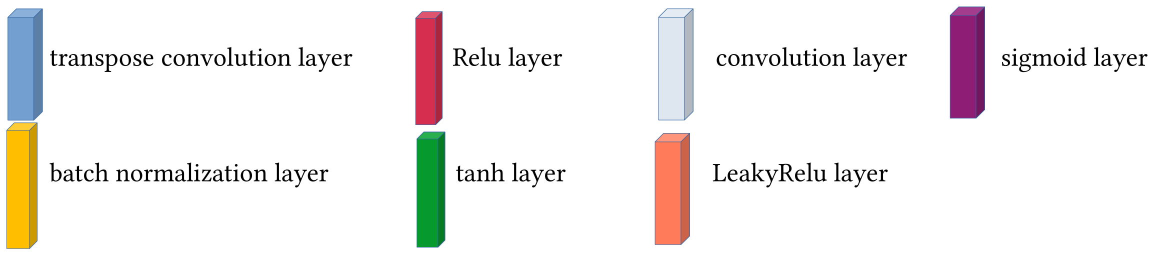
Clautour Amélie A08 527 201

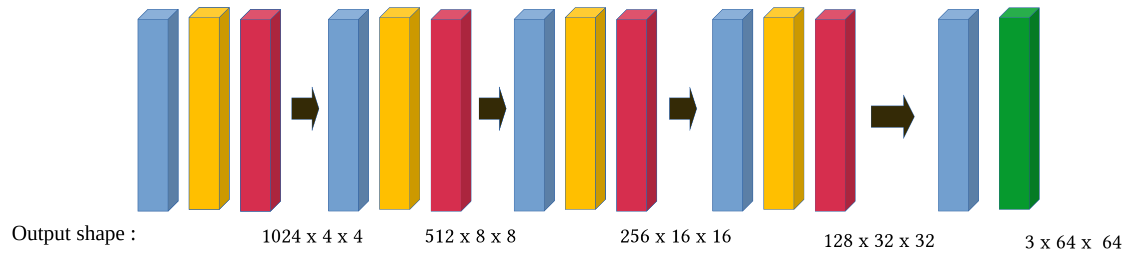
DLCV Homework 3

Problem 1: GAN

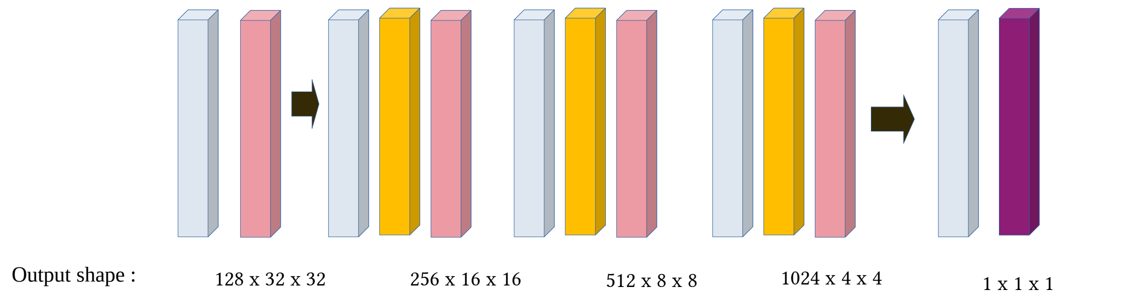
1. Describe the architecture & implementation details of your model.

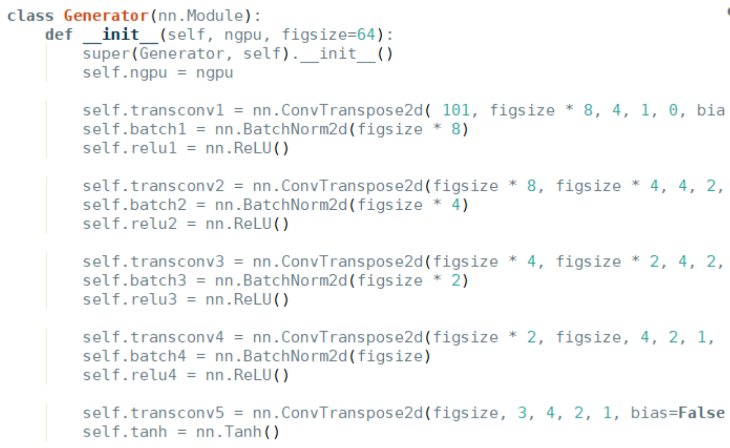


Generator :

