Course: Deep Learning for Computer Vision 3 December 2019
Name: Zeynep Cankara
Student Id: T08902113

Homework 03

Problem 1:

Part 1.1:

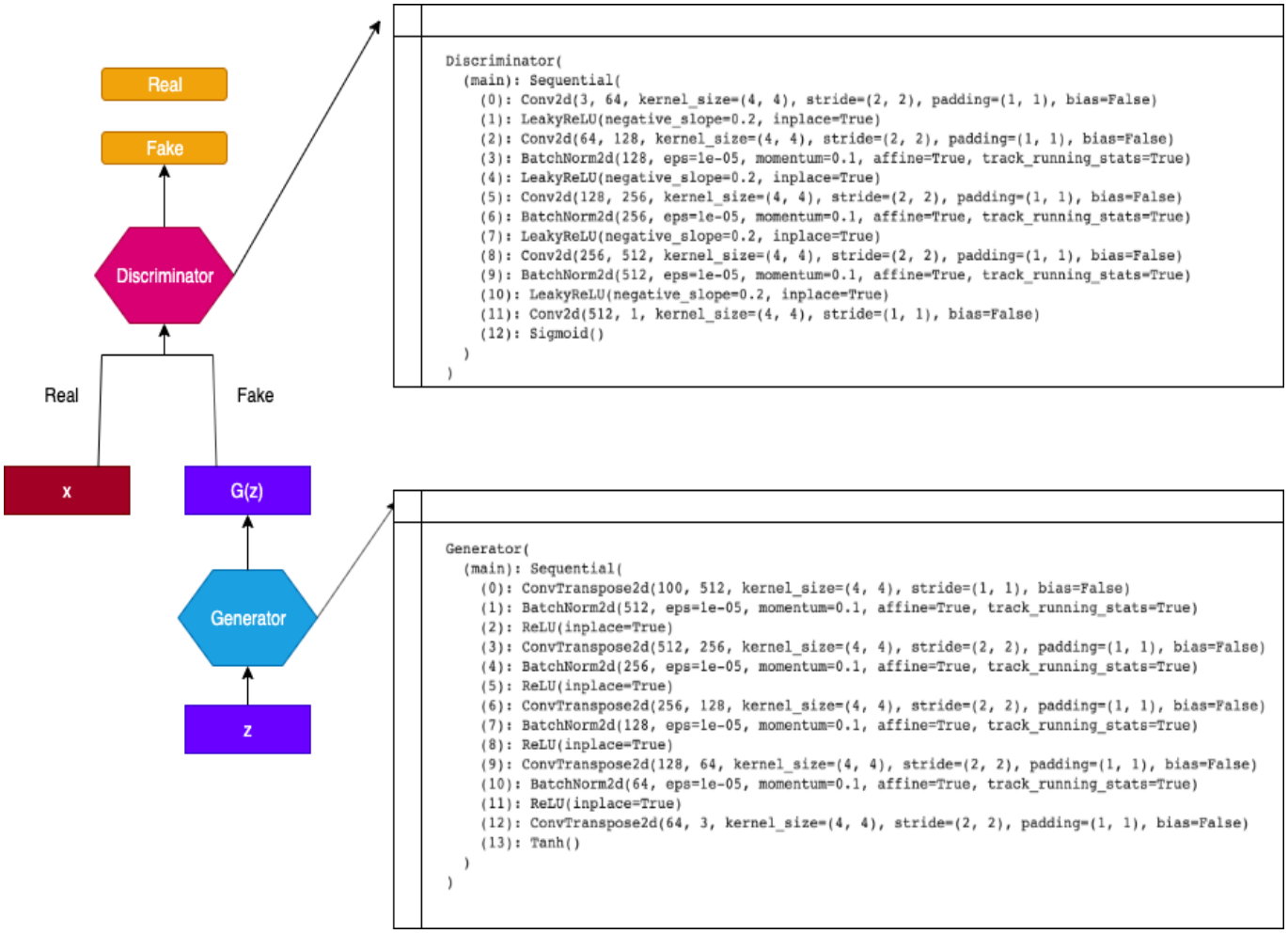


Figure 1: Architectural details of the GAN model

Implementation details of the GAN model:

- The model consists of 2 main modules. One of them is the Discriminator module and the other one is the Generator module. The Generator tries to fool the generator by generating images from the latent space vector and the Discriminator tries to guess whether the image is fake or real.
- Conversion of the latent space vector z into RGB image achieved via the generator module which consists series of strided 2D convolutional transpose layers, each paired with a 2D batch norm layers and a Relu activation.
- I used strided convolutions rather than the pooling layers to do down-sampling.

