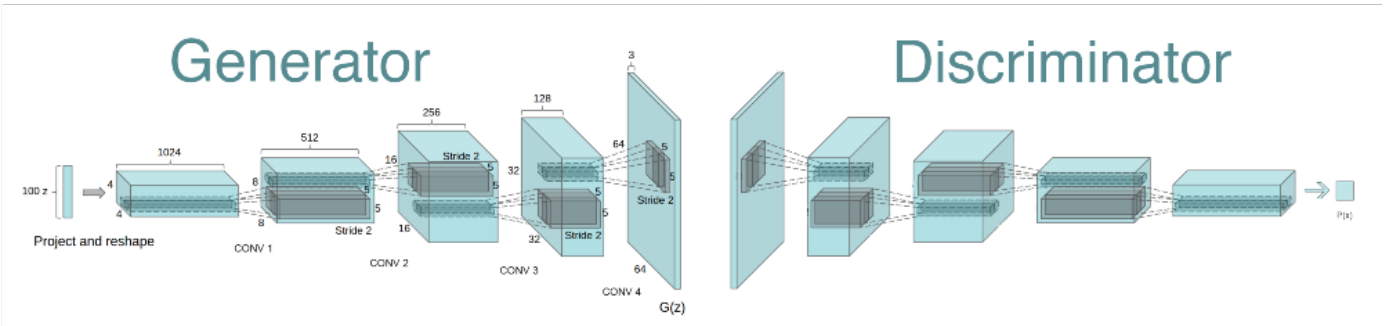
Report HW3 DLCV
STUDENT ID: R07943158

Problem 1: GAN (20%)

1. **Describe the architecture & implementation details of your model. (5%)**
   DCGAN is one of the popular and successful network design for GAN. It mainly composes of convolution layers without max pooling or fully connected layers. It uses convolutional stride and transposed convolution for the downsampling and the upsampling. The figure below is the network design for the generator and discriminator.



All models were trained with mini-batch ADAM optimizer with a mini-batch size of 128. All weights were initialized from a zero-centered Normal distribution with standard deviation 0.02. In the LeakyReLU, the slope of the leak was set to 0.2 in all models

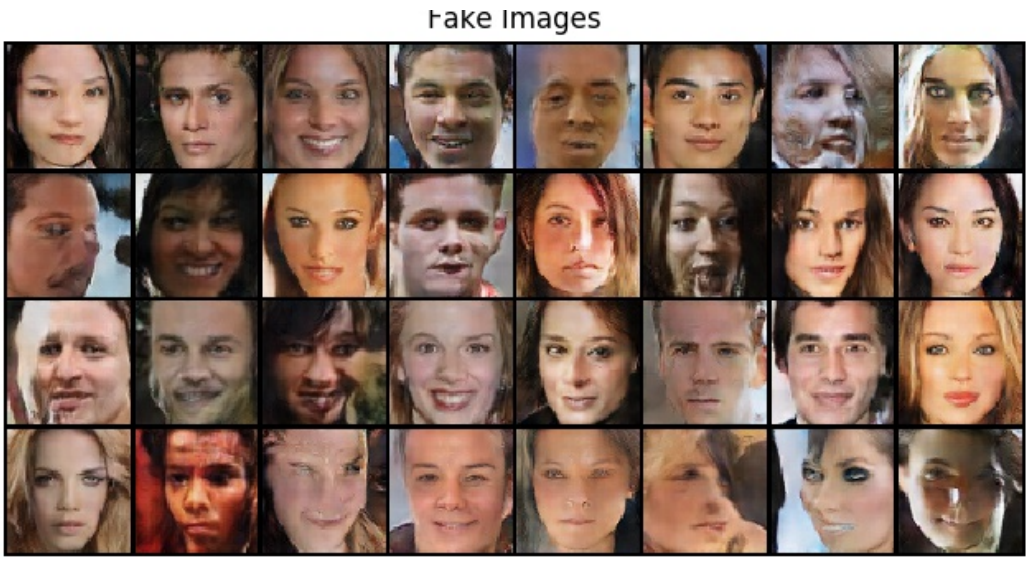
I use BCE Loss Function. I found leaving the momentum term β1 to 0.5 helped stabilize training.

```
Generator(
  (gen): Sequential(
    (0): ConvTranspose2d(101, 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)
    (12): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (13): Tanh()
  )
)
Discriminator(
  (dis): 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)
  )
  (output): Sequential(
    (0): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): Sigmoid()
  )
  (classifier): Sequential(
    (0): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): Sigmoid()
  )
)
```