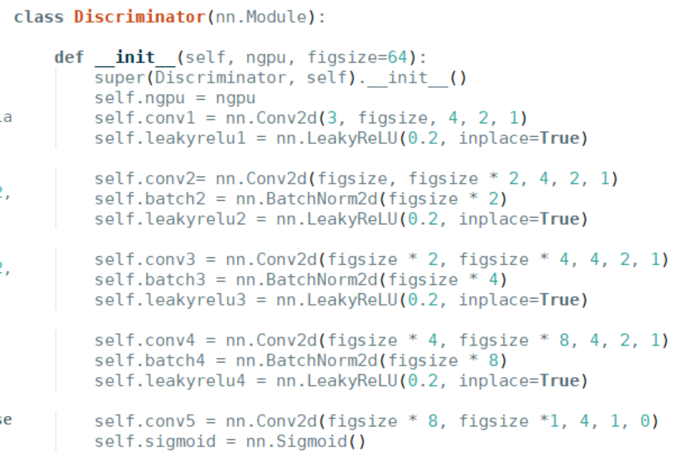


Discriminator :





```python
class Generator(nn.Module):
    def __init__(self, ngpu, figsize=64):
        super(Generator, self).__init__()
        self.ngpu = ngpu

        self.transconv1 = nn.ConvTranspose2d( 101, figsize * 8, 4, 1, 0, bia
        self.batch1 = nn.BatchNorm2d(figsize * 8)
        self.relu1 = nn.ReLU()

        self.transconv2 = nn.ConvTranspose2d(figsize * 8, figsize * 4, 4, 2,
        self.batch2 = nn.BatchNorm2d(figsize * 4)
        self.relu2 = nn.ReLU()

        self.transconv3 = nn.ConvTranspose2d(figsize * 4, figsize * 2, 4, 2,
        self.batch3 = nn.BatchNorm2d(figsize * 2)
        self.relu3 = nn.ReLU()

        self.transconv4 = nn.ConvTranspose2d(figsize * 2, figsize, 4, 2, 1,
        self.batch4 = nn.BatchNorm2d(figsize)
        self.relu4 = nn.ReLU()

        self.transconv5 = nn.ConvTranspose2d(figsize, 3, 4, 2, 1, bias=False
        self.tanh = nn.Tanh()
```



```python
class Discriminator(nn.Module):

    def __init__(self, ngpu, figsize=64):
        super(Discriminator, self).__init__()
        self.ngpu = ngpu
        self.conv1 = nn.Conv2d(3, figsize, 4, 2, 1)
        self.leakyrelu1 = nn.LeakyReLU(0.2, inplace=True)

        self.conv2= nn.Conv2d(figsize, figsize * 2, 4, 2, 1)
        self.batch2 = nn.BatchNorm2d(figsize * 2)
        self.leakyrelu2 = nn.LeakyReLU(0.2, inplace=True)

        self.conv3 = nn.Conv2d(figsize * 2, figsize * 4, 4, 2, 1)
        self.batch3 = nn.BatchNorm2d(figsize * 4)
        self.leakyrelu3 = nn.LeakyReLU(0.2, inplace=True)

        self.conv4 = nn.Conv2d(figsize * 4, figsize * 8, 4, 2, 1)
        self.batch4 = nn.BatchNorm2d(figsize * 8)
        self.leakyrelu4 = nn.LeakyReLU(0.2, inplace=True)

        self.conv5 = nn.Conv2d(figsize * 8, figsize *1, 4, 1, 0)
        self.sigmoid = nn.Sigmoid()
```

Implementation details :

batch size = 128
latent dimension = 101
learning rate = 0.0002
beta1 = 0.5
number of epochs = 130

loss function = BCEloss
optimizers = AdamOptimizer for both generator and discriminator
normalized images with : mean = 0.5 std = 0.5
latent vectors = from normal distribution (mean = 0 , std = 1)

