CSE641 - Deep Learning Assignment 3 Report

Aditya Chetan[1], Brihi Joshi[2] (Group 28)

{[1]aditya16217, [2]brihi16142}@iiitd.ac.in

May 15, 2020

1 Problem 1

1.1 VAE Implementation

Model Architecture - The encoder and decoder used are purely convolutional. The encoder contains 5 Convolution layers, along with a BatchNorm and ReLU layer. The last convolution layers is mapped to two FC layers, having a size of 10, which is also our bottleneck size. While training, after passing an input through the encoder, the outputs are the *mu* and *logvar*, which are then sent to the parameterize method.
The decoder takes in the latent representation as an input, upscales it with the help of an FC layer, and then has 5 Deconvolution layers corresponding to those in the encoder.
The detailed model description is as follows -

```
  VanillaVAE(
(enc1): Sequential(
  (0): Conv2d(1, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (2): LeakyReLU(negative_slope=0.01)
)
(enc2): Sequential(
  (0): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (2): LeakyReLU(negative_slope=0.01)
)
(enc3): Sequential(
  (0): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (2): LeakyReLU(negative_slope=0.01)
)
(enc4): Sequential(
  (0): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (2): LeakyReLU(negative_slope=0.01)
)
(enc5): Sequential(
  (0): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (2): LeakyReLU(negative_slope=0.01)
)
(fc_mu): Linear(in_features=512, out_features=10, bias=True)
(fc_var): Linear(in_features=512, out_features=10, bias=True)
(decoder_input): Linear(in_features=10, out_features=512, bias=True)
(dec5): Sequential(
  (0): ConvTranspose2d(512, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
  output_padding=(1, 1))
  (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (2): LeakyReLU(negative_slope=0.01)
)
```

```
  (dec4): Sequential(
    (0): ConvTranspose2d(256, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
    output_padding=(1, 1))
    (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): LeakyReLU(negative_slope=0.01)
  )
  (dec3): Sequential(
    (0): ConvTranspose2d(128, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
    output_padding=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): LeakyReLU(negative_slope=0.01)
  )
  (dec2): Sequential(
    (0): ConvTranspose2d(64, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
    output_padding=(1, 1))
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): LeakyReLU(negative_slope=0.01)
  )
  (dec1): Sequential(
    (0): ConvTranspose2d(32, 1, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
    output_padding=(1, 1))
  )
)
```

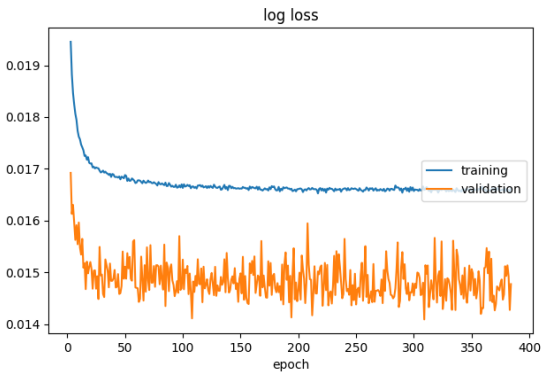
The model is trained for 400 epochs. The loss plot can be seen in Figure 1a. Some of the implementation of the VAE is borrowed from [4].

1.2 Visualising the Generated images

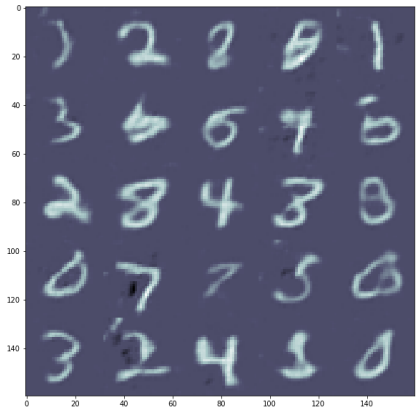
After the model is trained, the following steps are followed to generate the images -

- A batch of arrays of size 10 is sampled from the normal distribution
- The batch is passed through the decoder, the output for which is an image of the required size
- The encoder is **not used** in this step, as we just want to evaluate how well is the decoder able to generate images on it's own without the prior.

The generated image can be seen in Figure 1b.



