DLCV Homework 3

張緣彩 R07922141

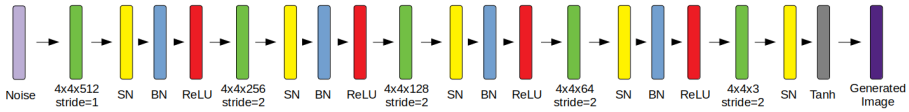
2/12/2019

Collaborators

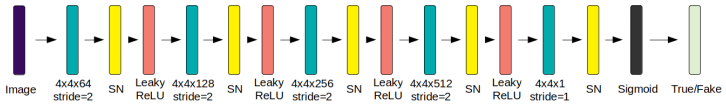
1. 林子淵 R07921100
2. 黃孟霖 R07922170
3. 楊之郡 R08922050

Problem 1. GAN

Q1. Describe the architecture and implementation details of your model.



(a) Model architecture of Generator.



(b) Model architecture of Discriminator.

Figure 1: GAN model architecture. I used *ConvTranspose2d* in Generator and *Conv2d* Discriminator with parameters showed above.

For the DCGAN model (GAN), I referred to Pytorch DCGAN tutorial[1] and make some improvement on the architecture which recommends by these websites[2, 3]. From the figures above, I added spectral normalization layer[4] for the generate and discriminator. However, I removed didn't add batch normalization layer[5] for the discriminator.

For training this model, since predict true or fake is a binary classification problem, so I used binary cross-entropy loss (BCELoss) which is a build-in loss function in Pytorch. I calculate and update discriminator weight before generator in every iteration. In order to generate fake image, I random sample noise from normal distribution with build-in function. Then forward the noises to generator to generate fake images and detach it to let discriminator to predict whether the generated images is fake or not. Moreover, I also pass real images to discriminator and calculate bce loss to update discriminator weight. After

the discriminator was trained in the iteration, I pass the same fake images to discriminator again without detaching because we need the gradient back-propagate to the generator in order to train the generator weight.

Q2. Plot 32 random images generated from your model.

Figure 2: 32 random generated images.

Q3. Discuss what you've observed and learned from implementing GAN.

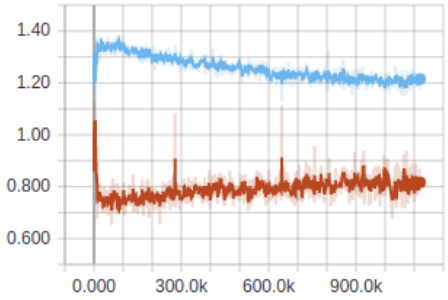


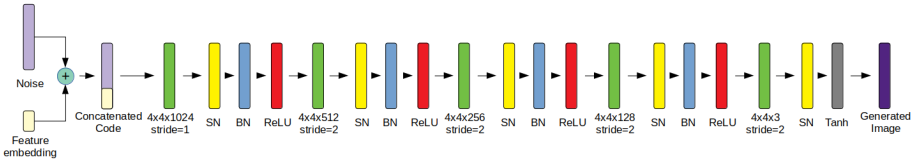
Figure 3: Loss of generator (red) and discriminator (blue).

DCGAN is really unstable in training. But I found that if spectral normalization is added to both generator and discriminator, model will be more stable. And by removing batch normalization in discriminator and add convolution layer's bias will make training easier, generated image will be prettier. In my training experiences, I tried to make discriminator less powerful than generator will have

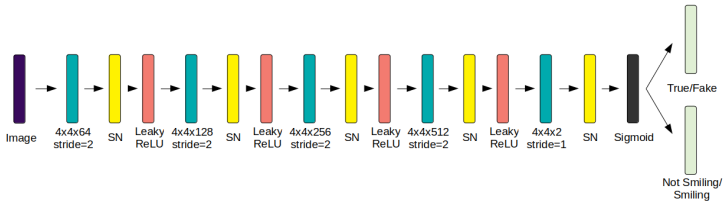
better result at the end. From the figure 3, we can see that loss of generator is lower than the discriminator, it will increase over time and discriminator loss will decrease in the meantime. At the end, I think the loss will converge at the same spot around 1.0 (which ACGAN's loss converge here, showed in figure 6). I trained 1000 epochs for DCGAN and ACGAN, and ACGAN's loss converge around 1.0 but DCGAN's loss didn't. I think because using label as auxiliary to train GAN makes it easier.

Problem 2. ACGAN

Q1. Describe the architecture and implementation details of your model.



(a) Model architecture of Generator.



(b) Model architecture of Discriminator.