2. Plot 32 random images generated from your model. [fig1_2.jpg]

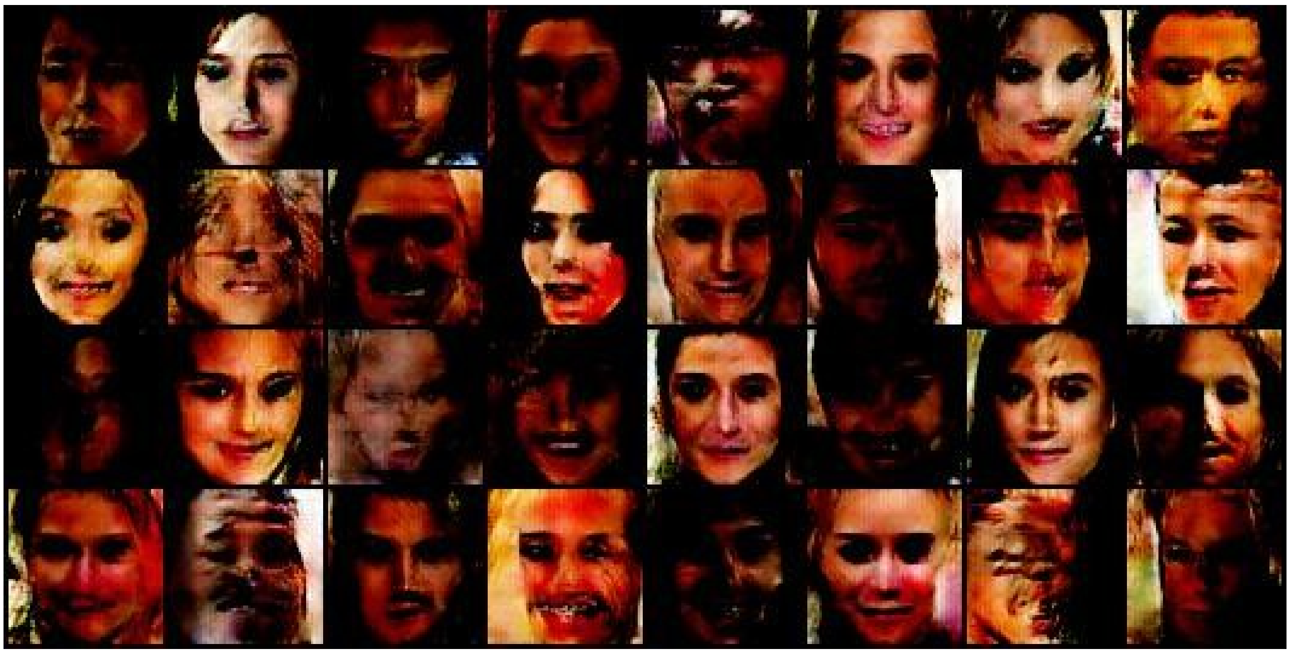


fig1_2.jpg

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

The discriminator loss drops a lot more faster and to a lower score than the generator : it can indeed drop under 0.02 while the generator loss is still around 0.1. But when the discriminator loss is very low the generator doesn't produce as good quality image as when the two losses are more even.

Also I noted that for two models with roughly the same generator loss but different discriminator loss, the lower the discriminator loss the better the output images whereas for two models with the same discriminator loss but different generator losses there wasn't a significant difference of quality between outputs. Thus we can think that the discriminator impacts the quality more than the generator.

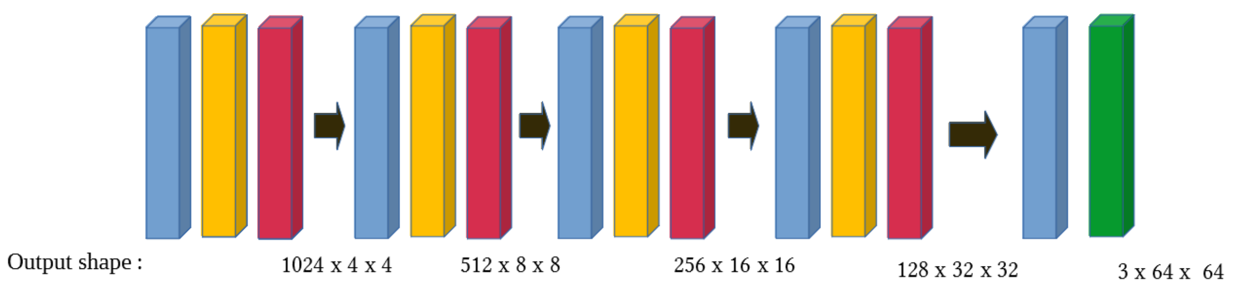
Another point is that the quality of the generated faces is also dependent on the randomly created input noise.

