Deep Learning for Computer Vision

Homework #2

經濟四 李品樺 **B07303024**

Collaborators: None

Problem 1: GAN

- **Build your generator and discriminator from scratch and show your model architecture in your report. 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)**

```
======= Generator =======
G(
  (main): 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=True)
    (3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU(inplace=True)
    (9): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (11): ReLU(inplace=True)
  )
  (out): Sequential(
    (0): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): Tanh()
  )
)

======= Discriminator =======
D(
  (main): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (6): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (7): LeakyReLU(negative_slope=0.2, inplace=True)
    (8): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (9): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): LeakyReLU(negative_slope=0.2, inplace=True)
    (11): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (12): Sigmoid()
  )
)
```

- We use Adam as optimizer with 0.0002 learning rate, and $\beta_1 = 0.5, \beta_2 = 0.999$ for both Generator and Discriminator. The Generator render an image from a 100-dimension vector from Gaussian distribution to a face image, and the discriminator is a binary classification problem to tell the whether the image is real or fake. For data augmentation, we randomly flip the training image horizontally. It is useful to implement smooth label. For example, when feeding the "True" labels into the model, instead of feeding 1, we randomly assign a number from 0.85 to 1.2 so as to give the output some flexibility. On the other hand, when feeding the "Fake" labels into the model, we randomly assign a number from 0 to 0.3 instead of giving all zeros. After training for 500 epochs, we can find out that the loss saturates.

- **First 32 images**

- **Evaluate your 1000 generated images by implementing two metrics:**

  (1) Fréchet inception distance (FID): 24

  (2) Inception score (IS): 2.1

- **Discuss what you've observed and learned from implementing GAN.**

  From implementing GAN, we've learned to build a model from scratch and start to modify the architecture by observing its output. When I first built a GAN model, the architecture of Discriminator was too complicated and strong. Thus, it could easily tell the difference

between a fake and a real image. On the other hand, the Generator model is not strong enough to learn from the original training dataset. Thus, I removed some layers from the Discriminator model and the performance improved. Moreover, it is better not to fix the random seed when training.

- **Reference:** DCGAN Tutorial — PyTorch Tutorials 1.10.0+cu102 documentation

Problem 2: ACGAN

- **Build your ACGAN model from scratch and show your model architecture in your report. Then, train your model on the mnistm dataset and describe implementation details.**

```
======= Generator =======
G(
  (main): Sequential(
    (0): ConvTranspose2d(110, 896, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): BatchNorm2d(896, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): ConvTranspose2d(896, 448, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (4): BatchNorm2d(448, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): ConvTranspose2d(448, 224, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (7): BatchNorm2d(224, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU(inplace=True)
    (9): ConvTranspose2d(224, 112, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (10): BatchNorm2d(112, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (11): ReLU(inplace=True)
    (12): ConvTranspose2d(112, 112, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (13): BatchNorm2d(112, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (14): ReLU(inplace=True)
    (15): ConvTranspose2d(112, 28, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (16): BatchNorm2d(28, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (17): ReLU(inplace=True)
  )
  (out): Sequential(
    (0): ConvTranspose2d(28, 3, kernel_size=(1, 1), stride=(1, 1), padding=(2, 2), bias=False)
    (1): Tanh()
  )
)
```

```
======= Discriminator =======
D(
  (layer1): Sequential(
    (0): Conv2d(3, 28, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Dropout(p=0.2, inplace=False)
  )
  (layer2): Sequential(
    (0): Conv2d(28, 56, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): BatchNorm2d(56, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): LeakyReLU(negative_slope=0.2, inplace=True)
    (3): Dropout(p=0.3, inplace=False)
  )
  (layer3): Sequential(
    (0): Conv2d(56, 112, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): BatchNorm2d(112, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): LeakyReLU(negative_slope=0.2, inplace=True)
    (3): Dropout(p=0.3, inplace=False)
  )
  (layer4): Sequential(
    (0): Conv2d(112, 448, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): BatchNorm2d(448, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): LeakyReLU(negative_slope=0.2, inplace=True)
    (3): Dropout(p=0.5, inplace=False)
  )
  (layer5): Sequential(
    (0): Conv2d(448, 28, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
  )
  (linear_dis): Linear(in_features=28, out_features=1, bias=True)
  (linear_aux): Linear(in_features=28, out_features=10, bias=True)
  (sigmoid): Sigmoid()
  (softmax): Softmax(dim=None)
)
```

- ACGAN feeds not only the image but also the condition into the model. To give the model additional information of the digit (condition), we render a 100-dimension random noise from Gaussian distribution. Then, we encode the digit into a 10-dimension one hot vector. For example, digit 0 represents [1,0,0,0,0,0,0,0,0,0]. We concatenate the noise with the 10-dimension vector as the input of Generator. Thus, the input channels of the Generator are $100 + 10 = 110$. As for the output layers of Discriminator, we need two classifiers. One is for truth/fake prediction, and another is for label prediction. Other training details are the same as in Problem 1. Note that using the smooth label technique and add dropout to the Discriminator, it can achieve a better result. It is important to track the output image and loss of the generator since sometimes the loss will increase a lot at the early stage thus leads to unsatisfactory synthesis results.
- **Accuracy using the provided classifier:** 96%