(a) Loss plot for Vanilla VAE

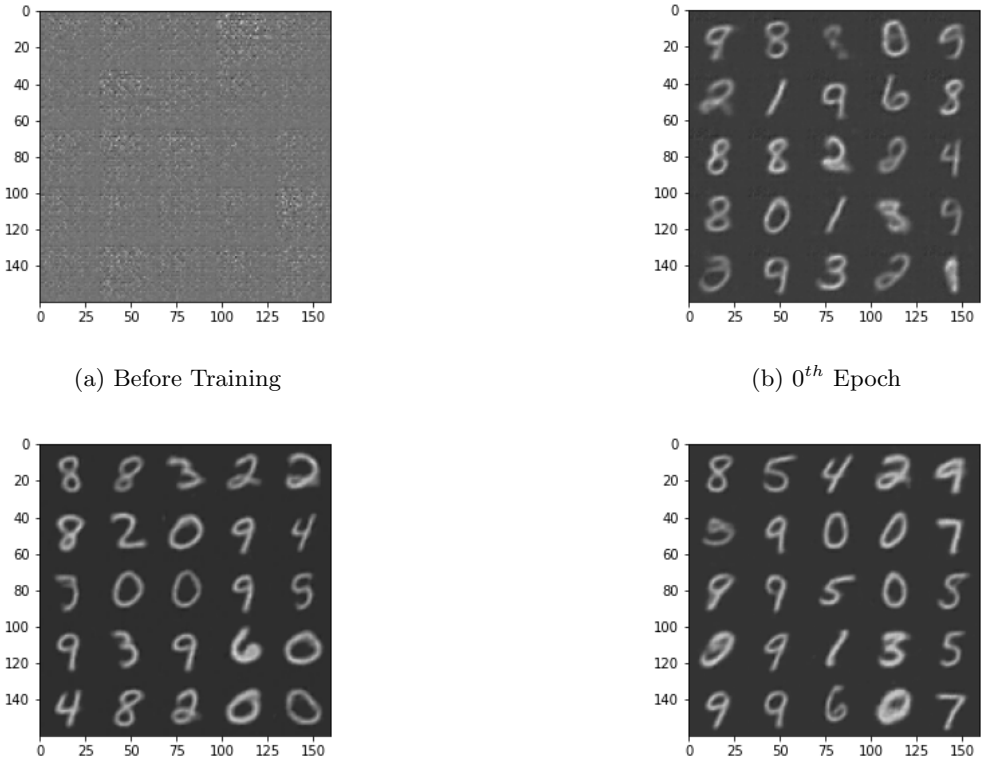


(b) Batch of images generated by the decoder

Figure 1: Plots for Q1 part (1) and (2)

1.3 Improvement in reconstruction quality of MNIST

The reconstruction quality of the MNIST images over epochs is shown in Figure 2.



(a) Before Training

(b) 0^{th} Epoch

(c) 50^{th} Epoch

(d) 380^{th} Epoch

Figure 2: Plots for Q1 part (3)

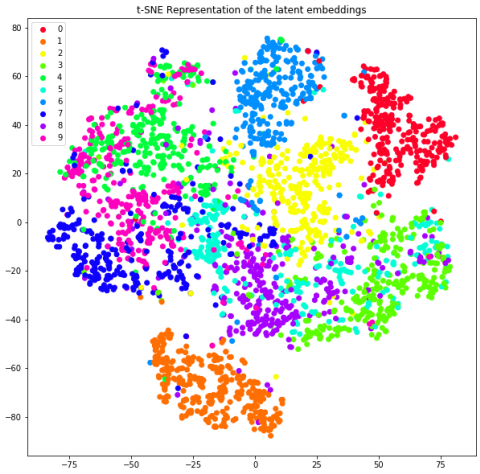
1.4 t-SNE Plot of the Latent representation

On the test set, the latent representation was plotted, colour coded by the class. It can be seen in Figure 3a. It can be seen clearlt that the TSNT embeddings of the latent representations of different digits cluster separately in distinct clusters. This is expected and shows a good quality of training of the VAE.

