DLCV HW3 Report

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

Problem 1: GAN

1. Architecture

(1) Generator

```
class Generator(nn.Module):
    def __init__(self, nz):
        super(Generator, self).__init__()
        self.nz = nz
        self.hidden = nn.Sequential(
            nn.Linear(self.nz, 512*4*4),
            nn.LeakyReLU(inplace=True),
        )
        self.net = nn.Sequential(
            nn.ConvTranspose2d(512, 256, kernel_size=4, stride=2, padding=1, bias=False),  # (b, 256, 8, 8)
            nn.BatchNorm2d(256),
            nn.LeakyReLU(inplace=True),
            nn.ConvTranspose2d(256, 128, kernel_size=4, stride=2, padding=1, bias=False),  # (b, 128, 16, 16)
            nn.BatchNorm2d(128),
            nn.LeakyReLU(inplace=True),
            nn.ConvTranspose2d(128, 64, kernel_size=4, stride=2, padding=1, bias=False),  # (b, 64, 32, 32)
            nn.BatchNorm2d(64),
            nn.LeakyReLU(inplace=True),
            nn.ConvTranspose2d(64, 3, kernel_size=4, stride=2, padding=1, bias=False),  # (b, 3, 64, 64)
            nn.Tanh(),
        )
```

The latent vector (noise) will first pass through a fully-connected layer before entering a series of de-convolution layers. The final activation function is Tanh while others are LeakyReLU.

(2) Discriminator

```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.net = nn.Sequential(  # (b, 3, 64, 64)
            nn.Conv2d(3, 64, kernel_size=4, stride=2, padding=1, bias=False),  # (b, 64, 32, 32)
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1, bias=False),  # (b, 128, 16, 16)
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(128, 256, kernel_size=4, stride=2, padding=1, bias=False),  # (b, 256, 8, 8)
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(256, 512, kernel_size=4, stride=2, padding=1, bias=False),  # (b, 512, 4, 4)
            nn.BatchNorm2d(512),
            nn.LeakyReLU(0.2, inplace=True),
        )
        self.hidden = nn.Linear(512*4*4, 1)
        self.sig = nn.Sigmoid()
```

The images will first pass through a series of convolution layers; then, they

will be mapped to a single value through a full-connected layer.

(3) Policy

Learning rate is decreased every epoch using the “poly” policy.

2. Visualization

3. Discussion

For this problem, I have experimented with different model structures and improvement methods, which are:

(1) Noisy label:

By adding noise (randomly flipping) to some of the real/fake labels which decreases with time, it can weaken the discriminator at the beginning of the training, granting the generator more freedom to learn better feature.

(2) Soft label:

I have experimented with the use of soft label, which replaces the real labels with random samples from (0.7, 1.2), and fake labels with random samples from (0, 0.3). While this indeed smooths the images, the images tend to “collapse” more easily.

(3) fc/no fc layers:

I have also experimented with whether fully-connected (fc) layers should be added the generator and the discriminator. I discovered that the addition of fc layers might strengthen the network’s ability to learn facial structures. Hence, even if the results are lousy, the basic structure of generate faces are usually correct. This may be due to the fact that fc layers are more powerful at extracting high-level information.

After some experimentation, my final model is trained with the addition of fc layers and is trained using noisy label.

Problem 2: ACGAN

1. Architecture

(1) Generator

```
class AC_Generator(nn.Module):
    def __init__(self, nz):
        super(AC_Generator, self).__init__()
        self.nz = nz
        self.hidden = nn.Sequential(
            nn.Linear(self.nz, 512 * 4 * 4),
            nn.LeakyReLU(inplace=True),
        )
        self.net = nn.Sequential(
            nn.ConvTranspose2d(512, 256, kernel_size=4, stride=2, padding=1, bias=False),  # (b, 256, 8, 8)
            nn.BatchNorm2d(256),
            nn.LeakyReLU(inplace=True),
            nn.ConvTranspose2d(256, 128, kernel_size=4, stride=2, padding=1, bias=False),  # (b, 128, 16, 16)
            nn.BatchNorm2d(128),
            nn.LeakyReLU(inplace=True),
            nn.ConvTranspose2d(128, 64, kernel_size=4, stride=2, padding=1, bias=False),  # (b, 64, 32, 32)
            nn.BatchNorm2d(64),
            nn.LeakyReLU(inplace=True),
            nn.ConvTranspose2d(64, 3, kernel_size=4, stride=2, padding=1, bias=False),  # (b, 3, 64, 64)
            nn.Tanh(),
        )
```

I use the same generator as that of Problem 1. The only difference that the input of the network is the concatenation of noise and condition.