Reference:

1. Unsupervised representation learning with Deep convolutional generative adversarial networks CVPR 2017
2. https://gluon.mxnet.io/chapter14_generative-adversarial-networks/dcgan.html

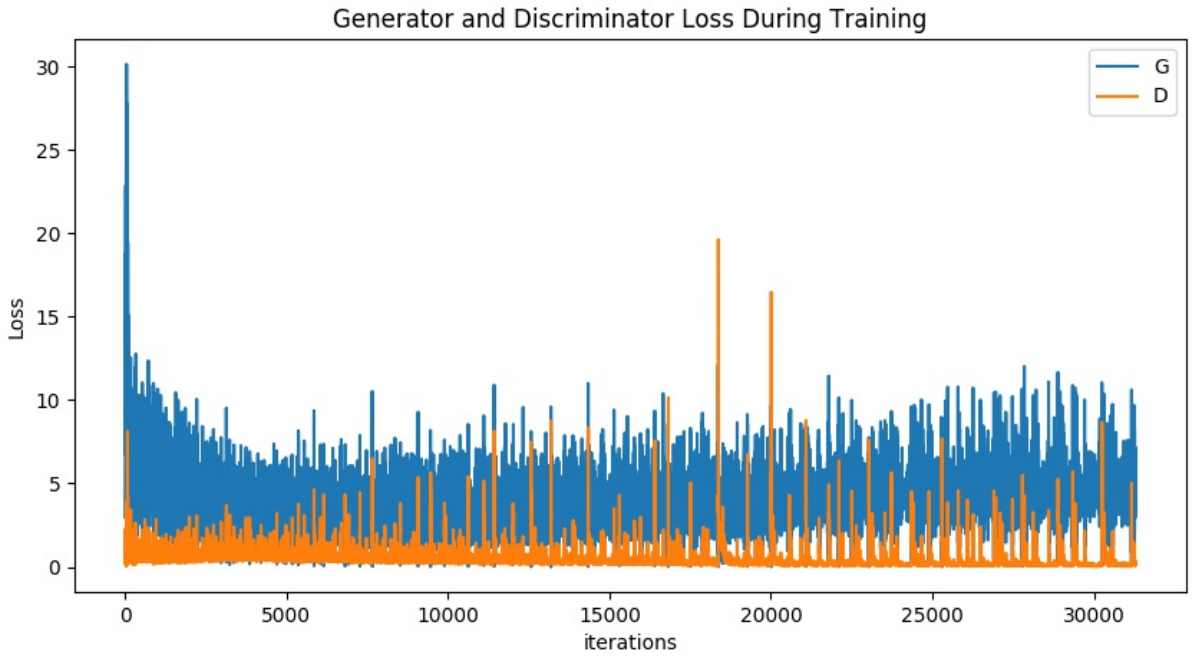
2. Plot 32 random images generated from your model. [fig1_2.jpg] (10%)



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

Here are some conclusions:

a. The generator of the DCGAN uses the transposed convolution technique to perform up-sampling of 2D image size.
b. GANS are difficult to train and tune.
c. One thing I did differently was that I used a different kernel weight initializer . It seems to have improved the quality of images but it is not clear by how much.
d. The generator's loss quantifies how well it was able to trick the discriminator. Intuitively, if the generator is performing well, the discriminator will classify the fake images as real. Here is my training loss diagram:



e. During training, the *generator* progressively becomes better at creating images that look real, while the *discriminator* becomes better at telling them apart. The process reaches equilibrium when the *discriminator* can no longer distinguish real images from fakes.