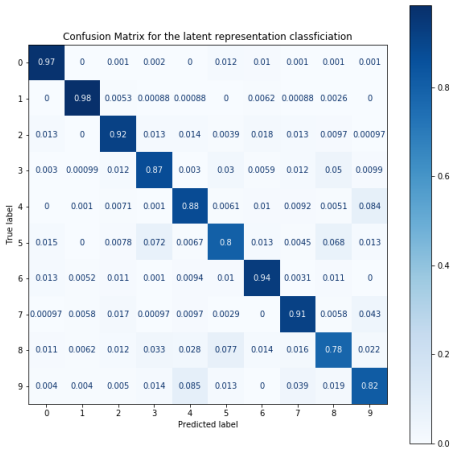
1.5 Training an SVM on the Latent representation

An SVM was trained on the latent representation of the training set, and tested on the latent representation of the test set. The classification report for the same has been provided below. The confusion matrix can be seen in Figure 3b. We can see that the classification performance of the SVM on latent representations if quite good, which indicates that latent representations are able to capture class information and can be used to compress the dataset size, thus enabling smaller sized classification models and more efficient training.

class	precision	recall	f1-score	support
0	0.94	0.97	0.96	980
1	0.98	0.98	0.98	1135
2	0.92	0.92	0.92	1032
3	0.87	0.87	0.87	1010
4	0.85	0.88	0.86	982
5	0.82	0.80	0.81	892
6	0.92	0.94	0.93	958
7	0.91	0.91	0.91	1028
8	0.82	0.78	0.80	974
9	0.83	0.82	0.82	1009
accuracy			0.89	10000
macro avg	0.89	0.89	0.89	10000
weighted avg	0.89	0.89	0.89	10000



(a) t-SNE plot of the latent representation



(b) Confusion Matrix for the SVM trained on Latent representations

Figure 3: Plots for Q1 part (4) and (5)

2 Problem 2

2.1 Cycle Consistent VAE Implementation

The encoder and the decoder architectures are the same as that specified in the paper. Initially, we used Batch Normalisation in the convolution and deconvolution layers. However we observed that the quality of the images were blurry and some colours (like green) were dominating the generated images of the sprites. We then changed it to Instance Normalisation and that increased the quality by a huge margin. The model is trained for 500 epochs and the loss plots for the model are in Figure 4 We referred to the implementation by Jha. et. al [2] and the hyper-parameter settings and training strategy were kept same as that provided in [6]

2.2 Style Transfer

In order to demonstrate qualitative disentanglement of the specified features, we created the Style transfer grids in Figure 5. The topmost row contains the images from where the specified features (s) is taken, and the first column contains images from where the unspecified features (z) is taken. These images have been taken from the **Test set**.

2.2.1 Analysis

The results look promising, however there is some ambiguity in terms of the facial features like eyes, nose, etc. For poses that do not occur frequently (4th row in 5b), the model finds it difficult to generate the specified features properly (the 4th row, 5th column sprite does not have pink hair). We believe this can be solved by adding more samples of that kind in the dataset.

2.3 Interpolation

For the interpolation plots in Figure 6, the first image (1,1) and the last image (8,8) are actual sprites taken from the **Test set**. Across the columns, the specified vector (class identification) gets interpolated and across the rows, the unspecified vector (pose identification) gets interpolated.

2.3.1 Analysis

- **Across Specified Features** - One can notice the gradual change in the hair colour, skin colour and the clothes of the sprites. For example in Figure 6d, one can see the intermediate hair colours like Purple being formed midway Pink and Blue haired sprites.

