_features=2, bias=True)
  )
)
```

The other hyper-parameters of the two domain adaptation DANN model:

```
n_workers = 4
batch_size = 128
num_channels = 3
num_classes = 10
num_epochs = 100
lr = 0.0002
num_gpus = 1
```

6. (7%)

At first, my domain adaptation DANN model's (SVHN -> MNIST-M) performance on accuracy is worse than the model (lower bound) which trained on source and tested on the target(43.06%<45.23%), which is really weird. Thus I checked my model again, and found that I added a dropout layer in the fearture extractor part. My mind came up with an idea: The feature extractor is the dominant part of this model. If I add the dropout layer in this part, which may cause the model to learn incompletely. So I removed the dropout layer, and the accuracy result became 45.38%, which is slightly better than the lower bound.

I also found that the model which trained and tested on different domain data would increase its accuracy while high epoches(~90). But the domain adaptation model reaches its accuracy peak in the early epoches(~30).

Problem 4: (35%)

Collaboration:
B05901027 詹書愷、B05602042 林奕廷、B05901074 陳泓均

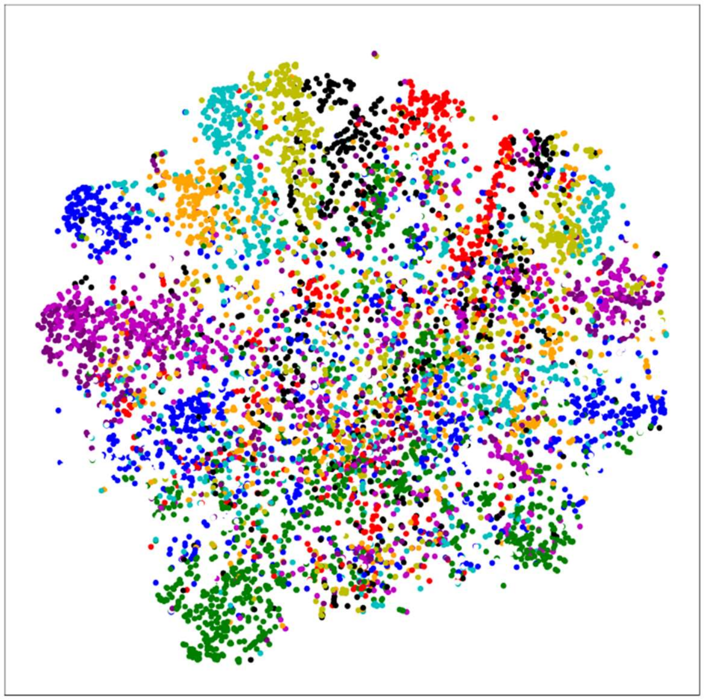
Reference:
1. https://arxiv.org/pdf/1702.05464.pdf
2. https://github.com/corenel/pytorch-adda

1. (6+10%)

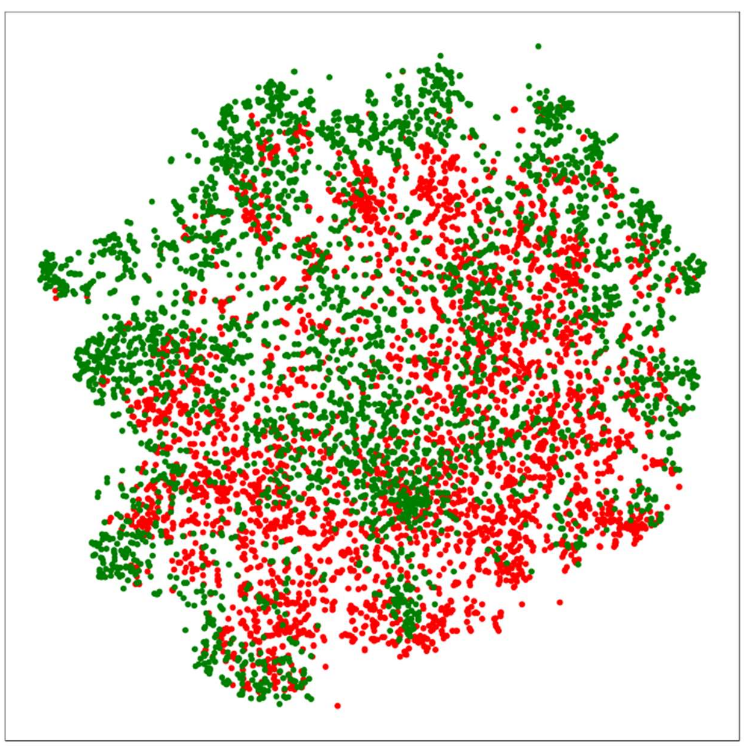
	SVHN -> MNIST-M	MNIST-M -> SVHN
Adaptation(ADDA)	0.5955(59.55%)	0.4581(45.81%)

2. (6%)

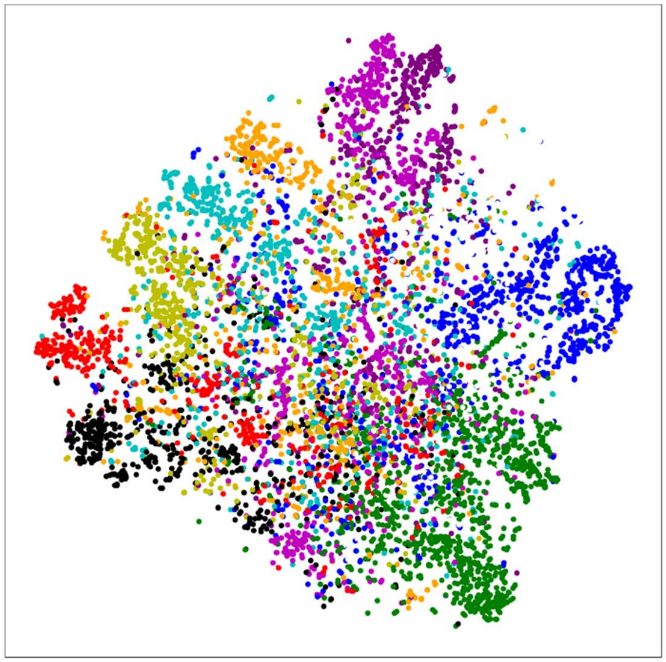
SVHN -> MNIST-M (a) different digits classes 0-9 (I use part of the data)



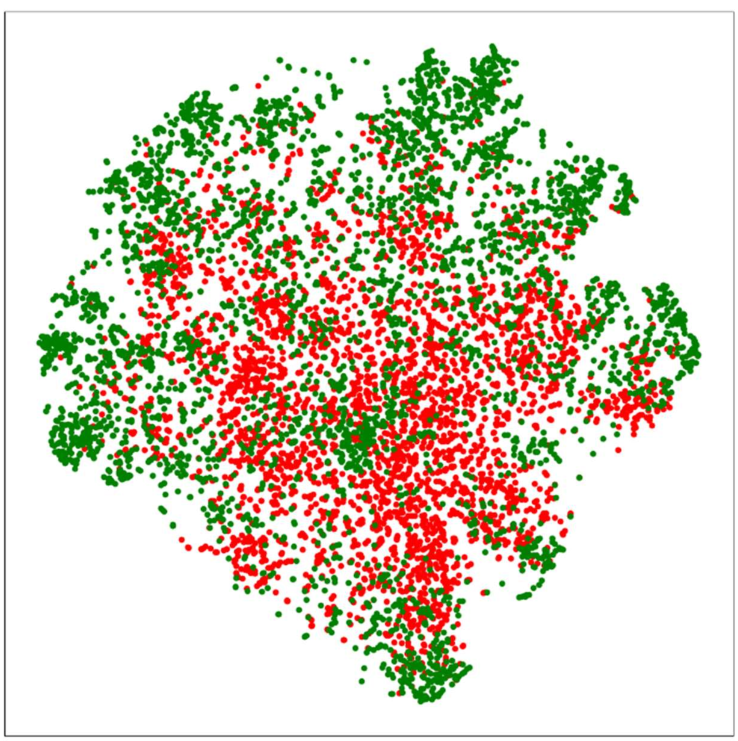
SVHN -> MNIST-M (b) different domains (I use part of the data)



MNIST-M -> SVHN (a) different digits classes 0-9 (I use part of the data)



MNIST-M -> SVHN (b) different domains (I use part of the data)



3. (6%)
My model has three types of sub-model: encoder, classifier and discriminator:

Encoder:

```python
class Encoder(nn.Module):    # modified LeNet
    def __init__(self):
        super(Encoder, self).__init__()
        self.feature_extrator = nn.Sequential(
            nn.Conv2d(1, 20, kernel_size=5, stride=1),
            nn.MaxPool2d(kernel_size=2, stride=2, dilation=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(20, 50, kernel_size=5, stride=1),
            nn.Dropout2d(p=0.5),
            nn.MaxPool2d(kernel_size=2, stride=2, dilation=1),
            nn.ReLU(inplace=True)
        )
        self.fc1 = nn.Linear(800, 500)

    def forward(self, img):
        ft = self.feature_extrator(img)
        ft = ft.view(-1, 800)
        ft = self.fc1(ft)
        return ft
```

Classifier:

```python
class Classifier(nn.Module):
    def __init__(self):
        super(Classifier, self).__init__()
        self.clf = nn.Sequential(
            nn.ReLU(),
            nn.Dropout(p=0.5),
            nn.Linear(500, 10)
        )

    def forward(self, ft):
        prob = self.clf(ft)
        return prob
```

Discriminator:

```python
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.dis = nn.Sequential(
            nn.Linear(500, 500),
            nn.ReLU(inplace=True),
            nn.Linear(500, 2),
        )

    def forward(self, img):
        prob = self.dis(img)

        return prob
```

he other hyper-parameters of the my model:

```python
n_workers = 4
batch_size = 128
num_classes = 10
num_epochs = 100
lr = 0.0002
num_gpus = 1
beta1 = 0.5
```

I trained my source encoder and source classfier for 25 epochs, and initialize the weight of target encoder with the weight of target encoder. Then I trained the target encoder and discriminator for 100 epochs.

4. (7%)
Discuss what you've observed and learned from implementing your improved UDA model.
At first, my model's performance is really bad, the accuracy is between 9~20%, which is not better than the model of problem 3.
By observing the loss between classifier&encoder 、 discriminator, I found that is discriminator is too strong. Thus, I modified my discriminator model to a weaker version(delete one fc layer). And, after training my source encoder, I use the weight of source encoder to initialize my target encoder instead of doing nothing.

By doing so, the performance better better. The SVHN -> MNIST-M model’s accuracy is 59%, which is really out of my expectation.