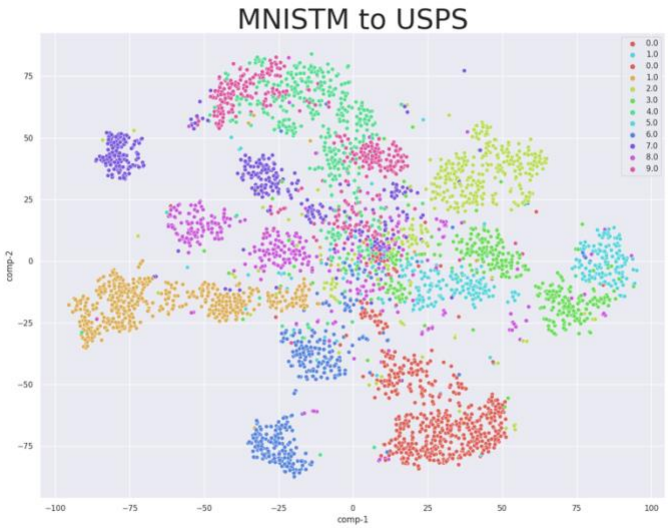
- **10 images for each digit (0-9)**

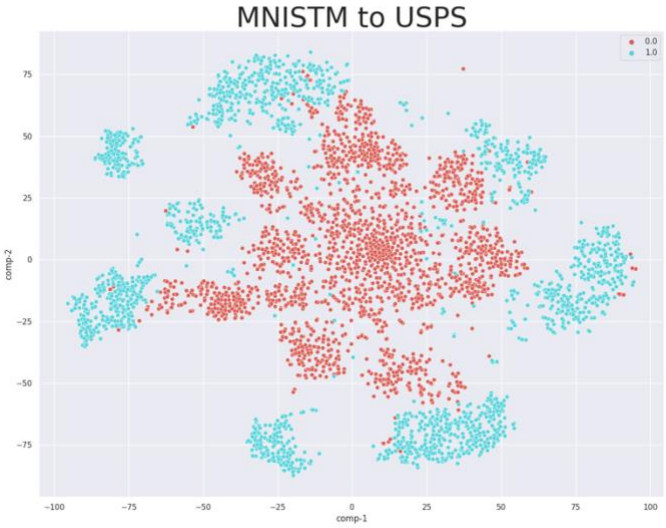
- **Reference:** ACGAN by PyTorch | Kaggle

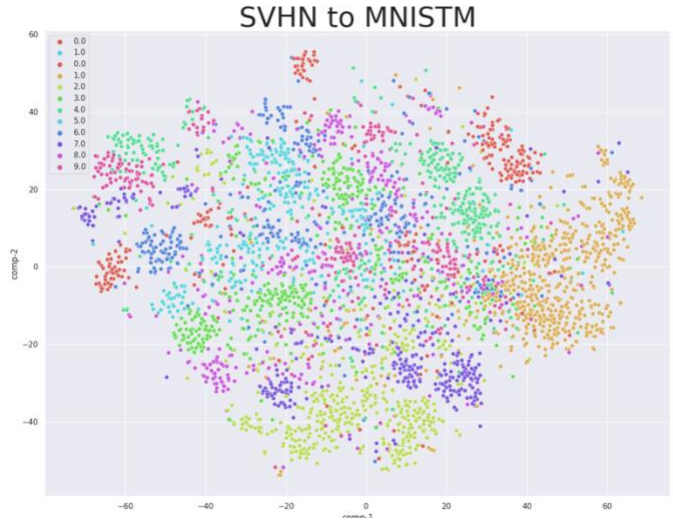
Problem 3: DANN

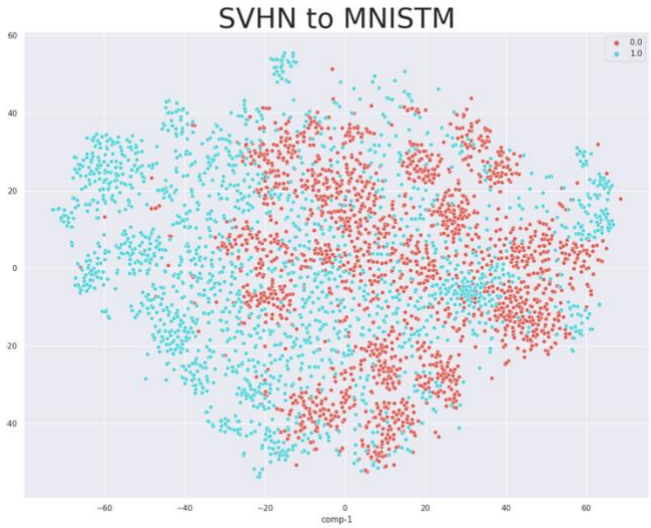
	MNIST-M → USPS	SVHN → MNIST-M	USPS → SVHN
Trained on source	73.19%	37.2%	21.25%
DANN Adaptation	74.94%	46.81%	28.05%
Trained on target	96.96%	96.32%	91.16%