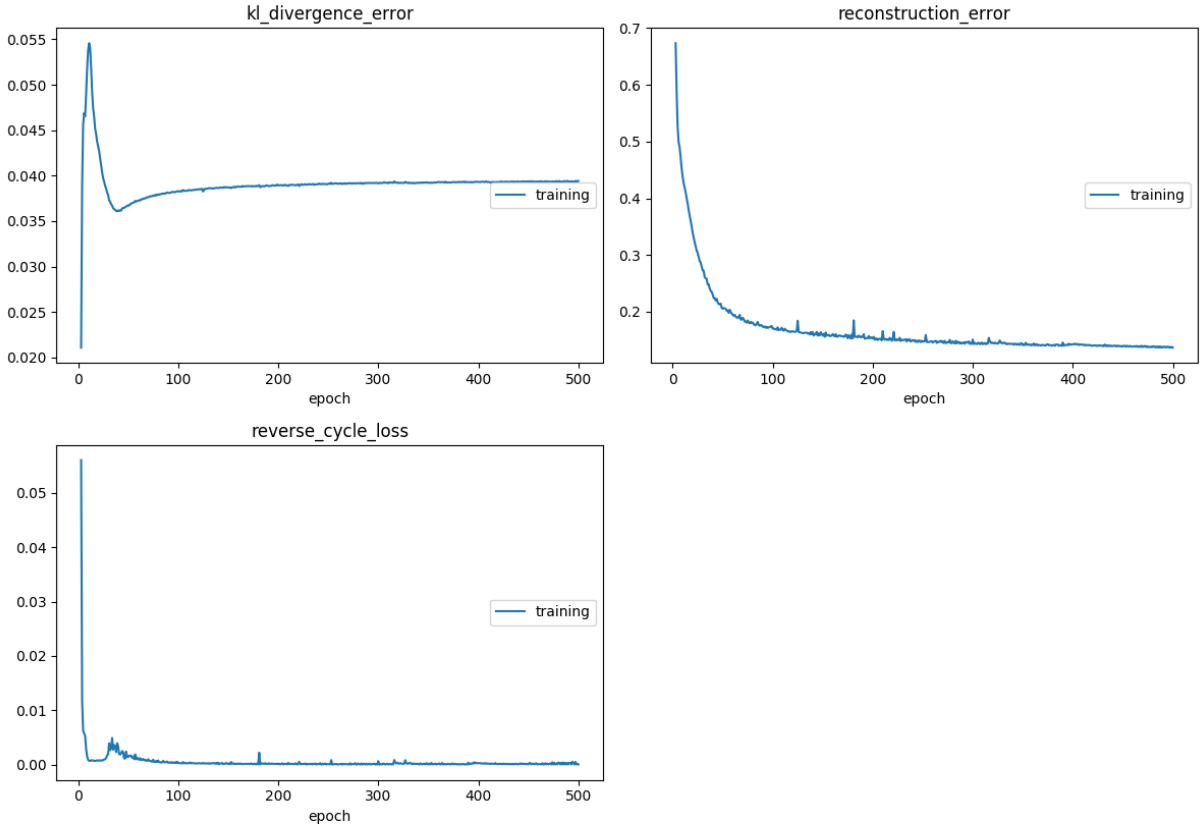


Figure 4: Loss Plots for Cycle Consistent VAEs

- **Across Unspecified Features** - One can notice the gradual change in the pose and weapons of the sprites. For example, in Figure 6c, consider the 1^{st} column. One can see that the leg and arm positions change gradually to incorporate the change in angle of the body.

2.4 Classifiers on Latent Representations

2.4.1 Specified Latent Representation

The classifier is a series of 3 FC layers, followed by Softmax, for predicting for **672 classes**.
Train Accuracy: 0.65
Test Accuracy: 0.677

The training plots can be seen in Figure 7. The model description is given below.

```
Classifier(
  (fc_model): Sequential(
    (fc_1): Linear(in_features=64, out_features=2048, bias=True)
    (leakyrelu_1): LeakyReLU(negative_slope=0.2, inplace)
    (fc_2): Linear(in_features=2048, out_features=1024, bias=True)
```

(a) (b)

Figure 5: Style Transfer Plots

```
    (leakyrelu_2): LeakyReLU(negative_slope=0.2, inplace)
    (fc_3): Linear(in_features=1024, out_features=672, bias=True)
  )
)
```

2.4.2 Unspecified Latent Representation

The classifier is a series of 3 FC layers, followed by Softmax, for predicting for **672 classes**.
Train Accuracy: 0.0024
Test Accuracy: 0.002

The training plots can be seen in Figure 8. The model description is given below.

```
Classifier(
  (fc_model): Sequential(
    (fc_1): Linear(in_features=512, out_features=2048, bias=True)
    (leakyrelu_1): LeakyReLU(negative_slope=0.2, inplace)
    (fc_2): Linear(in_features=2048, out_features=1024, bias=True)
    (leakyrelu_2): LeakyReLU(negative_slope=0.2, inplace)
    (fc_3): Linear(in_features=1024, out_features=672, bias=True)
  )
)
```

2.4.3 Analysis

As it can be clearly seen, the test accuracy for specified features is definitely more than that of unspecified features. This demonstrates that the latent space is clearly disentangled so that the specified features contain information about the class properties of the sprite, whereas the unspecified features contain very less information about the class properties.