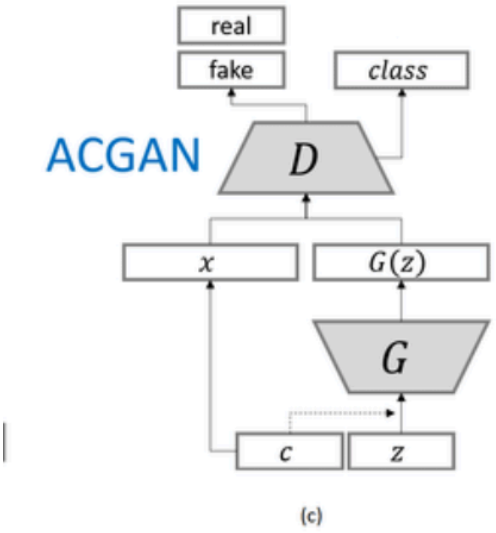
Problem 2: ACGAN (20%)

1. **Describe the architecture & implementation details of your model. (5%)**

   In the ACGAN, every generated sample has a corresponding class label, in addition to the noise z. G uses both to generate images X_{fake} = G(c, z). The discriminator gives both a probability distribution over sources and a probability distribution over the class labels, D(X) = P(S | X), P(C | X)

   Some parameters are:

   Batch size = 128, epochs = 100, Optimizer = Adam, learning rate = 0.0002



ACGAN architectures, where x denotes the real image, c the class label, z the noise vector, G the Generator, and D the Discriminator

```
Generator(
  (gen): Sequential(
    (0): ConvTranspose2d(101, 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)
    (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)
    (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)
    (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)
    (12): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (13): Tanh()
  )
)
Discriminator(
  (dis): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): LeakyReLU(negative_slope=0.2, inplace)
    (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)
    (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)
    (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)
  )
  (output): Sequential(
    (0): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): Sigmoid()
  )
  (classifier): Sequential(
    (0): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): Sigmoid()
  )
)
```

2. Plot 10 random pairs of generated images from your model, where each pair should be generated from the same random vector input but with opposite attribute. This is to demonstrate your model's ability to disentangle features of interest. [fig2_2.jpg] (10%)

First Row represents the smiling attribute of 10 randomly picked images whereas the second row represents different attribute of same 10 persons.

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

The objective function of ACGAN has two parts: the log-likelihood of the correct source, LS, and the log-likelihood of the correct class, L4C . D is trained to maximize LS + LC while G is trained to maximize LC − LS in terms of Architecture. For ACGAN, the input to the discriminator is an image, whilst the output is the probability that the image is real and its class label. One of the reasons why AC-GAN performs well on visual inspection and Inception Score is because these metrics reward the generator for sampling easily-recognized images uniformly across the image classes. If, however, the true distribution contains points that are fundamentally

difficult to classify because they lie near the true decision boundary, then AC-GAN will down-sample these points and thus learn a biased distribution! It shouldn't come as too much of a surprise that AC-GAN down-samples points near the decision boundary. After all, being near the decision boundary, pretty much by definition, means the image is hard to classify.

Reference

1. https://gluon.mxnet.io/chapter14_generative-adversarial-networks/dcgan.html

Problem 3: DANN (35%)

In this problem, you need to implement DANN and consider the following 2 scenarios:

(Source domain → Target domain) (1) MNIST-M → SVHN, (2) SVHN → MNIST-M

1. **Compute the accuracy on target domain, while the model is trained on source domain only. (lower bound) (3%)**

MNIST-M —> SVHN (source code in mnist_to_svhn.py)

Accuracy: 26.52%

For this I have saved the result in predict_mnistm_to_svhn.csv

SVHN —> MNIST-M (source code in svhn_mnist.py)

Accuracy : 42.01%

For this I have saved the result in predict_svhn_to_mnistm.csv

2. **Compute the accuracy on target domain, while the model is trained on source and target domain. (domain adaptation) (3+7%)**

Training file source code : domain_adaptation.py and domain_adaptation_2.py

MNIST-M —> SVHN

Accuracy: 46.40058389674247%

SVHN —> MNIST-M

Accuracy: 45.21%

3. **Compute the accuracy on target domain, while the model is trained on target domain only. (upper bound) (3%)**

   a. SVHN—> MNIST-M (source code file : mnist.py)

Model is trained on MNIST and tested on MNIST.