(2) Discriminator

```
class AC_Discriminator(nn.Module):
    def __init__(self):
        super(AC_Discriminator, self).__init__()
        self.extractor = nn.Sequential(  # (b, 3, 64, 64)
            nn.Conv2d(3, 64, kernel_size=4, stride=2, padding=1, bias=False),  # (b, 64, 32, 32)
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1, bias=False),  # (b, 128, 16, 16)
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(128, 256, kernel_size=4, stride=2, padding=1, bias=False),  # (b, 256, 8, 8)
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(256, 512, kernel_size=4, stride=2, padding=1, bias=False),  # (b, 512, 4, 4)
            nn.BatchNorm2d(512),
            nn.LeakyReLU(0.2, inplace=True),
        )
        self.classifier = nn.Conv2d(512, 2, kernel_size=4, bias=False)  # (b, 2, 1, 1)
        self.fc = nn.Sequential(
            nn.Linear(512*4*4, 1),
            nn.Sigmoid(),
        )
```

The discriminator is also similar to that of Problem 1. However, since the discriminator has to differentiate between real/fake images as well as classify them, the features extracted by the extractor will be sent to both the classifier

and the fc network (which will decide the authenticity of images).

(3) Policy

The learning rate decreases with time using the “poly” policy.

2. Visualization

3. Discussion

Same as Problem 1, I experimented with the same 3 improvements. After different tries, my final model is trained with all improvements: addition of fc layers, noisy label, and soft label.

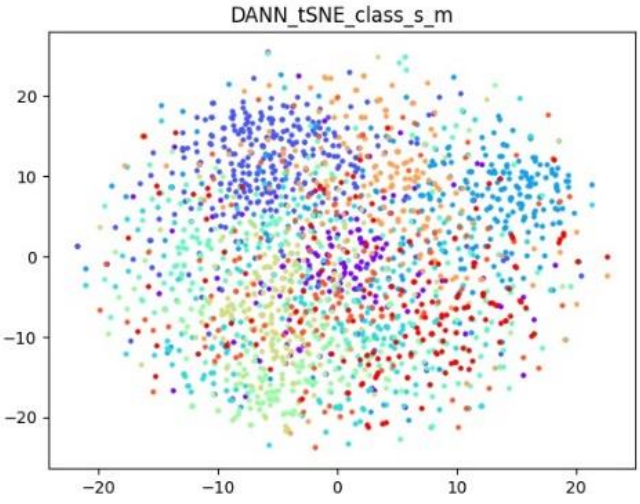
Problem 3: DANN

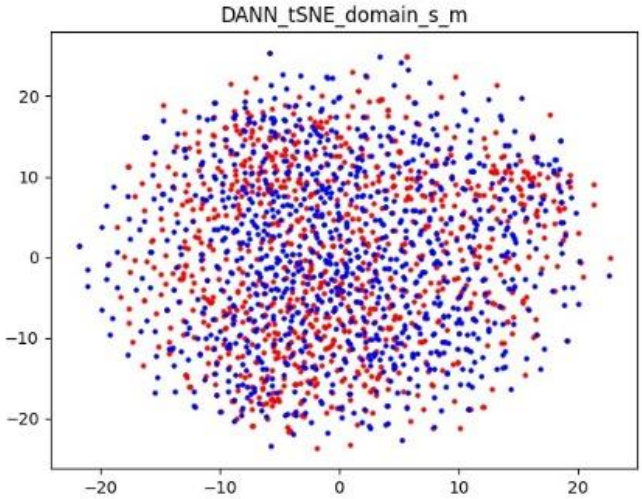
1. Transferring Accuracy

	SVHN -> MNISTM	MNISTM -> SVHN
Trained on Source	43.26 %	42.39 %
DANN	50.05 %	44.71 %
Trained on Target	97.06 %	90.52 %