2.5 Prediction networks on Latent Representations

2.5.1 Going from $z \rightarrow s'$

Input: Unspecified Representation
Output: Specified Representation

The network was trained for 200 epochs. The network artitechture is given below. **MSE Loss** was used as a loss, since the output of the architecture was a vector with floating point values. The loss plot for the same is found in figure 9

```
UnspecToSpecClassifier(
  (fc_model): Sequential(
    (fc_1): Linear(in_features=512, out_features=256, bias=True)
    (leakyrelu_1): LeakyReLU(negative_slope=0.2, inplace)
    (fc_2): Linear(in_features=256, out_features=128, bias=True)
    (leakyrelu_2): LeakyReLU(negative_slope=0.2, inplace)
    (fc_3): Linear(in_features=128, out_features=64, bias=True)
  )
)
```

The original and reconstructed batch are compared in Figure 10.
Analysis - It can be observed that there are no mis-classifications in the reconstructed batch. This is probably because, unspecified features (like pose) might contain cues about the hair/skin colour and clothes of the sprite, and thus, the class of the sprite can be constructed with the help of the pose. Another reason why this might happen is that for each class, there are multiple poses, thus the data gives better indication for the construction of specified features from the unspecified features.

2.5.2 Going from $s \rightarrow z'$

Input: Specified Representation
Output: Unspecified Representation

The network was trained for 200 epochs. The network artitechture is given below. **MSE Loss** was used as a loss, since the output of the architecture was a vector with floating point values. The loss plot for the same is found in figure 11

```
SpecToUnspecClassifier(
  (fc_model): Sequential(
    (fc_1): Linear(in_features=64, out_features=128, bias=True)
    (leakyrelu_1): LeakyReLU(negative_slope=0.2, inplace)
    (fc_2): Linear(in_features=128, out_features=256, bias=True)
    (leakyrelu_2): LeakyReLU(negative_slope=0.2, inplace)
    (fc_3): Linear(in_features=256, out_features=512, bias=True)
  )
)
```

The original and reconstructed batch are compared in Figure 12.
Analysis - It can be observed that **all of the reconstructed images are mis-classified** for the batch. This is probably because of the one-to-many mapping that is present in the dataset. Each class has multiple poses and thus, given a class representation, it is difficult to pinpoint the exact pose that a sprite of a certain class would have. Thus, even though the poses are not random, they are not the poses corresponding to the original batch. These results re-assert the motivation for the Cyclic VAE architecture, which is to ensure that the specified feature information does not leak into the unspecified feature space.