- Batch Norm and Leaky Relu being used to achieve healthy gradient flow.

Part 1.2

Figure 2: 32 Random Images Generated by the GAN Model

Part 1.3: Observations from implementing a GAN model:

- GAN's use the training data's distribution to generate new images from the distribution it learns.
- Replacing fully connected layers in the traditional GAN architectures with the convolutional and convolutional-transpose layers makes it easier to train the GAN model.
- Equilibrium point of optimization of the GAN is when the Discriminator makes random guess (50%) about whether the data is real or coming from the generator.
- Using the Adam optimizer and Relu functions together with the Batch Norm layers helped to increased the stability of the model.
- Generator loss going up sometimes does not mean model over-fitting and instead of halting the training letting the model training with more epochs gives better results.

Problem 2:

Part 2.1:

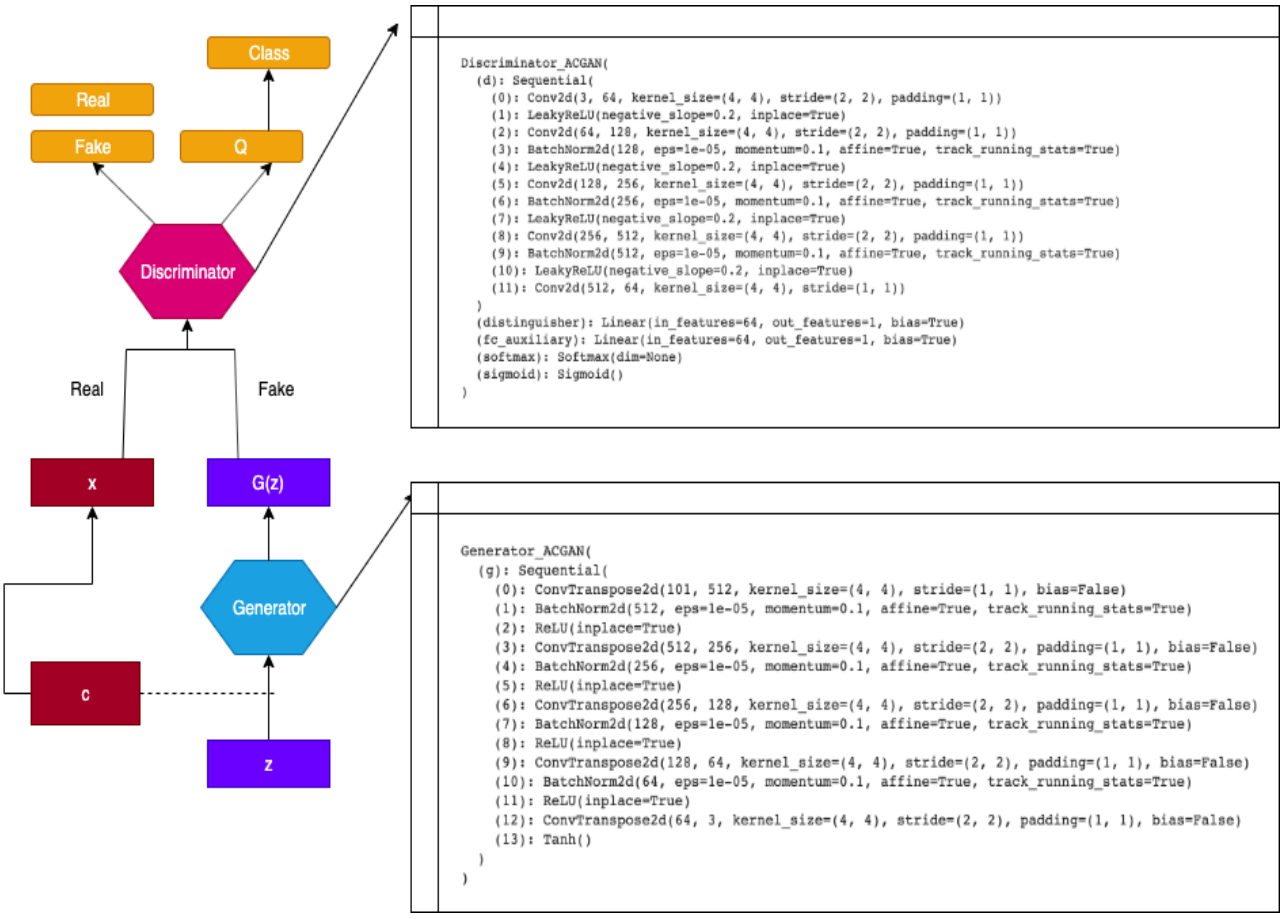


Figure 3: Architectural details of the ACGAN model

Implementation details of the ACGAN model:

- The model consists of 2 main modules. One of them is the Discriminator module and the other one is the Generator module. Differing from the GAN architecture ACGAN has the ability of generating an image with a specific attribute. On top of that it can differentiate whether an image is fake or real via the Discriminator module.
- I used the Adam optimizer with a learning rate of 0.0002. For parameters beta1 and beta2, I used 0.5 and 0.999 respectively. For the loss function I used Binary Cross-Entropy loss.
- I used strided convolutions rather than the pooling layers to do down-sampling.
- Batch Norm and Leaky Relu being used to achieve healthy gradient flow.
- Trained the model with 100 epochs with batch size of 64.

Part 2.2:

Figure 4: 10 Random Image Pairs, feature disentanglement attribute 'Smiling'

Part 2.3: Observations from implementing an ACGAN model:

- Changing learning rate within every 40 epochs helped the model to converge faster. I trained the model with 100 epochs.
- Replacing fully connected layers in the traditional ACGAN architectures with the convolutional and convolutional-transpose layers makes it easier to train the AC-GAN model.
- Using the Adam optimizer and Relu functions together with the Batch Norm layers helped to increased the stability of the model.
- The model is able to generate the image with the specific attribute in this case the smiling attribute.

Problem 3:

Part 3.1 and 3.2 and 3.3:

Source –> Target	SVHN –> MNISTM	MNISTM –> SVHN
Trained on source	38.43%	38.20%
Adaptation (DANN)	42.03%	40.38%
Trained on Target	96.57%	88.70%

Figure 5: Accuracy of the DANN model

Part 3.4:

The figures are the latent space visualization of the test dataset via t-sne using the DANN model. The blue color represents the source domain and the red color represents the target domain. The class information labelled on the data points.