Problem 2: ACGAN

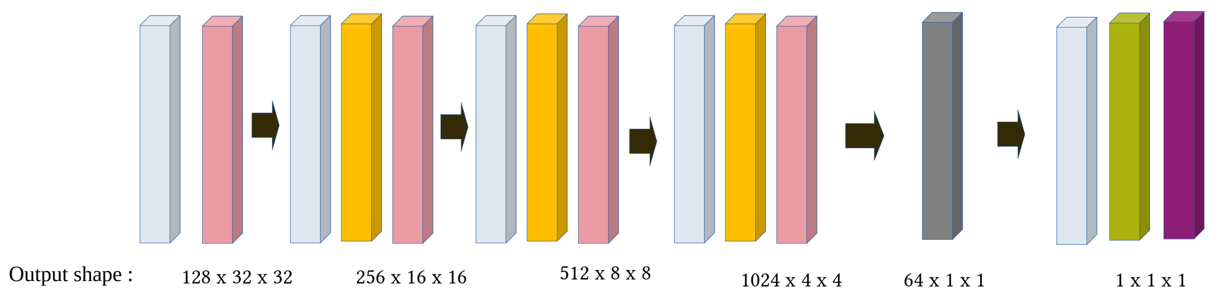
1. Describe the architecture & implementation details of your model.

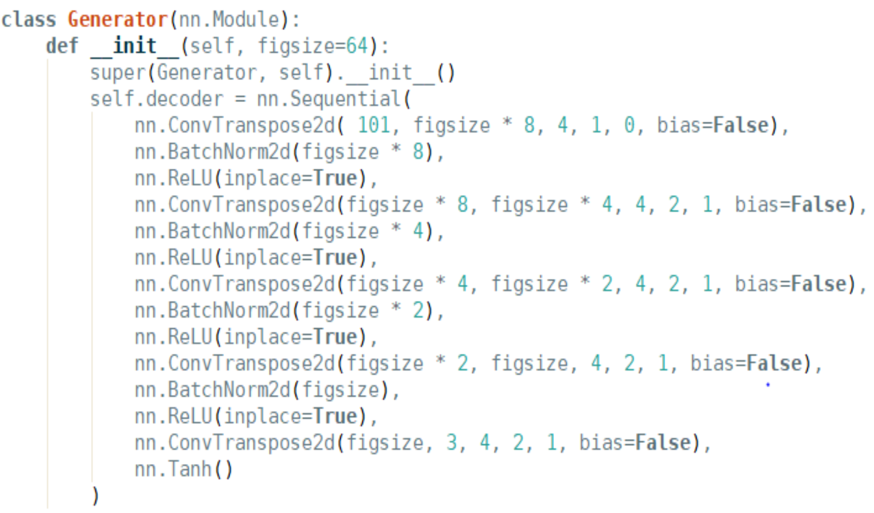
🟦 transpose convolution layer
🟥 Relu layer
⬜ convolution layer
🟪 sigmoid layer
⬛ linear layer
🟨 batch normalization layer
🟩 tanh layer
🟧 LeakyRelu layer
🟩 softmax layer

Generator :