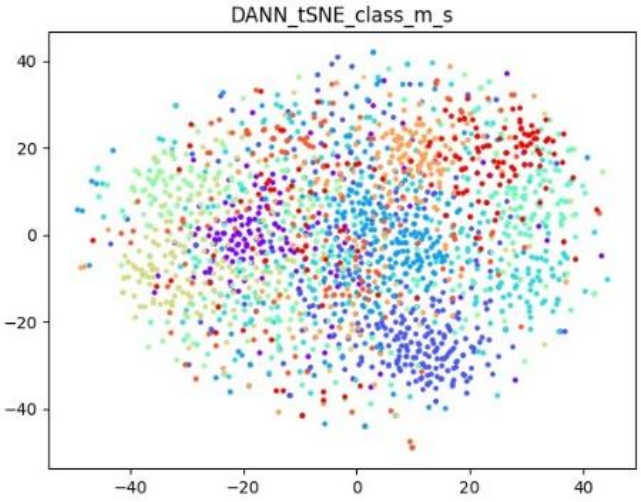
2. T-SNE Visualization

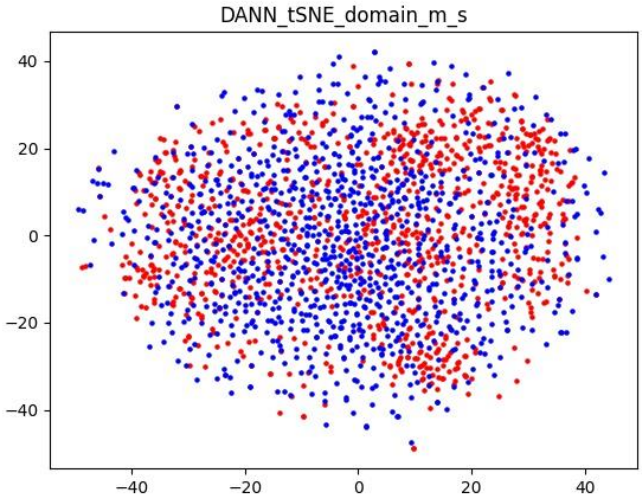
(1) SVHN -> MNISTM





(2) MNISTM -> SVHN





3. Architecture

```
class DANN(nn.Module):
    def __init__(self, n_class=10):
        super(DANN, self).__init__()
        self.n_class = n_class
        self.extractor = nn.Sequential(  # (b, 3, 28, 28)
            nn.Conv2d(3, 128, kernel_size=4, stride=2, padding=1, bias=False),  # (b, 128, 14, 14)
            nn.BatchNorm2d(128),
            nn.Dropout2d(),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(128, 64, kernel_size=4, stride=2, padding=1, bias=False),  # (b, 64, 7, 7)
            nn.BatchNorm2d(64),
            nn.Dropout2d(),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(64, 32, kernel_size=5),  # (b, 32, 3, 3)
            nn.BatchNorm2d(32),
            nn.Dropout2d(),
            nn.LeakyReLU(0.2, inplace=True),
        )
        self.lbl_classifier = nn.Sequential(
            nn.Linear(32*3*3, 100),
            nn.Dropout(),
            nn.ReLU(inplace=True),
            nn.Linear(100, self.n_class),
        )
        self.domain_classifier = nn.Sequential(
            nn.Linear(32*3*3, 100),
            nn.Dropout(),
            nn.ReLU(inplace=True),
            nn.Linear(100, 1),
            nn.Sigmoid(),
```

In a lot of ways, the structure of DANN is very similar to that of the discriminator of ACGAN, the only difference being that the “generator” is replaced by gradient

reversal, which is implemented by adding a Reversal Layer (shown as below) before entering domain classifier in order to ensure that the domain is in fact being

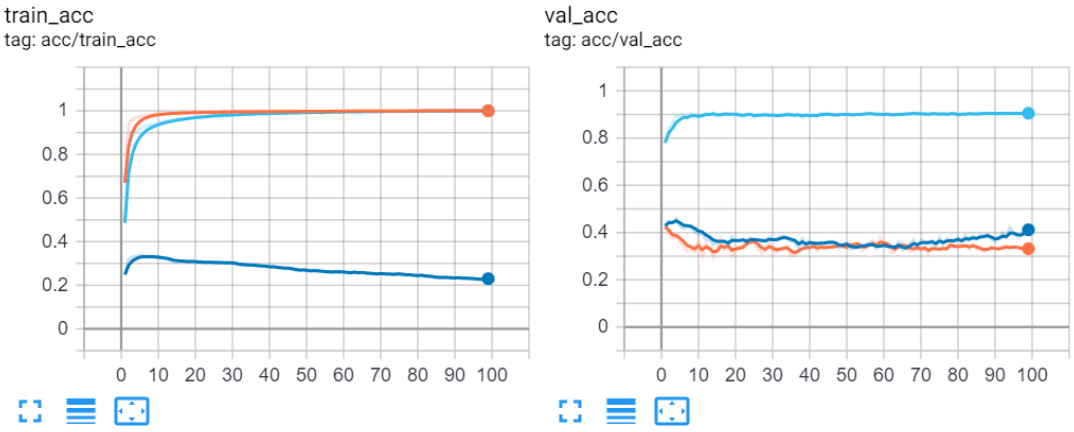
```
class ReverseLayer(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x, alpha):
        ctx.alpha = alpha
        return x

    @staticmethod
    def backward(ctx, grad_output):
        output = grad_output.neg() * ctx.alpha
        return output, None
```

confused.