3 Problem 3

3.1 Toy Dataset

- In this question we were supposed to implement the Shallow DANN and replicate the experiment on the Toy Twinning Moon dataset given in the paper.
- For the Shallow DANN, I followed the exact architecture as shown in the paper. For the Gradient Reversal Layer, I used an extension of the PyTorch Function module, referred from a GitHub repository [1].
- For the part where we had to deactivate the GRL, I simply allowed the gradients to backpropagate normally.
- For the dataset, the paper mentioned that they showed results on 150 points from each domain. However, there was no mention of the training set size or whether the points plotted were the same points that the model was trained on.
- To be rigorous in my analysis, and as suggested on the Classroom discussion, I generated 375 points for each domain, and split them in the ratio of 4:1.
- This way, I had 150 points in a validation set, 75 points in each domain, which I would not expose to the model and use only for generating results. The validation set points are plotted in Figure 15.
- We referred to a GitHub repository [5] for generating and plotting code for this dataset.
- I trained on the other remaining points. I used Cross Entropy loss for both domain and classification branches (This would have the same effect as the losses mentioned in the paper), and trained using Adam Optimiser with a learning rate of 0.005 for 1000 epochs, as SGD was not helping me replicate the plots.
- The training plots for Shallow DANN and NN are shown in Figures 13 and 14 respectively.
- The Label Classification decision boundaries for Shallow DANN and Normal NN are shown in Figure 16. As we can see in the plot, the Shallow DANN can classify all the validation Target Domain points correctly, whereas the Simple NN cannot to that, even though it gets the Source domain points right, which is expected, as it does not use the Gradient Reversal layer and does not perform domain regularization.
- The Domain Classification Decision boundaries for Shallow DANN and Normal NN are shown in Figure 17. Again, we see that the DANN does not work so well in separating the points on the basis of domains, which is expected, as it learns domain-invariant features.
- For the Shallow DANN, the label classification performance and domain classification performance over the validation set was as follows:

```
domain: src

label accuracy
                precision    recall  f1-score   support

         0.0       1.00      1.00      1.00        37
         1.0       1.00      1.00      1.00        38

    accuracy                           1.00        75
   macro avg       1.00      1.00      1.00        75
weighted avg       1.00      1.00      1.00        75

----------------------------------------------------------

domain: tar

label accuracy
                precision    recall  f1-score   support

         0.0       1.00      1.00      1.00        37
         1.0       1.00      1.00      1.00        38

    accuracy                           1.00        75
   macro avg       1.00      1.00      1.00        75
weighted avg       1.00      1.00      1.00        75

----------------------------------------------------------
```

```
domain accuracy
                precision    recall  f1-score   support

         0.0       0.38      0.32      0.35        75
         1.0       0.41      0.47      0.43        75

    accuracy                           0.39       150
   macro avg       0.39      0.39      0.39       150
weighted avg       0.39      0.39      0.39       150
```

As we can see, the Shallow DANN has adapted to the Target domain extremely well, giving a 100% validation accuracy for Label Classification on both the Source and Target domains. At the same time, the Domain Classification accuracy is extremely low, as it has learnt domain-invariant features.

- For the Simpls NN, the label classification performance and domain classification performance over the validation set are reported below. We can notice that the simple NN does not perform as well on the Target domain for label classification, as it has not learnt domain-invariant features. However, the domain classification accuracy is quite high, as it is able to backpropagate the gradients from the Domain Classification loss correctly.

```
domain: src

label accuracy
                precision    recall  f1-score   support

         0.0       1.00      1.00      1.00        37
         1.0       1.00      1.00      1.00        38

    accuracy                           1.00        75
   macro avg       1.00      1.00      1.00        75
weighted avg       1.00      1.00      1.00        75

--------------------------------------------------------

domain: tar

label accuracy
                precision    recall  f1-score   support

         0.0       0.96      0.65      0.77        37
         1.0       0.74      0.97      0.84        38

    accuracy                           0.81        75
   macro avg       0.85      0.81      0.81        75
weighted avg       0.85      0.81      0.81        75

--------------------------------------------------------

domain accuracy
                precision    recall  f1-score   support

         0.0       0.97      0.92      0.95        75
         1.0       0.92      0.97      0.95        75

    accuracy                           0.95       150
   macro avg       0.95      0.95      0.95       150
weighted avg       0.95      0.95      0.95       150
```

3.2 MNIST → MNIST-m

- I normalized the images by performing Standard scaling using the mean and standard deviation of the pixel values.
- For both the models, I had to apply Batch Normalisation between the Convolutional layers, otherwise the models were not learning properly. The rest of the architecture I used is same as that given in the paper. I also used the modification scheme for the Domain Regularization constant (λ) that was given in the paper.