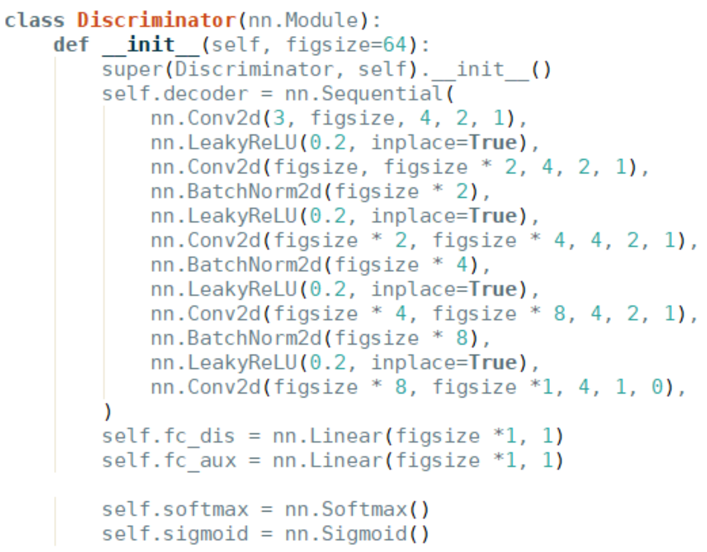


Discriminator :





```
class Generator(nn.Module):
    def __init__(self, figsize=64):
        super(Generator, self).__init__()
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d( 101, figsize * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(figsize * 8),
            nn.ReLU(inplace=True),
            nn.ConvTranspose2d(figsize * 8, figsize * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(figsize * 4),
            nn.ReLU(inplace=True),
            nn.ConvTranspose2d(figsize * 4, figsize * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(figsize * 2),
            nn.ReLU(inplace=True),
            nn.ConvTranspose2d(figsize * 2, figsize, 4, 2, 1, bias=False),
            nn.BatchNorm2d(figsize),
            nn.ReLU(inplace=True),
            nn.ConvTranspose2d(figsize, 3, 4, 2, 1, bias=False),
            nn.Tanh()
        )
```



```
class Discriminator(nn.Module):
    def __init__(self, figsize=64):
        super(Discriminator, self).__init__()
        self.decoder = nn.Sequential(
            nn.Conv2d(3, figsize, 4, 2, 1),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(figsize, figsize * 2, 4, 2, 1),
            nn.BatchNorm2d(figsize * 2),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(figsize * 2, figsize * 4, 4, 2, 1),
            nn.BatchNorm2d(figsize * 4),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(figsize * 4, figsize * 8, 4, 2, 1),
            nn.BatchNorm2d(figsize * 8),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(figsize * 8, figsize *1, 4, 1, 0),
        )
        self.fc_dis = nn.Linear(figsize *1, 1)
        self.fc_aux = nn.Linear(figsize *1, 1)

        self.softmax = nn.Softmax()
        self.sigmoid = nn.Sigmoid()
```

Implementation details :

batch size = 64	loss function = BCEloss
latent dimension = 101	optimizers = AdamOptimizer for both generator and discriminator
learning rate = 0.0002	normalized images with : mean = 0.5 std = 0.5
beta1 = 0.5	latent vectors = from normal distribution (mean = 0 , std = 1)
number of epochs = 200	

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]

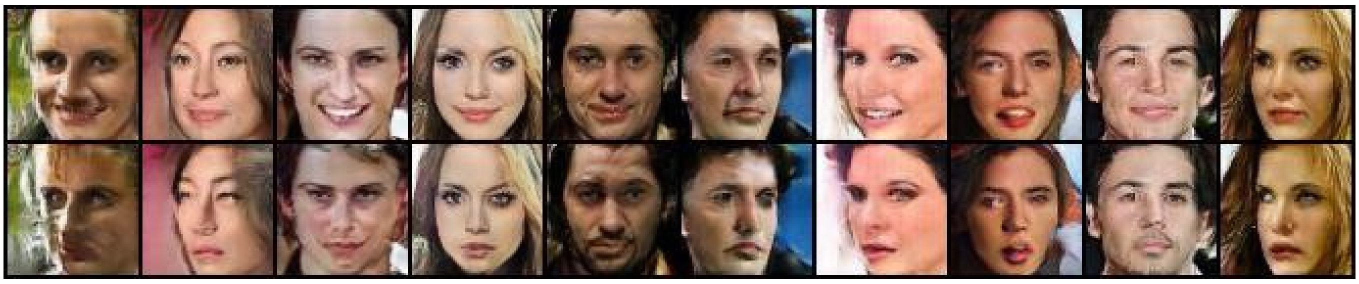


fig2_2.jpg

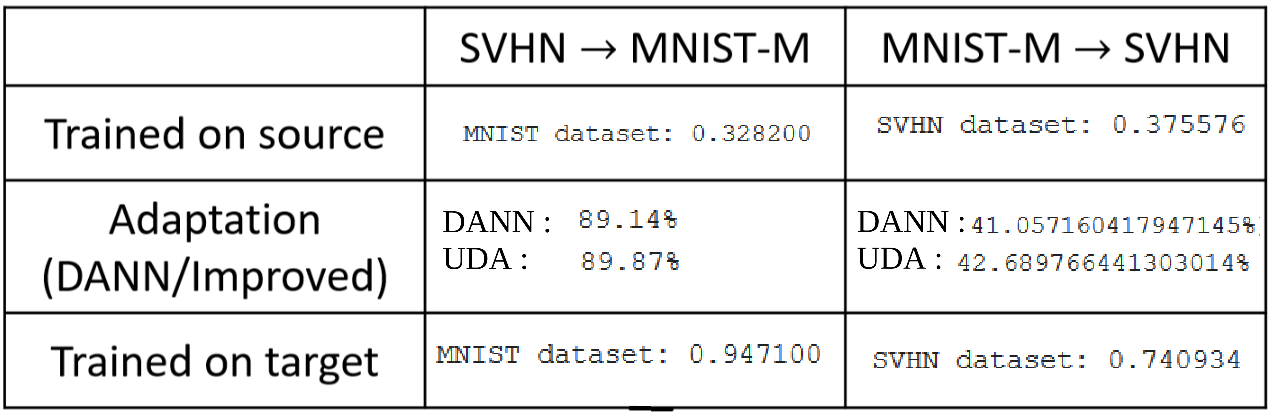
3. Discuss what you've observed and learned from implementing ACGAN.

