Ting-An Chen

Homework #2

Deep Learning for Computer Vision

Problem 1: GAN (35%)

1. Build your generator and discriminator from scratch and show your model architecture in your report (You can use “print(model)” in PyTorch directly). Then, train your model on the face dataset and describe implementation details. (Include but not limited to training epochs, learning rate schedule, data augmentation, and optimizer) (5%)

Model architecture:

```
Generator(
  (tconv1): ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (tconv2): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (tconv3): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (bn3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (tconv4): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (bn4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (tconv5): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
)
Discriminator(
  (conv1): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (conv2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv4): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (bn4): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv5): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
)
```

Implementation details

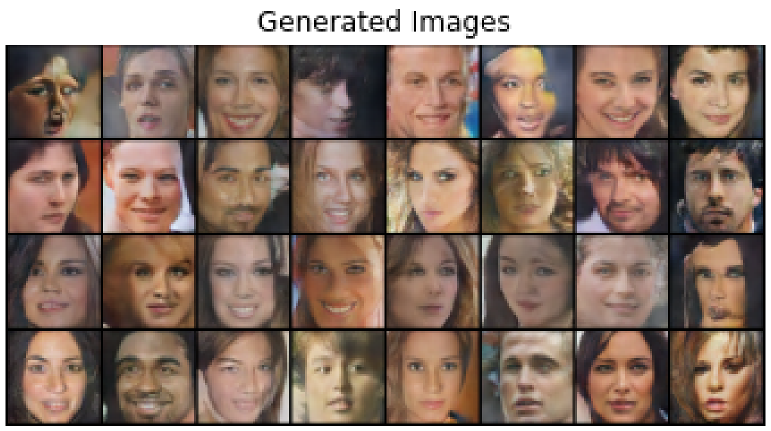
I. Model

- **Discriminator:** to discriminate the real and fake input data
- **Generator:** to generate fake input data (from noise vectors) that the discriminator is hardly to identify

III. Hyperparameters

- **Learning rate:** 0.0005 (decay rate: 0.1)
- **Dimension of noise vectors:** 100
- **Optimizer**: Adam
- **Epoch:** 200
- **Data augmentation:** only resize, crop, and normalize the image

2. Please sample 1000 noise vectors from Normal distribution and input them into your Generator. Save the 1000 generated images in the assigned folder path for evaluation, and show the first 32 images in your report. (5%)



3. Evaluate your 1000 generated images by implementing two metrics:
   (1) Fréchet inception distance (FID): **29.787**
   (2) Inception score (IS): **2.018**

4. vs. Baseline (20%)

5. Discuss what you've observed and learned from implementing GAN. (5%)

   This is my first time using noise vectors as inputs rather than the real image data. It is interesting and can hardly imagine GAN is feasible to be realized. However, it is still hard for me to generate the faces as the real person's faces. It might be that the two models, the generator and the discriminators, focus on different targets.

Problem 2: ACGAN (30%)

1. Build your ACGAN model from scratch and show your model architecture in your report (You can use "print(model)" in PyTorch directly). Then, train your model on the mnistm dataset and describe implementation details. (e.g. How do you input the class labels into the model?) (10%)

   Model architecture:

```
Generator(
  (label_emb): Embedding(10, 15)
  (l1): Sequential(
    (0): Linear(in_features=15, out_features=6272, bias=True)
  )
```

```
 (conv_blocks): Sequential(
   (0): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
   (1): Upsample(scale_factor=2.0, mode=nearest)
   (2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
   (3): BatchNorm2d(128, eps=0.8, momentum=0.1, affine=True, track_running_stats=True)
   (4): LeakyReLU(negative_slope=0.2, inplace=True)
   (5): Upsample(scale_factor=2.0, mode=nearest)
   (6): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
   (7): BatchNorm2d(64, eps=0.8, momentum=0.1, affine=True, track_running_stats=True)
   (8): LeakyReLU(negative_slope=0.2, inplace=True)
   (9): Conv2d(64, 3, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
   (10): Tanh()
 )
)
Discriminator(
 (conv_blocks): Sequential(
   (0): Conv2d(3, 16, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
   (1): LeakyReLU(negative_slope=0.2, inplace=True)
   (2): Dropout2d(p=0.25, inplace=False)
   (3): Conv2d(16, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
   (4): LeakyReLU(negative_slope=0.2, inplace=True)
   (5): Dropout2d(p=0.25, inplace=False)
   (6): BatchNorm2d(32, eps=0.8, momentum=0.1, affine=True, track_running_stats=True)
   (7): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
   (8): LeakyReLU(negative_slope=0.2, inplace=True)
   (9): Dropout2d(p=0.25, inplace=False)
   (10): BatchNorm2d(64, eps=0.8, momentum=0.1, affine=True, track_running_stats=True)
   (11): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
   (12): LeakyReLU(negative_slope=0.2, inplace=True)
   (13): Dropout2d(p=0.25, inplace=False)
   (14): BatchNorm2d(128, eps=0.8, momentum=0.1, affine=True, track_running_stats=True)
 )
 (adv_layer): Sequential(
   (0): Linear(in_features=512, out_features=1, bias=True)
   (1): Sigmoid()
 )
 (aux_layer): Sequential(
   (0): Linear(in_features=512, out_features=10, bias=True)
   (1): Softmax(dim=None)
 )
)
```