Figure 4: ACGAN model architecture. I used *ConvTranspose2d* and an Embedding Layer in Generator and *Conv2d* Discriminator with parameters showed above.

The ACGAN model is same as DCGAN model with one more embedding layer added. I used embedding layer to extract and store smiling and not smiling features. The dimension of embedding layer is 4. The noise dimension I used 120 instead of 128 in the DCGAN model. The training procedure same as training DCGAN, but generator need to learn to generate image with noises and label(smiling/not smiling) and discriminator need to verify real/fake images and which label(smiling/not smiling) they belongs to.

For the generator, I pass random sampled noises and labels into the model. The model will first get the label's embedded features from the embedding layer

and concatenate it with the noises and forward it to the convolution blocks to generate images.

For the discriminator, I pass real and fake images into the model. The model need to verify whether these images are real or fake and whether these images are labeled as smiling or not.

Q2. 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.

Figure 5: 10 random pairs of generated images. Not smiling at left side and smiling at right side.

Q3. Discuss what you've observed and learned from implementing ACGAN.

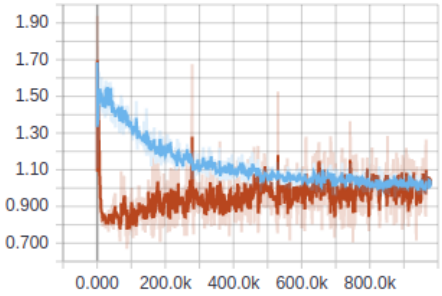


Figure 6: Loss of generator (red) and discriminator (blue).

Since ACGAN have labels acts as an auxiliary to help model converge faster and more stable. From figure 6, we can see that generator's loss and discriminator's loss converge around 1.0 in about 1000 epochs in ACGAN while DCGAN need to take more epochs than 1000 to converge into around 1.0 (see figure 3).

I also tried to use all the labels, but it became harder to train, generated images were uglier. So I took a look on the label, the distribution of some label were unbalance. After that, I used 'Smiling' and 'Male' to retrain the model, the training was very successful, the generated images were much more realistic. But I didn't use that model for this problem because I don't whether we were allowed to use others label than 'Smiling' or not.

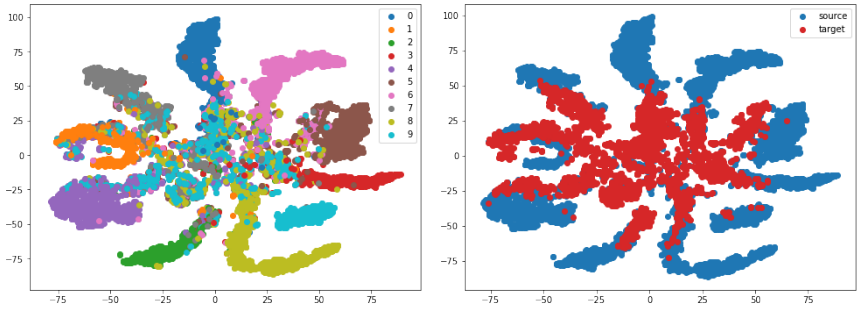
Problem 3. DANN

Q1. Accuracy on target domain in different scenarios.

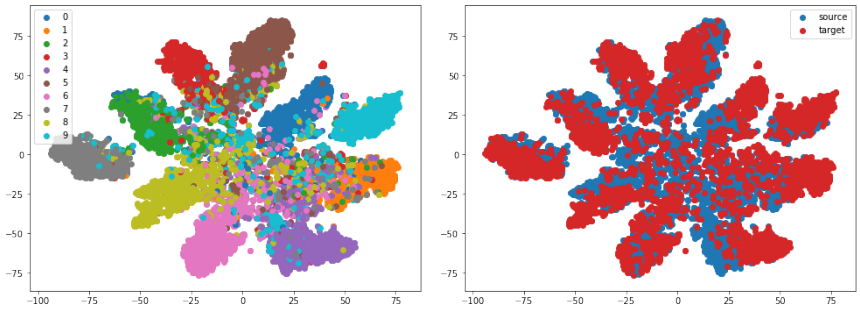
	MNIST-M > SVHN	**SVHN > MNIST-M**
Trained on source	42.27489	50.29
Adaptation (DANN)	47.94868	68.83
Trained on target	92.62062	97.60

Table 1: Accuracy on target domain in 2 scenarios(source -> target): MNIST-M -> SVHN and SVHN -> MNIST-M.

Q2. 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).



(a) MNIST-M -> SVHN. Left figure is different digit classes 0-9. Right figure is different domains.



(b) SVHN -> MNIST-M. Left figure is different digit classes 0-9. Right figure is different domains.