4. Discussion

The idea of reversing the gradient is quite novel to me. Even though the improvement is not significant, it is clearly shown in the training curve (the dark blue curve) that with more iteration, the accuracy of the training set (source) steadily decreases in exchange for the increase of the validation accuracy (target).



Problem 4: Improved UDA -

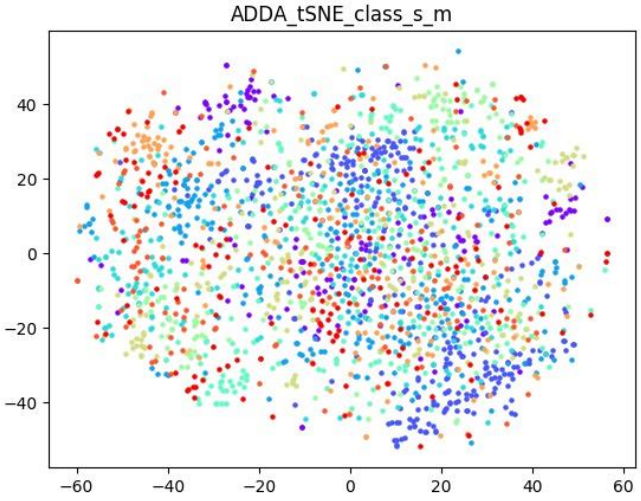
Adversarial Discriminative Domain Adaptation (ADDA)