Accuracy: 91%

```
Test Accuracy of the model on the test images: 91
Test Accuracy of the model on the Class 0 : 95 %
Test Accuracy of the model on the Class 1 : 95 %
Test Accuracy of the model on the Class 2 : 92 %
Test Accuracy of the model on the Class 3 : 88 %
Test Accuracy of the model on the Class 4 : 87 %
Test Accuracy of the model on the Class 5 : 90 %
Test Accuracy of the model on the Class 6 : 94 %
Test Accuracy of the model on the Class 7 : 90 %
Test Accuracy of the model on the Class 8 : 88 %
Test Accuracy of the model on the Class 9 : 91 %
```

b. MNISTM —> SVHN (source code file :svhn.py)

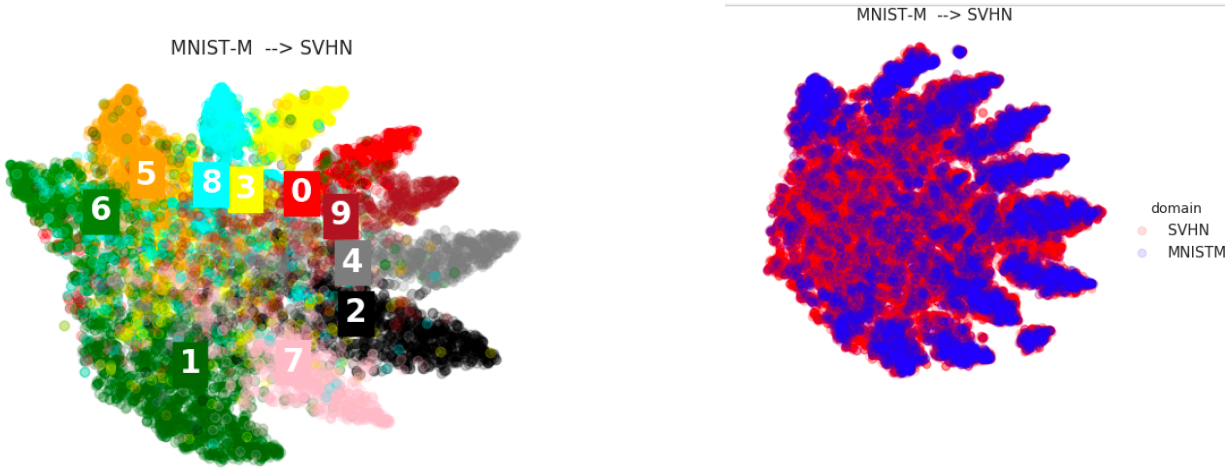
Accuracy: 90%

```
Test Accuracy of the model on the test images: 90 %
Test Accuracy of the model on the Class 0 : 90 %
Test Accuracy of the model on the Class 1 : 93 %
Test Accuracy of the model on the Class 2 : 92 %
Test Accuracy of the model on the Class 3 : 85 %
Test Accuracy of the model on the Class 4 : 93 %
Test Accuracy of the model on the Class 5 : 89 %
Test Accuracy of the model on the Class 6 : 89 %
Test Accuracy of the model on the Class 7 : 91 %
Test Accuracy of the model on the Class 8 : 83 %
Test Accuracy of the model on the Class 9 : 89 %
```

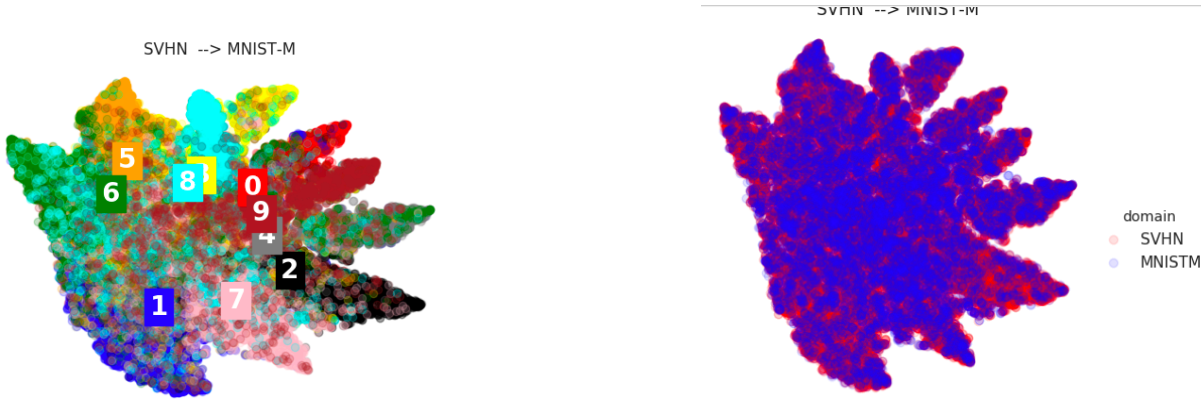
Model is trained on SVHN and tested on SVHN.