Figure 7: Visualize the latent space by mapping the testing images to 2D space with t-SNE. DANN algorithm used for domain adaptation. 1000 random features in each class sampled from source and target.

Q3. Describe the architecture and implementation detail of your model.

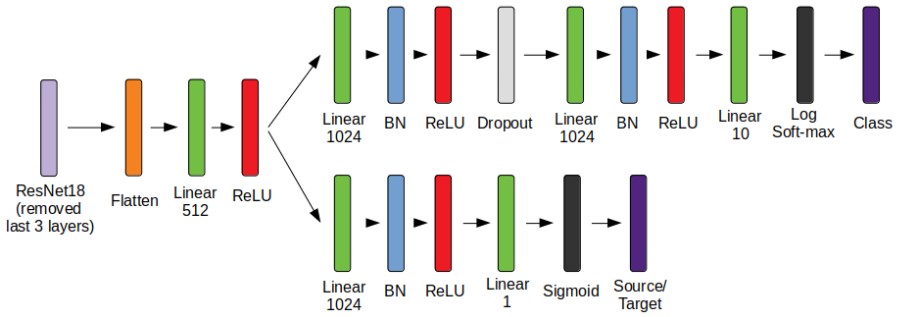


Figure 8: Model architecture of DANN. ResNet18 used as feature extractor of DANN and followed by label predictor and domain classifier.

For DANN model architecture, I referred to the paper[6] and this GitHub[7]. I implemented the model described inside the paper, but I can't reproduce the accuracy as good as the paper. So I decided to change to a more powerful feature extractor, ResNet18. The accuracy boosted up, but still not as good as the paper stated. After the image's features extracted by the feature extractor, it will pass to label predictor and domain classifier. Before passing to domain classifier, it will go thought a Gradient Reversal Layer (GRL). The GRL back-propagated negative gradient multiplied with a lambda (constant) to feature extractor during back-propagation.

For training procedure, I zipped source-dataset and target-dataset together if the target-dataset is given, else it will only train on source-dataset (update model weight with label loss only). So my implementation able to train on source only (upper/lower bound model) or to train domain adaptation model.

The loss for label prediction are Cross entropy loss since there are multi-class and Binary cross entropy loss for domain classification since we only need to classifier whether the features is from source or target.

Q4. Discuss what you've observed and learned from implementing DANN.

Training Unsupervised Domain Adaptation (UDA) models are really unstable. Because implement the same model with same hyper-parameters still can't reproduce the result showed in the paper. Moreover, it's very sensitive to data. It's easier to get better accuracy when adapting SVHN's domain into MNIST-M's domain than adapting MNIST-M's domain into SVHN's domain.

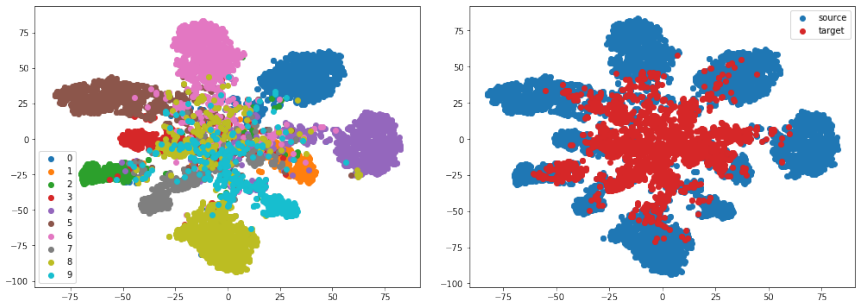
Problem 4. Improved UDA model.

Q1. Accuracy on target domain in different scenarios.

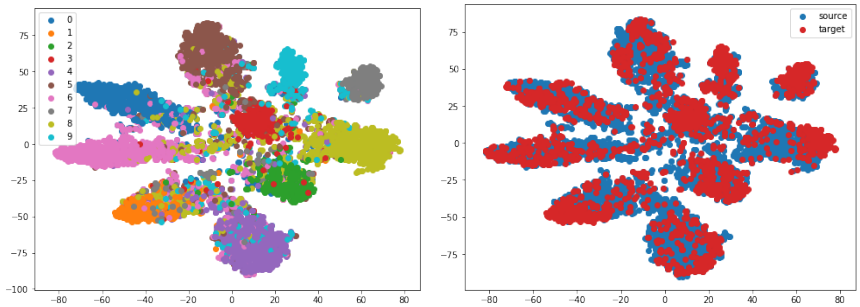
	MNIST-M > SVHN	SVHN > MNIST-M
Trained on source	35.46789	48.09
Adaptation (DSN)	52.58912	71.69
Trained on target	92.70513	97.60

Table 2: Accuracy on target domain in 2 scenarios(source -> target): MNIST-M -> SVHN and SVHN -> MNIST-M.