The generator of ACGAN is the same as the generator of problem1 GAN and the discriminators are also almost the same except than in ACGAN it returns two informations (data and attribute) but the quality of the output images is a lot more greater in this problem which I can only explain by the fact that the training of ACGAN lasted five times longer than the training of GAN.

It is also noticeable that the mouth on which is applied (or not) the "smiling" attribute is often the part of the face that is the most well-designed and realistic. The generator therefore might perform better with constraint because it's not only constraint but also further information to generate from.

Problem 3: DANN

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

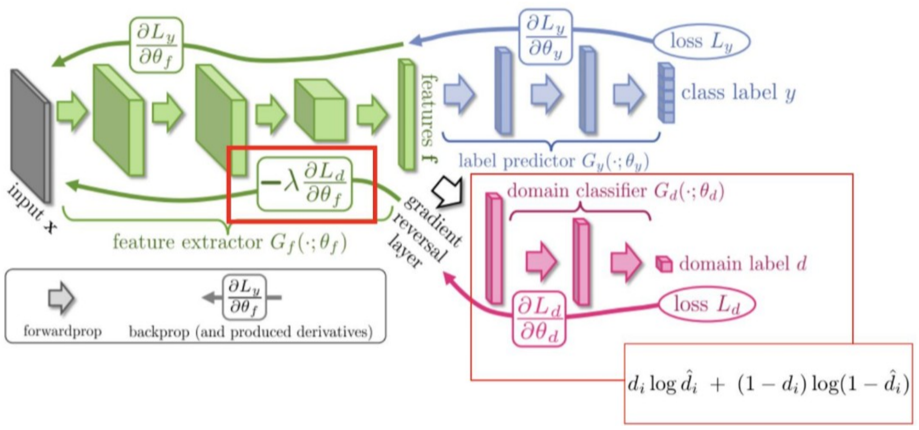


	SVHN → MNIST-M	MNIST-M → SVHN
Trained on source	`MNIST dataset: 0.328200`	`SVHN dataset: 0.375576`
Adaptation (DANN/Improved)	DANN : `89.14%` UDA : `89.87%`	DANN : `41.057160417947145%` UDA : `42.689766441303014%`
Trained on target	`MNIST dataset: 0.947100`	`SVHN dataset: 0.740934`

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).

Not done.

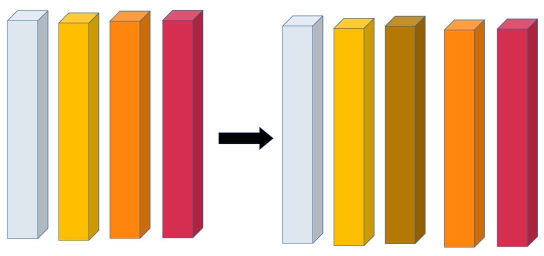
5. Describe the architecture & implementation detail of your model.



$\frac{\partial L_y}{\partial \theta_f}$
$\frac{\partial L_y}{\partial \theta_y}$
loss L_y
class label y
features $\mathbf{f}$
label predictor $G_y(\cdot;\theta_y)$
input $\mathbf{x}$
$-\lambda \frac{\partial L_d}{\partial \theta_f}$
gradient reversal layer
feature extractor $G_f(\cdot;\theta_f)$
domain classifier $G_d(\cdot;\theta_d)$
domain label d
loss L_d
$\frac{\partial L_d}{\partial \theta_d}$
forwardprop
backprop (and produced derivatives)
$d_i \log \hat{d}_i + (1-d_i)\log(1-\hat{d}_i)$

🟧 batch normalization layer
🟧 Maxpool layer
⬜ convolution layer
⬛ linear layer
🟨 batch normalization 1D layer
🟫 Dropout layer
🟥 Relu layer
🟩 Logsoftmax layer