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). (6%)

MNIST-M —> SVHN

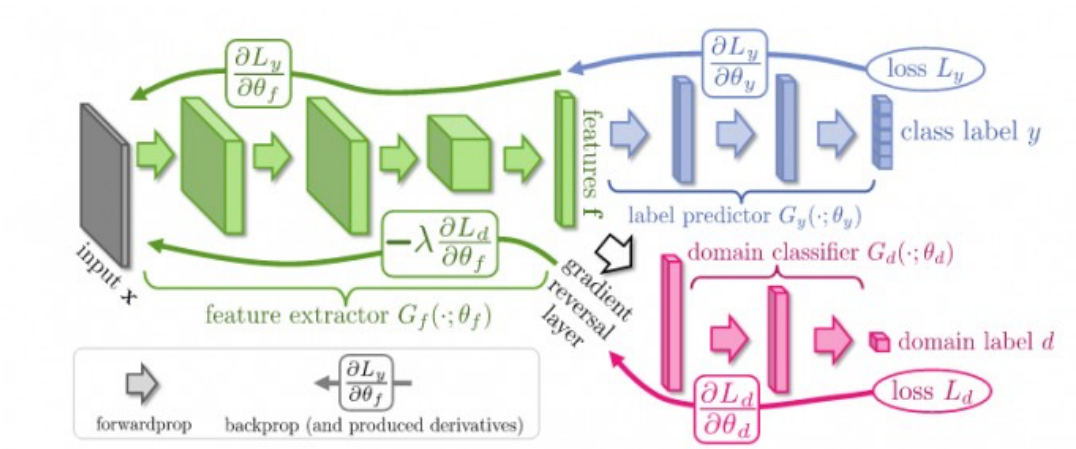


SVHN —> MNIST-M



5. Describe the architecture & implementation detail of your model. (6%)

I follow the the architecture of the paper : Unsupervised Domain Adaptation by Backpropagation.



Following are important parameters

Batch size: 128

Epoch : 50

Learning Rate : 1e-4

Optimizer: ADAM

I use only one CNN to use feature images from two different domains.

```
CNNModel(
  (feature): Sequential(
    (f_conv1): Conv2d(3, 64, kernel_size=(5, 5), stride=(1, 1))
    (f_bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (f_pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (f_relu1): ReLU(inplace=True)
    (f_conv2): Conv2d(64, 50, kernel_size=(5, 5), stride=(1, 1))
    (f_bn2): BatchNorm2d(50, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (f_drop1): Dropout2d(p=0.5, inplace=False)
    (f_pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (f_relu2): ReLU(inplace=True)
  )
  (class_classifier): Sequential(
    (c_fc1): Linear(in_features=800, out_features=100, bias=True)
    (c_bn1): BatchNorm1d(100, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (c_relu1): ReLU(inplace=True)
    (c_drop1): Dropout2d(p=0.5, inplace=False)
    (c_fc2): Linear(in_features=100, out_features=100, bias=True)
    (c_bn2): BatchNorm1d(100, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (c_relu2): ReLU(inplace=True)
    (c_fc3): Linear(in_features=100, out_features=10, bias=True)
    (c_softmax): LogSoftmax()
  )
  (domain_classifier): Sequential(
    (d_fc1): Linear(in_features=800, out_features=100, bias=True)
    (d_bn1): BatchNorm1d(100, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (d_relu1): ReLU(inplace=True)
    (d_fc2): Linear(in_features=100, out_features=2, bias=True)
    (d_softmax): LogSoftmax()
  )
)
```

6. Discuss what you've observed and learned from implementing DANN. (7%)

DANN uses only one CNN to get features of images from two different domains. Some of the key observations are:

a. For effective domain transfer to be achieved, predictions must be made based on features that cannot discriminate between the training (source) and test (target) domains.

b. Unsupervised domain adaptation is achieved by adding a domain classifier (red) connected to the feature extractor via a gradient reversal layer that multiplies the gradient by a certain negative constant during the backpropagation-based training.

c. Also for SVHN —> MNIST-M case, there is less difference in accuracy after domain transfer as compared to source training.