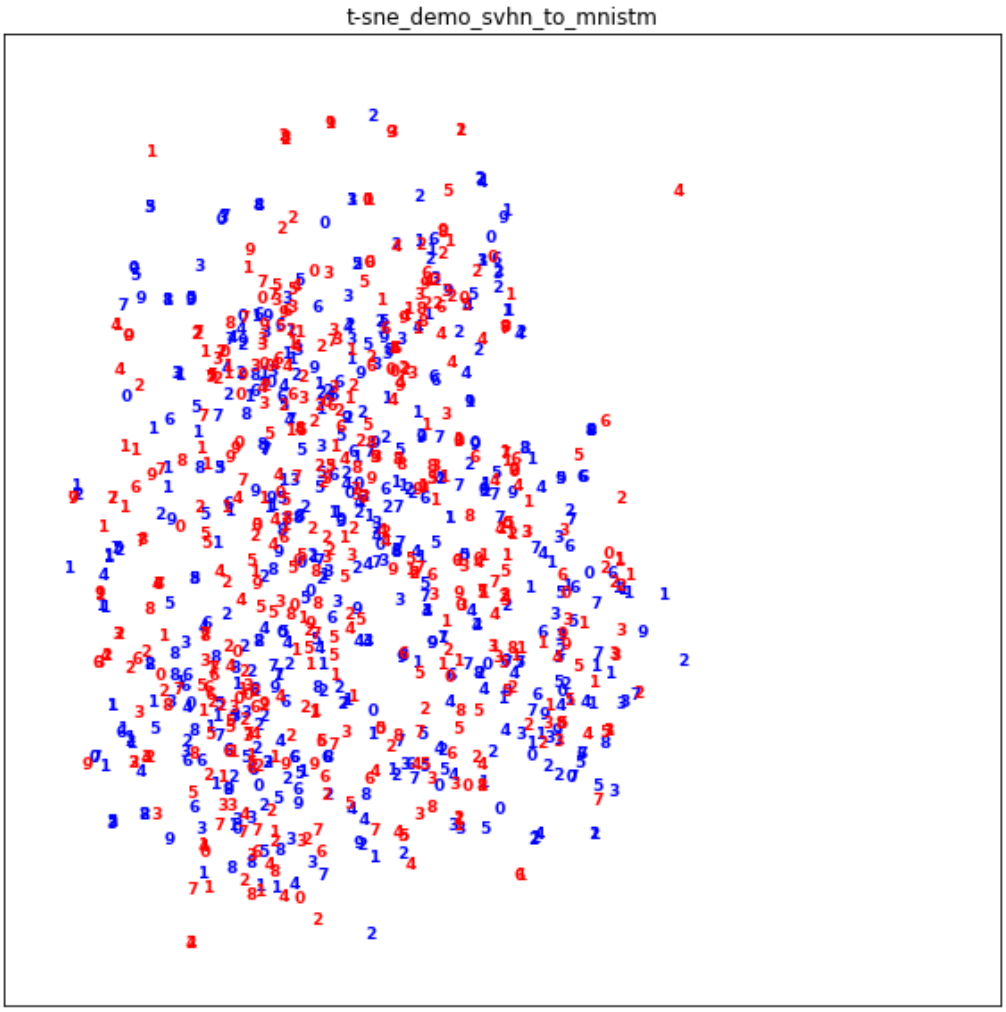


Figure 6: t-SNE latent space: source SVHN, target MNISTM