Implementation details:

The noise vectors and randomly given labels are as inputs of the generator to generate the fake images with the given digits. After then, the discriminator will discriminate the fake images and the real images.

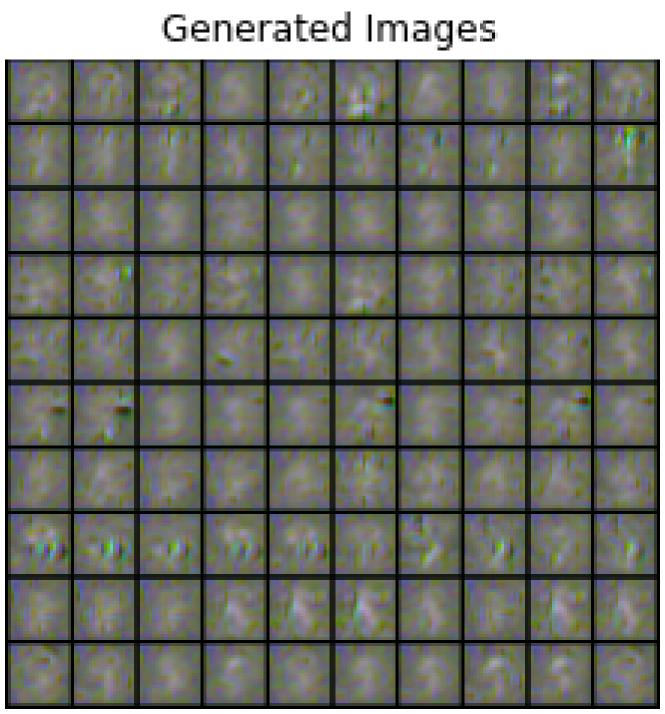
2. Sample random noise and generate 100 conditional images for each digit (0-9). Save a total of 1000 outputs in the assigned folder path for further evaluation. We will provide a digit classifier to evaluate your output images.

3. You should name your output digit images as the following format: (The first number of your filename indicates the corresponding digit label)

4. We will evaluate your generated output by the classification accuracy with a pretrained digit classifier, and we have provided the model architecture [digit_classifer.py] and the weight [Classifier.pth] for you to test. (15%)

   The accuracy: **83.0%**

5. Show 10 images for each digit (0-9) in your report. You can put all 100 outputs in one image with columns indicating different digits and rows indicating different noise inputs. [see the below example] (5%)



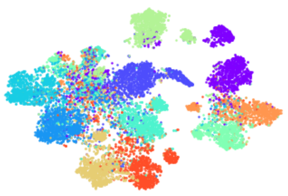
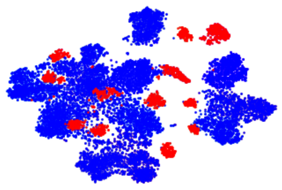
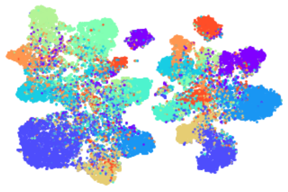
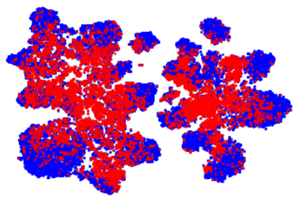
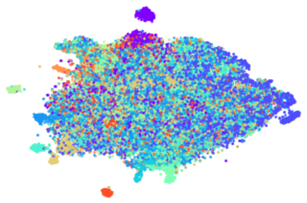
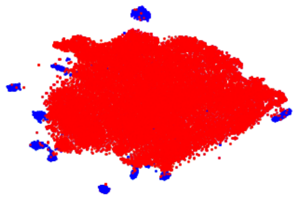
   E.g. The first row is 0. The last row is 9.