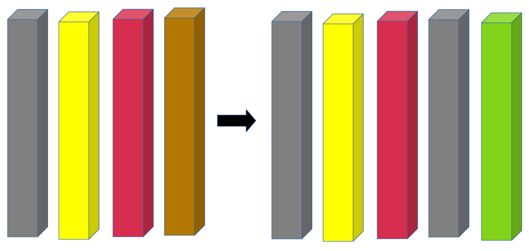
Feature extractor :



Output shape : 64 x 13 x 13 128 x 5 x5

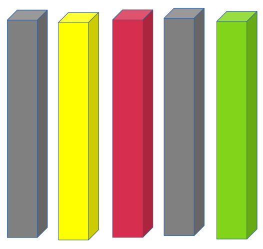
```
class DANN_Neural_Network(nn.Module):

    def __init__(self):
        super(DANN_Neural_Network, self).__init__()
        self.attr = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=5),
            nn.BatchNorm2d(64),
            nn.MaxPool2d(2),
            nn.ReLU(True),
            nn.Conv2d(64, 50, kernel_size=5),
            nn.BatchNorm2d(50),
            nn.Dropout2d(),
            nn.MaxPool2d(2),
            nn.ReLU(True))

        self.cclass = nn.Sequential(
            nn.Linear(50 * 4 * 4, 100),
            nn.BatchNorm1d(100),
            nn.ReLU(True),
            nn.Dropout2d(),
            nn.Linear(100, 100),
            nn.BatchNorm1d(100),
            nn.ReLU(True),
            nn.Linear(100, 10),
            nn.LogSoftmax())

        self.domain = nn.Sequential(
            nn.Linear(50 * 4 * 4, 100),
            nn.BatchNorm1d(100),
            nn.ReLU(True),
            nn.Linear(100, 2),
            nn.LogSoftmax(dim=1))
```

Label predictor :



Output shape : 100 10

Domain classifier :



Output shape : 2

<u>Implementation details :</u>

batch size = 128
learning rate = 0.0001
beta1 = 0.5
optimizer = AdamOptimizer
loss function = NLLLoss
normalized images with : mean = 0.5 std = 0.5
number of epochs = 100

6. Discuss what you've observed and learned from implementing DANN.

When the loss of the domain classifier was more than two times the loss of the feature extractor it made the loss of the class classifier rise: so when the domain classifier was outperformed by the feature extractor it then led to wrong label classifications. Therefore the capacity of the feature extractor, class classifier and domain classifier should be around the same level during the training.

The accuracy is very different whether we are in (1) svhn to mnist or (2) mnist to svhn. I think it may because the ratio between test and training are also very different whether we take the mnist dataset (1/6) or the svhn dataset (1/4) so that models are relatively more trained on mnist. It's especially true for the adaptation model which performs a lot more on the mnist test dataset.

Problem 4: Improved UDA model

I implemented the ADDA model.

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

On SVHN test dataset :

```
42.6897664413030148%
```

On MNIST test dataset :

```
89.87%
```

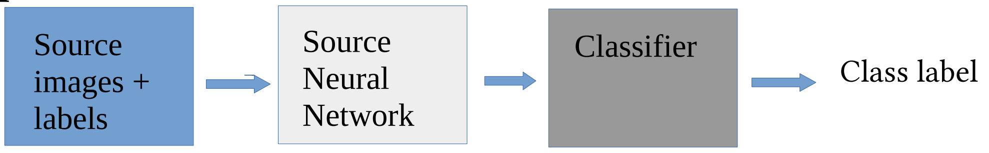
2. Visualize the the latent space by mapping the testing images to 2D space (with t-SNE) and use different colors to indicate data of (a) different digits classes 0-9 and (b) different domains (source/target).

Not done.

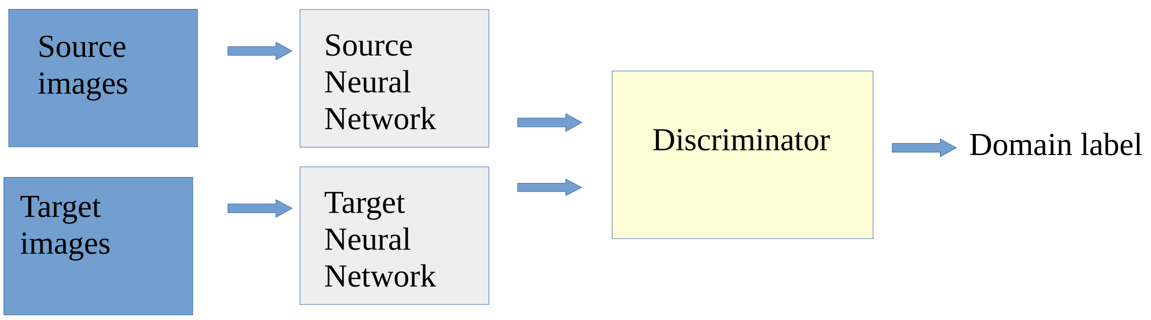
3. Describe the architecture & implementation detail of your model.