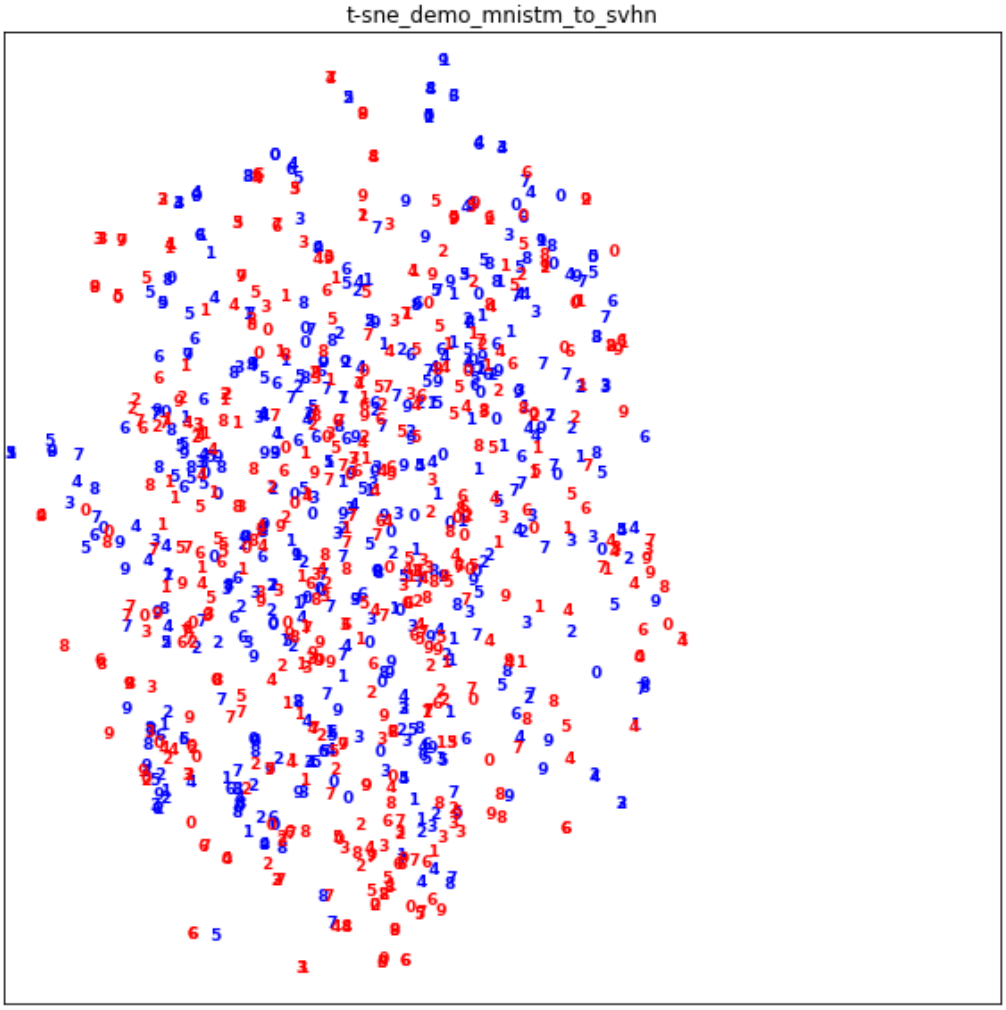


Figure 7: t-SNE latent space: source MNISTM, target SVHN

Part 3.5:

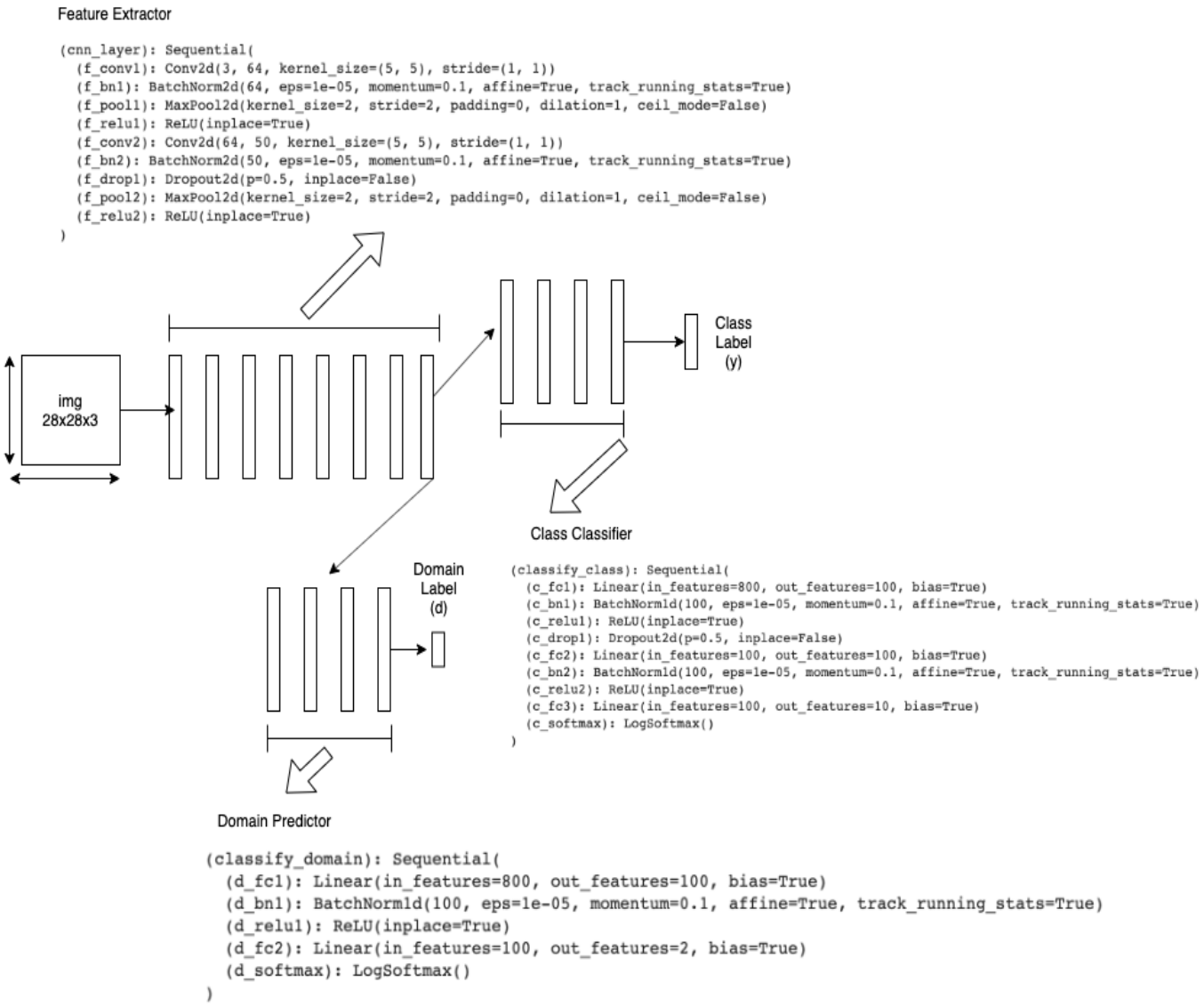


Figure 8: Architectural details of the DANN model

Implementation details of the DANN model:

- The model is a single CNN module which consists of a CNN backbone as a feature extractor and 2 output pipeline. One of the output pipeline is for classifying the domain which the data comes from and the other one is for predicting the class which data belongs to.
- Softmax function being used at the end of the domain and class classifier pipelines.
- I used strided convolutions together with a Max-pool layer in the feature extractor part of the DANN model.
- Batch Norm and Leaky Relu being used to achieve healthy gradient flow.
- Negative log likelihood loss being used for both the domain predictor and the class classifier part of the DANN model.
- Model trained with 20 epochs with batch size of 128. Initial learning rate of 0.001. Even with the small epoch size convergence like behaviour observed.

Part 3.6: Observations from implementing a DANN model:

- Model trained with 10 epochs surpassed the model trained with 20 epochs in the case of domain adaptation. The model trained with 20 epochs showed over-fitting behaviour.
- The model able to achieve higher accuracy when trained with the 'SVHN' dataset and take 'MNISTM' as the target. The reason might be the number of images in 'SVHN' is much more than the 'MNISTM'dataset and maybe the model learning more generalisable data distribution.

Problem 4:

Part 4.1

Source –> Target	SVHN –> MNISTM	MNISTM –> SVHN
Adaptation (UDA)	53.20%	56.22%

Figure 9: Accuracy of the Improved UDA model

Part 4.2:

The figures are the latent space visualization of the test dataset via t-sne using the improved UDA model. The blue color represents the source domain and the red color represents the target domain. The class information labelled on the data points.