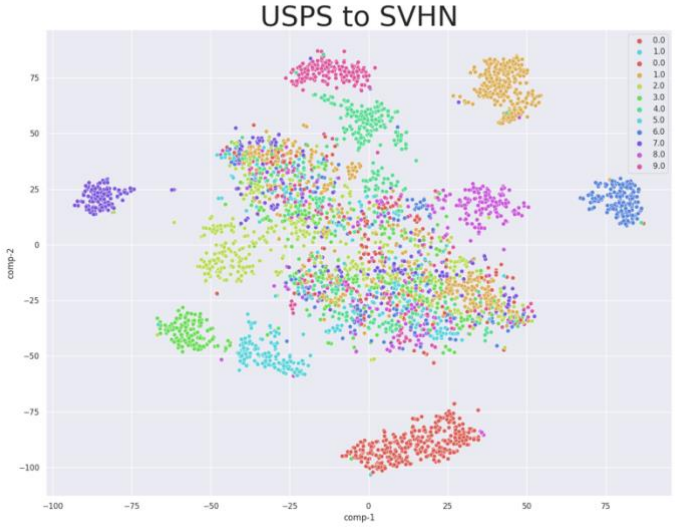
- **Visualize the latent space by mapping the testing images to 2D space with t-SNE and use different colors to indicate data of (a) different digit classes 0-9 and (b) different domains (source/target).**

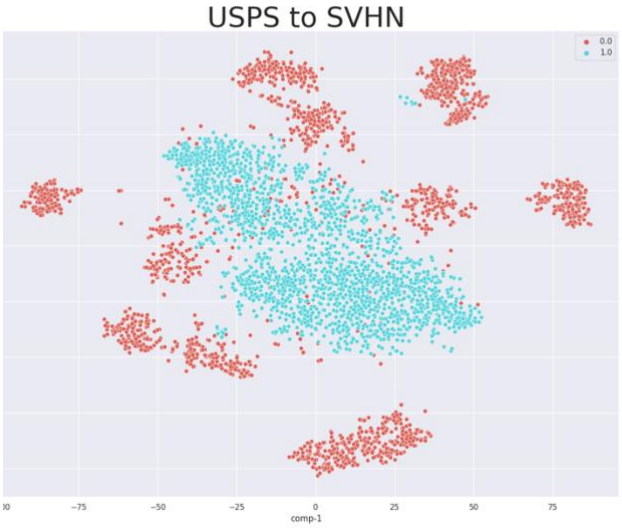












- **Describe the implementation details of your model and discuss what you've observed and learned from implementing DANN.**

  For this problem, we use a Feature Extractor, a Label Predictor and a Domain Classifier to perform domain adaptive problem. First, we want to train the Domain Classifier. Concatenate the source domain data and target domain data and feed the result into Feature Extractor to get the mixed domain feature. The loss of the domain classifier needs to be subtracted from the digit loss. In order to control the importance of domain adaptiveness, we multiply a domain adaptation parameter λ_p to the domain loss. Finally, we backward the result. Then, we train the Feature Extractor and Label predictor. Use the source domain data and label to predict the digit, and backward the loss as well. We choose binary cross entropy for Domain classifier, and Cross Entropy Loss for the digit prediction. Using Adam as optimizers for all, with 0.002 learning rate, the model can achieve a good result.

  From the figures above, we can observe that the first column indicates the different digits of data, the second column indicates the different domains of data. Every color represents a different class.

Bonus: Improved UDA model

	MNIST-M → USPS	SVHN → MNIST-M	USPS → SVHN
Original model	74.94%	46.81%	28.05%
Improved model	76.88%	49.44%	75.13%

- According to the original paper, we modify the domain adaptation parameter $\lambda_p = \frac{2}{1+\exp(-10*p)} - 1$, where p is the training progress linearly changing from 0 to 1. In the beginning stage of training, p=0, which leads to $\lambda_p = 0$ as well. In this case, it allows the domain classifier of the Discriminator to be less sensitive to the noisy labels in the beginning. Using the same model but only a minor difference in loss function can help us achieve significant improvement. As the table above shows, domain adaptation from USPS to SVHN performs the best of all scenarios, reaching an accuracy of 75.13%. From this problem, we learned the importance of modifying loss function and dynamically adjusting the parameter so as to outperform the original model.
- Reference: Domain-Adversarial Training of Neural Networks