ADDA concept:

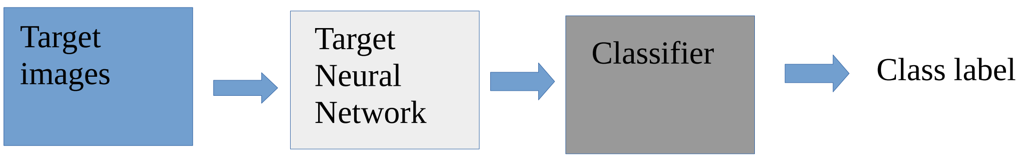
Pre-training :

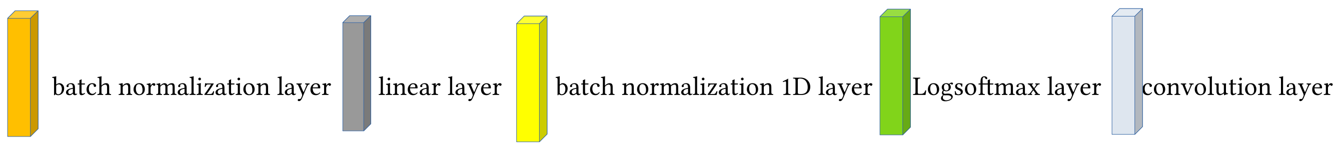


Adversarial Adaptation :

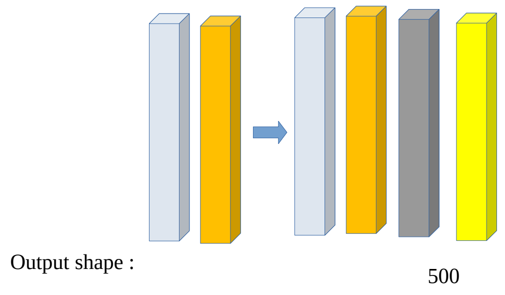


Testing :



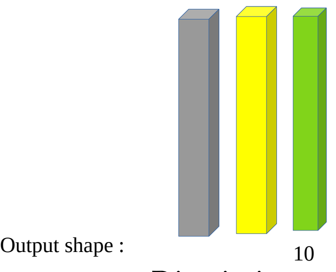


Neural Network (source and target)



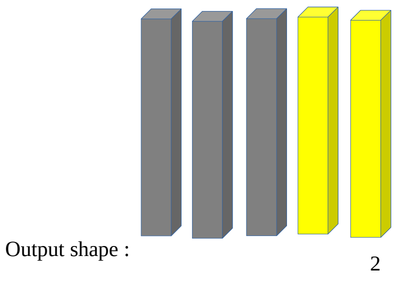
```
class UDA_NeuralNetwork(nn.Module):
    def __init__(self):
        super(UDA_NeuralNetwork, self).__init__()
        self.conv1 = nn.Conv2d(1, 20 , 5)
        self.bn1 = nn.BatchNorm2d(20)
        self.conv2 = nn.Conv2d(20, 50, 5)
        self.bn2 = nn.BatchNorm2d(50)
        self.fc1 = nn.Linear(50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

Classifier



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

Discriminator



```
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)
```

Implementation details :

batch size = 256	loss function = CrossEntropyLoss
learning rate = 0.0002	normalized images with : mean = 0.5 std = 0.5
beta1 = 0.5	optimizer = AdamOptimizer for discriminator and targetNN

4. Discuss what you've observed and learned from implementing your improved UDA model.

The ADDA model was more difficult to train than the DANN model, it's more likely due to the fact that there is two different feature extractors involved in the process but the accuracy results aren't significantly much better than the results of DANN considering the training was a lot longer.
Also we can make the same remarks as for the DANN model in regards to the difference of performance on the mnist test dataset and on the svhn test dataset.

References :
https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html

https://stephan-osterburg.gitbook.io/coding/coding/ml-dl/tensorfow/chapter-4-conditional-generative-adversarial-network/implementation-of-acgan-model

https://github.com/corenel/pytorch-adda/tree/master/models

No collaborator