Problem 3: DANN (35%)

1. Compute the accuracy on the target domain, while the model is trained on the source domain only. (lower bound) (3%)
2. Compute the accuracy on the target domain, while the model is trained on the source and target domain. (domain adaptation) (4+9%)
3. Compute the accuracy on the target domain, while the model is trained on the target domain only. (upper bound) (3%)

Training	MNIST-M → USPS	SVHN → MNIST-M	USPS → SVHN
Source only	85.75%	50.27%	23.03%
Adaptation	87.29%	51.61%	28.21%
Target only	93.32%	96.21%	88.12%

4. Visualize the latent space by mapping the testing images to 2D space with **t-SNE** and use different colors to indicate data of (a) different digit classes 0-9 and (b) different domains (source/target). (9%)
    ○ Note that you need to plot the figures of the three scenarios, so you would need to plot 6 figures in total in this sub-problem. (1.5% for each figure)

Task	(a) Class	(b) Domain (Source/Target)
MNIST-M → USPS		
SVHN → MNIST-M		
USPS → SVHN		

5. Describe the implementation details of your model and discuss what you've observed and learned from implementing DANN. (7%)

   Details of DANN:

   We use **ResNet-50** as the backbone of the feature extractor with the **pertained weights** from **ImageNet**. In addition, we use two fully connected layers as the classifier. The **loss** of DANN includes three terms: 1) the prediction loss of the source domain data, 2) the loss for the source domain discrimination and 3) the loss for the target domain discrimination.

   Observation:

   According to the table shown in Problem 3.1 to 3.3, the performance of the domain adaptation is usually between the performances of only using source data in training and only adopting target data, since the classification on a different domain is generally more challenging than on the same domain.

   In addition, as shown in the t-SNE in Problem 3.4, we can observe that DANN enables aligning the source and the target domains so that the source data can be trained to predict the target data.

Bonus: Improved UDA model (6%)

1. Compute the accuracy on the target domain, while the model is trained on the source and target domain. (**domain adaptation**) (3%)

Training	MNIST-M → USPS	SVHN → MNIST-M	USPS → SVHN
DANN	87.29%	51.61%	28.21%
DSAN (Improved)	88.99%	53.69%	30.64%

2. Briefly describe implementation details of your model and discuss what you've observed and learned from implementing your improved UDA model. (3%)

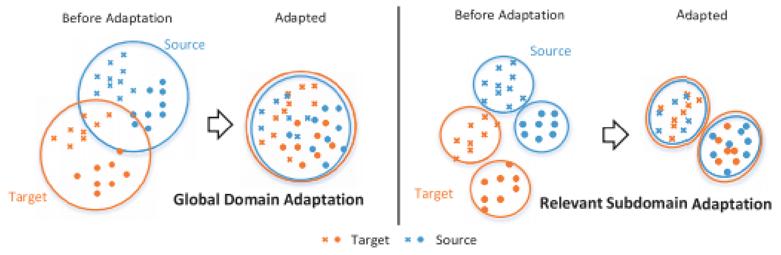
   Details of DSAN:

The improved UDA model, **DSAN** [1], learns a transfer network by aligning the relevant subdomain distributions of domain-specific layer activations across different domains based on a local maximum mean discrepancy (LMMD). DSAN [1] uses the LMMD to align the domains by considering the class-level domains. The features of images should be close in the same class, while should be distant in different classes.
The formula of LMMD:

$$d_{\mathcal{H}}(p, q) \triangleq \mathbf{E}_c \| \mathbf{E}_{p^{(c)}}[\phi(\mathbf{x}^s)] - \mathbf{E}_{q^{(c)}}[\phi(\mathbf{x}^t)] \|_{\mathcal{H}}^2 \qquad (5)$$

where $\mathbf{x}^s$ and $\mathbf{x}^t$ are the instances in $\mathcal{D}_s$ and $\mathcal{D}_t$, and $p^{(c)}$ and $q^{(c)}$ are the distributions of $\mathcal{D}_s^{(c)}$ and $\mathcal{D}_t^{(c)}$, respectively.

The goal of DSAN is to minimize the discrepancy between the source and the target features conditional to each class, noted as c.



Observation:

DANN only considers the domain-level relationships, i.e., aligns the features extracted from the source and the target domain by adversarial learning. However, DANN ignores the class-level relationships considered by DSAN. Therefore, DANN has lower accuracies than DSAN in domain adaptation tasks.

Reference

- [1] Zhu, Y., Zhuang, F., Wang, J., Ke, G., Chen, J., Bian, J., ... & He, Q. (2020). Deep subdomain adaptation network for image classification. *IEEE transactions on neural networks and learning systems*, *32*(4), 1713-1722.