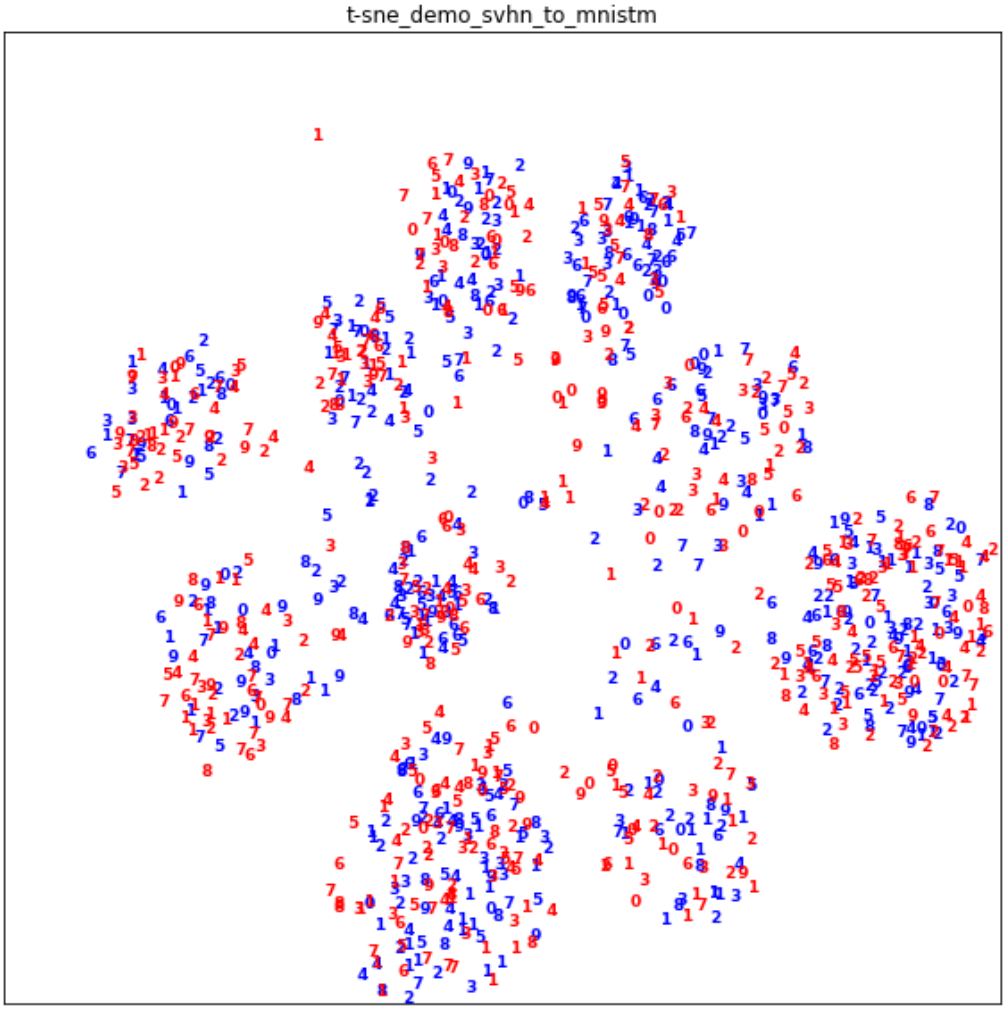


Figure 10: t-SNE latent space: source SVHN, target MNISTM