Expected Results on Report:

	SVHN —> MNIST-M	MNIST-M —> SVHN
Trained on Source	42.01%	26.52%
Adaptation (DANN)	45.21%	46.400%
Trained on Target	91%	90%

Reference :

1. https://github.com/fungtion/DANN

2. https://github.com/NaJaeMin92/pytorch_DANN

Grading – Problem 4

Improved UDA model (35%)

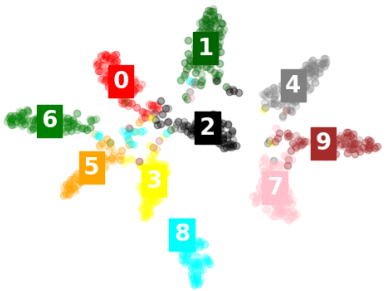
1. **Compute the accuracy on target domain, while the model is trained on source and target domain. (domain adaptation) (6+10%)**

Source code for training in

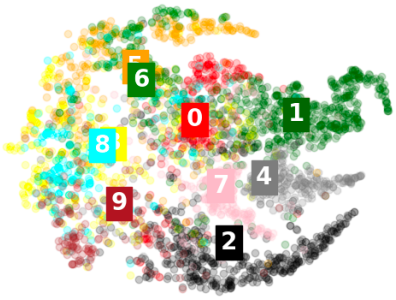
	SVHN —> MNIST-M	MNIST-M —> SVHN
Adaptation (Improved)	56.03%	61.274%

2. **Visualize the the latent space by mapping the *testing* images to 2D space (with t-SNE) and use different colors to indicate data of (*a*) different digits classes 0-9 and (*b*) different domains (source/target). (6%)**

MNISTM —> SVHN

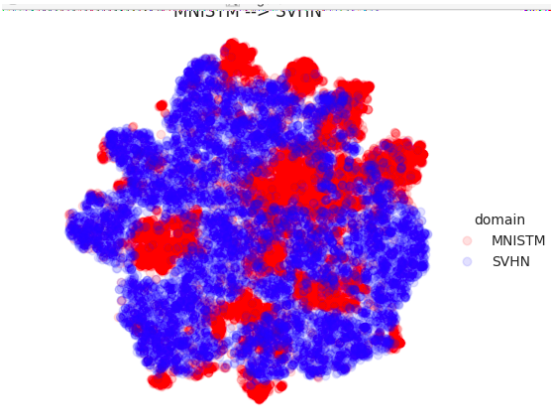


SVHN —>MNIST

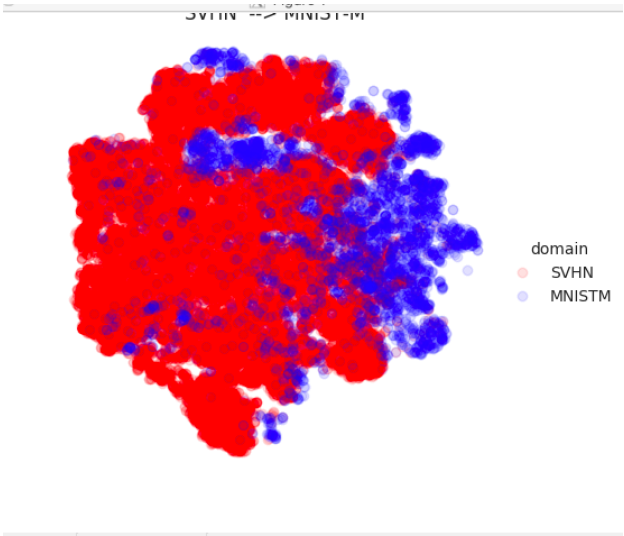


Different Domain:

MNISTM—> SVHN



SVHN —> MNISTM



3. **Describe the architecture & implementation detail of your model. (6%)**

The Structure of Encoder (Extractor) is:

```
class Extractor(nn.Module):
    def __init__(self):
        super(Extractor, self).__init__()
        self.conv1 = nn.Conv2d(1, 20 , 5)
        self.bn1 = nn.BatchNorm2d(20)
        self.conv2 = nn.Conv2d(20, 50, 5)
        self.bn2 = nn.BatchNorm2d(50)
        self.fc1 = nn.Linear(50 * 4 * 4, 500)
        self.bn3  = nn.BatchNorm1d(500)

    def forward(self, x):
        out = F.leaky_relu(self.bn1(self.conv1(x)))
        out = F.max_pool2d(out, 2)
        out = F.leaky_relu(self.bn2(self.conv2(out)))
        out = F.max_pool2d(out, 2)
        out = out.view(out.size(0), -1)
        out = F.leaky_relu(self.bn3(self.fc1(out)))
        return out
```