- I used Adam optimiser, with learning rate 0.01, and trained for 200 epochs. I incorporated a `LogSoftMax` layer into the model architecture and used the `NLLLoss` function in PyTorch.
- For loading MNIST-m in PyTorch, I referred to a GitHub Repository [3].
- For the DANN, the label classification performance over both the domains is given below:

```
    domain: target

                  precision    recall  f1-score   support

             0.0       0.85      0.80      0.82       878
             1.0       0.75      0.73      0.74      1016
             2.0       0.74      0.74      0.74       933
             3.0       0.71      0.75      0.73       908
             4.0       0.70      0.75      0.72       890
             5.0       0.71      0.78      0.74       807
             6.0       0.86      0.73      0.79       856
             7.0       0.85      0.72      0.78       914
             8.0       0.70      0.73      0.72       880
             9.0       0.68      0.76      0.72       919

        accuracy                           0.75      9001
       macro avg       0.75      0.75      0.75      9001
    weighted avg       0.75      0.75      0.75      9001

    ----------------------------------------------------

    domain: src

                  precision    recall  f1-score   support

             0.0       0.98      0.99      0.98       980
             1.0       0.99      0.98      0.99      1135
             2.0       0.97      0.97      0.97      1032
             3.0       0.98      0.97      0.97      1010
             4.0       0.97      0.97      0.97       982
             5.0       0.98      0.97      0.97       892
             6.0       0.98      0.97      0.98       958
             7.0       0.98      0.97      0.97      1028
             8.0       0.97      0.98      0.97       974
             9.0       0.95      0.97      0.96      1009

        accuracy                           0.97     10000
       macro avg       0.97      0.97      0.97     10000
    weighted avg       0.97      0.97      0.97     10000

    ----------------------------------------------------
```

  We see that the model has learned the Source domain (MNIST) pretty well. For the Target domain (MNIST-m) also the model gives a pretty decent accuracy.

- For the Source-only NN model, the label classification performance is given below:

```
    domain: target

                  precision    recall  f1-score   support

             0.0       0.41      0.49      0.45       878
             1.0       0.42      0.60      0.49      1016
             2.0       0.63      0.42      0.51       933
             3.0       0.55      0.46      0.50       908
             4.0       0.47      0.52      0.50       890
             5.0       0.77      0.40      0.53       807
             6.0       0.77      0.51      0.61       856
             7.0       0.78      0.47      0.59       914
             8.0       0.30      0.68      0.41       880
             9.0       0.61      0.40      0.49       919

        accuracy                           0.50      9001
       macro avg       0.57      0.50      0.51      9001
    weighted avg       0.57      0.50      0.51      9001

    ----------------------------------------------------
```

```
domain: src

                precision    recall  f1-score   support

           0.0       0.99      1.00      0.99       980
           1.0       0.98      1.00      0.99      1135
           2.0       0.99      0.98      0.99      1032
           3.0       0.97      1.00      0.98      1010
           4.0       0.99      0.99      0.99       982
           5.0       1.00      0.97      0.98       892
           6.0       0.99      0.99      0.99       958
           7.0       0.99      0.98      0.99      1028
           8.0       0.99      1.00      0.99       974
           9.0       0.99      0.98      0.98      1009

      accuracy                           0.99     10000
     macro avg       0.99      0.99      0.99     10000
  weighted avg       0.99      0.99      0.99     10000

--------------------------------------------------------
```

Here, again the Label Classification over the source domain is pretty great. However, the important thing to notice is that the label classification performance over the Target Domain decreases, since the Source only model has no Domain Classification head, and no Gradient reversal layer, therefore, it has no way to learn Domain Invariant features and the performance over the Target domain reduces. Hence, this result is as expected.

- Lastly, we see in Figure 18, the effect of domain adaptation on the activations of the Feature Extraction part of the network. They are the TSNE projections of the flattened activation obtained at the end of the Feature extraction part. Blue points correspond to the target domain and red points correspond to the source domain. In case of adaptation, the activations of the two domains are brought much closer than when there is no adaptation, indicating that the features learnt by the feature extractor in the case of domain adaptation are domain-invariant.
- This result is in agreement with the paper and is expected.
- Generating TSNE embeddings for the entire test set (approx. 18k samples) was taking too much time on Colab, hence I have generated embeddings for randomly sampled 30% of the test set.
- The training plots for the DANN and the Source-only network are given in Figure 19 and Figure 20 respectively.

References

[1] Gradient reversal layer. `https://github.com/fungtion/DANN/blob/master/models/functions.py`. We referred to this script to understand how Gradient Reversal Layer can be implemented in PyTorch.

[2] Jha et. al, eccv 2018. `https://github.com/ananyahjha93/cycle-consistent-vae`. This is the GitHub repo for the implementation of the Cycle Consistent VAEs.

[3] Mnist-m dataloader. `https://github.com/hehai131/DANN/blob/00113be0d4acf78137a94d1d01062db1fb145693/utils.py`. We referred to this script to for reference code for a PyTorch data loader for the MNIST-m dataset.