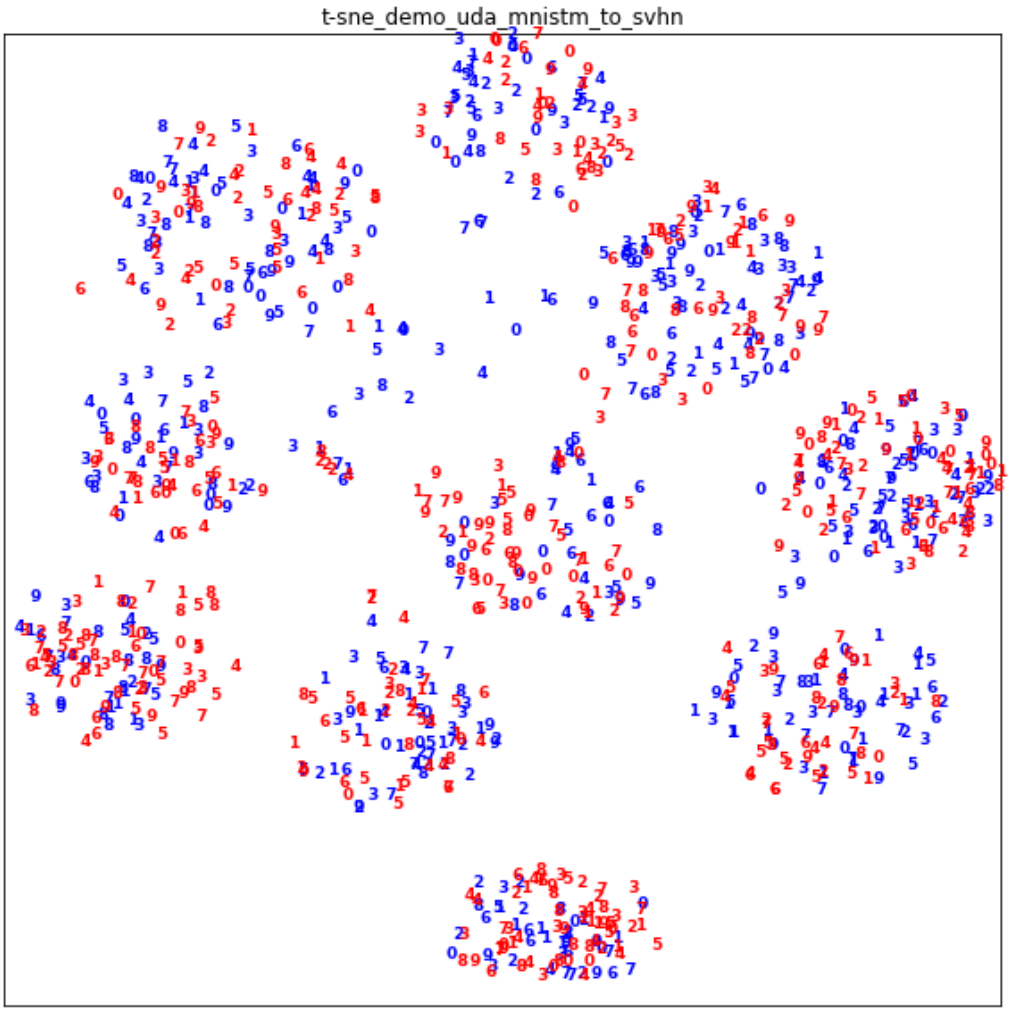
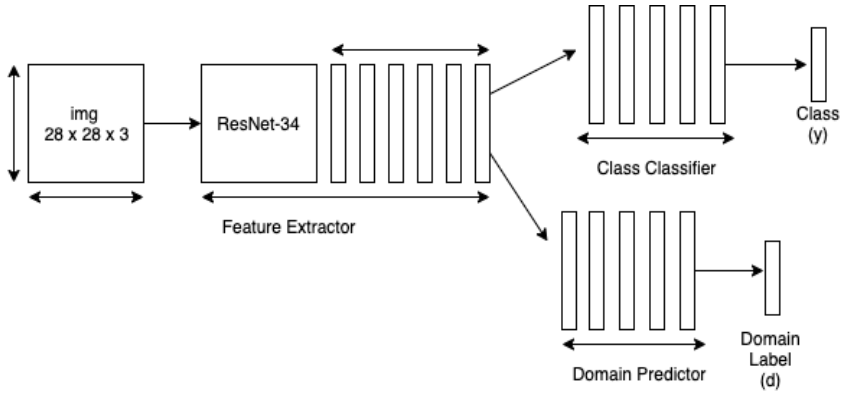