Q2. 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).



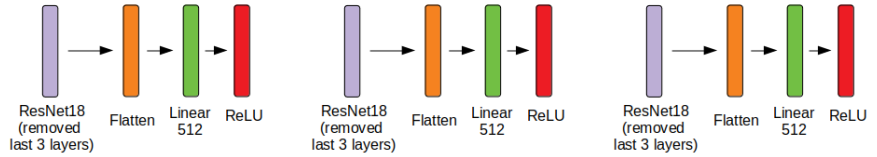
(a) MNIST-M -> SVHN. Left figure is different digit classes 0-9. Right figure is different domains.



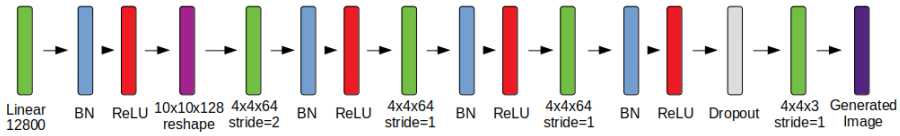
(b) SVHN -> MNIST-M. Left figure is different digit classes 0-9. Right figure is different domains.

Figure 9: Visualize the latent space by mapping the testing images to 2D space with t-SNE. DSN algorithm used for domain adaptation.

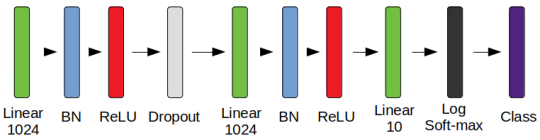
Q3. Describe the architecture and implementation detail of your model.



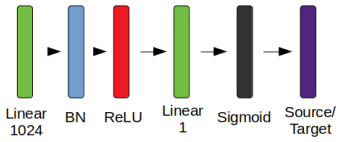
(a) Private source encoder. (b) Shared encoder. (c) Private target encoder.



(d) Shared decoder.



(e) Label classifier.



(f) Domain adversarial network.

Figure 10: Domain separation network (DSN) consist of private source/shared/target encoders, shared decoder, label classifier and domain adversarial network. I used ResNet18 followed by linear layer as feature extractor.

For Improved UDA model, I decided to implement Domain Separation Network[8] as my improved UDA model. Because other improved UDA models are GAN based model and DSN is AutoEncoder (AE) based model, so I decided to implement non-GAN based model. I referred to the paper[8] and this GitHub[9]. Since DSN architecture consist of 3 encoders, 1 decoders, 1 label classifier (label predictor in DANN) and 1 domain adversarial network (domain classifier in DANN). Most of the modules are same with DANN model, so I implemented the exactly same encoder (feature extractor), label classifier and domain classifier architecture. For the decoder architecture, I implemented almost the same architecture stated in the paper with minor changes.

For the training procedure, I used the same training pipeline as DANN. Implementation of improved UDA training also able to train on source only (upper/lower bound model) or to train domain adaptation model. The only differences between DANN are DSN need to calculate 2 more losses, $L_{different}$ and L_{recon}. The L_{task} is Cross entropy loss and $L_{similarity}$ is Binary cross entropy loss. The total loss L is the sum of these four losses. $L = L_{task} + \alpha L_{recon} + \beta L_{different} + \gamma L_{similarity}$, for $\alpha = 0.15$, $\beta = 0.075$ and $\gamma = 0.25$. If only to train on source, only L_{task} will be calculate, other losses will be ignored.

Q4. Discuss what you've observed and learned from implementing your improved UDA model.

By adding L_{recon} and $L_{different}$ to the model really helps model performance better on domain adaptation task. By comparing DANN and DSN model, we can see that it a slightly increment on accuracy. However, if the architecture of decoder too complex may hurt the performance. The training process of UDA models is really unstable, we may need to train few more round to get the average performance of the model.

References

[1] `https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html`

[2] `https://sthalles.github.io/advanced_gans/`

[3] `https://github.com/soumith/ganhacks`

[4] Spectral Normalization for Generative Adversarial Networks, `https://arxiv.org/pdf/1802.05957.pdf`

[5] Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, `https://arxiv.org/pdf/1502.03167.pdf`

[6] Unsupervised Domain Adaptation by Backpropagation, `http://sites.skoltech.ru/compvision/projects/grl/`

[7] `https://github.com/fungtion/DANN_py3`

[8] Domain Separation Networks, `https://arxiv.org/pdf/1608.06019.pdf`

[9] `https://github.com/fungtion/DSN`