The Structure of Classifier is:

```
class Classifier(nn.Module):
    def __init__(self):
        super(Classifier, self).__init__()
        self.fc2 = nn.Linear(500, 10)

    def forward(self, x):
        return self.fc2(x)
```

The Structure of Discriminator is:

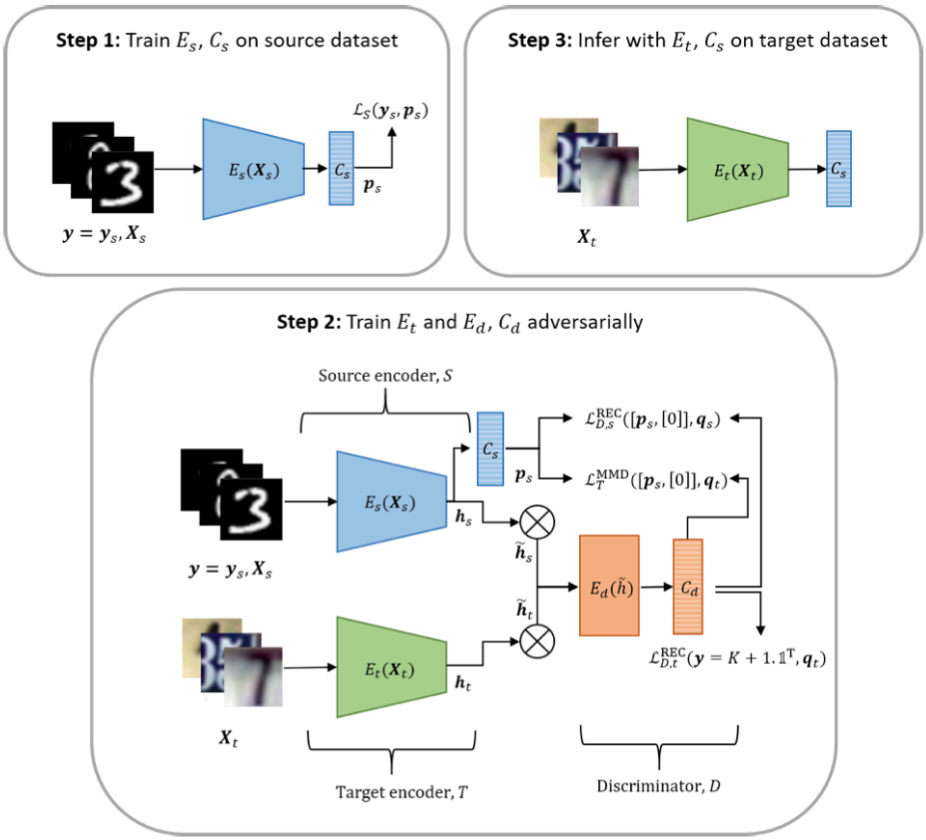
```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.fc1 = nn.Linear(500, numberHiddenUnitsD)
        self.fc2 = nn.Linear(numberHiddenUnitsD, numberHiddenUnitsD)
        self.fc3 = nn.Linear(numberHiddenUnitsD, 2)
        self.bn1 = nn.BatchNorm1d(numberHiddenUnitsD)
        self.bn2 = nn.BatchNorm1d(numberHiddenUnitsD)

    def forward(self, x):
        out = F.leaky_relu(self.bn1(self.fc1(x)))
        out = F.leaky_relu(self.bn2(self.fc2(out)))
        return self.fc3(out)
```

4. **Discuss what you've observed and learned from implementing your improved UDA model. (7%)**

Key Learnings and observations:

a. Adversarial discriminative domain adaptation (ADDA) is an efficient framework for unsupervised domain adaptation in image classification, where the source and target domains are assumed to have the same classes, but no labels are available for the target domain.

b.



This above diagram very effectively elaborates the working of ADDA. Initially we train Encoder and Classifier on Source dataset and then we train encoder on source and target adversarially.

c. This above approach is more reasonable as there are some unique features on both source and target.

D. The approach works reasonably good for MNIST-M —> SVHN. May be its due to unique features of source which performs better on target.

Reference:

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

Collaborators:

1. A08945201
2. T08902301