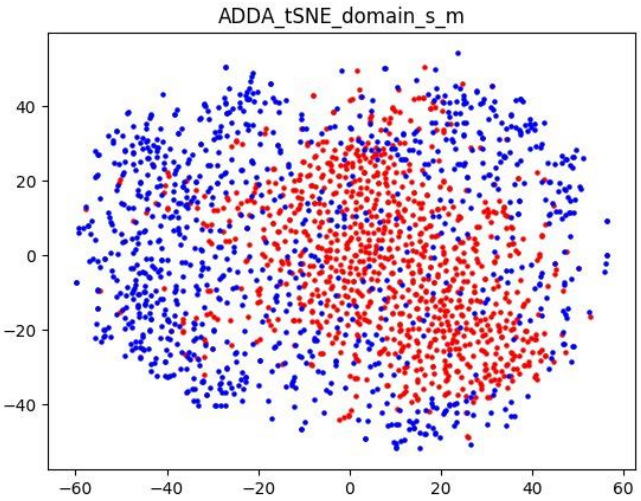
1. Transferring Accuracy

	SVHN -> MNISTM	MNISTM -> SVHN
ADDA	52.03 %	47.25 %

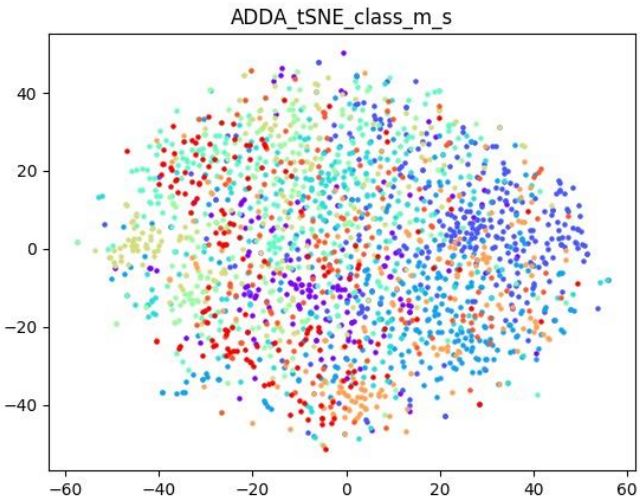
2. T-SNE Visualization

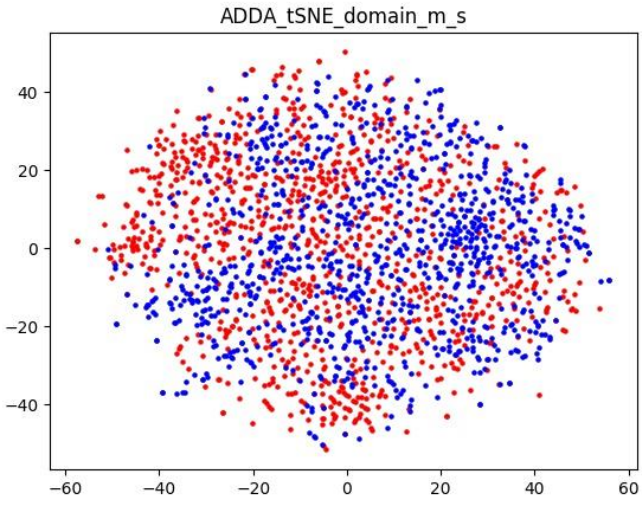
(1) SVHN -> MNISTM





(2) MNISTM -> SVHN





3. Architecture

(1) Discriminator

```
class D(nn.Module):
    def __init__(self, in_size=256*4*4):
        super(D, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(in_size, 256),
            nn.LeakyReLU(0.2, True),
            nn.Dropout(),
            nn.Linear(256, 256),
            nn.LeakyReLU(0.2, True),
            nn.Dropout(),
            nn.Linear(256, 1),
            nn.Sigmoid(),
        )
```

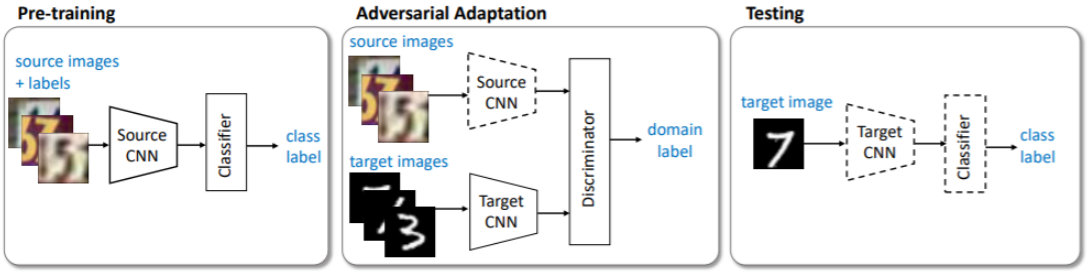
(2) Extractor (Generator)

```
class E(nn.Module):
    def __init__(self, in_ch=3, out_ch=256):
        super(E, self).__init__()
        self.net = nn.Sequential(  # (b, 3, 28, 28)
            nn.Conv2d(in_ch, 64, kernel_size=4, stride=2, padding=1, bias=False),  # (b, 64, 14, 14)
            nn.LeakyReLU(0.2, True),
            nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1, bias=False),  # (b, 128, 7, 7)
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2, True),
            nn.Dropout2d(),
            nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1, bias=False),  # (b, 128, 7, 7)
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2, True),
            nn.Dropout2d(),
            nn.Conv2d(128, out_ch, kernel_size=4, stride=1, padding=0, bias=False),  # (b, out_ch, 4, 4)
            nn.Tanh(),
        )
```

(3) Classifier

```
class C(nn.Module):
    def __init__(self, in_size=256*4*4, n_classes=10):
        super(C, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(in_size, 100),
            nn.Dropout(),
            nn.ReLU(True),
            nn.Linear(100, n_classes),
        )
```

(4) Details



ADDA views the extractor (CNN) as an embedding "generator", which can be trained in an adversarial manner with a domain discriminator. Hence, the main transfer model consists of two extractors, one for source and one for target. After pre-training the extractors on the source dataset, the target CNN will be encouraged to extract similar features from the target dataset to that of the source dataset, therefore transferring the knowledge.

4. Discussion

It took a while for me to balance the training between the target extractor (G) and the discriminator (D). The following are some tricks and observations:

(1) Strided convolution & LeakyReLU:

The original paper trained ADDA using the simplified LeNet architecture, which uses MaxPooling to subsample; however, sparse gradient can lead to instablility when training GAN. Following the advice from **ganhacks** (https://github.com/soumith/ganhacks), I altered the network and applied strided convolution and LeakyReLU for the sake of stability. The training soon stabilized.

(2) Dropout:

During training, I find that the loss of D often decreases too quickly, causing the loss of G to diverge. After applying Dropout to every layer of G and D, the loss of D decreased slower, and the training became more robust; I also encountered less sudden drop in accuracy.

(3) Noisy label:

As mention in Problem 1, this is a useful trick to strengthen the generator at first. Combined with (1) and (2), I achieved my best result for ADDA.