Figure 11: t-SNE latent space: source MNISTM, target SVHN

Part 4.3:



Feature Extractor

```
    )
  )
  (r_convt1): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (r_relu1): ReLU(inplace=True)
  (r_convt2): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (r_relu2): ReLU(inplace=True)
  (r_convt3): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (r_relu3): ReLU(inplace=True)
)
```

Domain Predictor

```
(classify_domain): 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()
)
```

Class Classifier

```
(classify_class): 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()
)
```

Figure 12: Architectural details of the improved UDA model

Implementation details of the improved UDA model:

- The model consists of Resnet34 backbone pre-trained with imagenet weights together with a CNN feature extractor head and 2 output pipeline. One of the output pipeline is for classifying the domain which the data comes from and the other one is for predicting the class which data belongs to.
- Softmax function being used at the end of the domain and class classifier pipelines.
- I used strided convolutions together with a Max-pool layer in the feature extractor part of the improved UDA model.
- Batch Norm and Leaky Relu being used to achieve healthy gradient flow.
- Negative log likelihood loss being used for both the domain predictor and the class classifier part of the improved UDA model.

Part 4.4: Observations from implementing a improved UDA model:

- Using a deeper and more sophisticated feature extractor module helped improve classical DANN architecture drastically.
- I freezed the ResNet34 parameters initially which can be un-freezed and re trained again to allow ResNet backend learn from the gradients of the data. Fine-tuning the model.
- After 3 epochs lowering the learning rate from 0.0002 to 0.00002 helped the model to learn faster.
- After 20 epochs over-fitting observed.

References and Collaborators
I used Google Colab's GPU: Tesla K80

1. https://pytorch.org/tutorials/beginner/dcgan-faces-tutorial.html
2. https://github.com/znxlwm/pytorch-generative-model-collections
3. https://github.com/fungtion/DANN-py3
4. https://github.com/thtang/ADLxMLDS2017/
5. https://github.com/NaJaeMin92/pytorch-DANN/blob/master/utils.py
6. https://arxiv.org/abs/1610.09585