[4] Pytorch-vae. `https://github.com/AntixK/PyTorch-VAE`. This is the GitHub repo we referred to for the implmentation of vanilla VAEs.

[5] Twinning moons data. `https://github.com/GRAAL-Research/domain_adversarial_neural_network/blob/master/experiments_moons.py`. We referred to this function to generate and plot our Twinning moons dataset.

[6] Jha, A. H., Anand, S., Singh, M., and Veeravasarapu, V. Disentangling factors of variation with cycle-consistent variational auto-encoders. In *Computer Vision – ECCV 2018* (Cham, 2018), V. Ferrari, M. Hebert, C. Sminchisescu, and Y. Weiss, Eds., Springer International Publishing, pp. 829–845.

(a) (b)

(c) (d)

Figure 6: Interpolation Plots

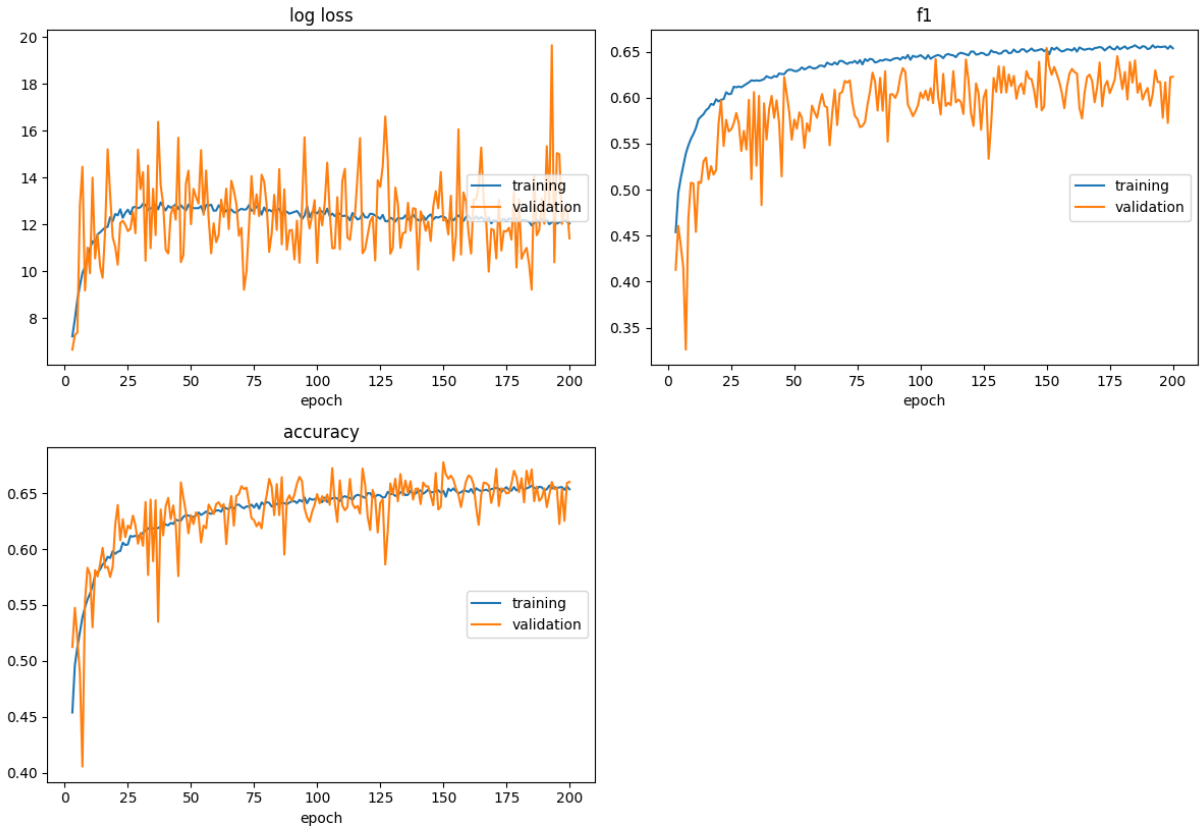


Figure 7: Loss Plots for classifier trained on Specified Latent Representation

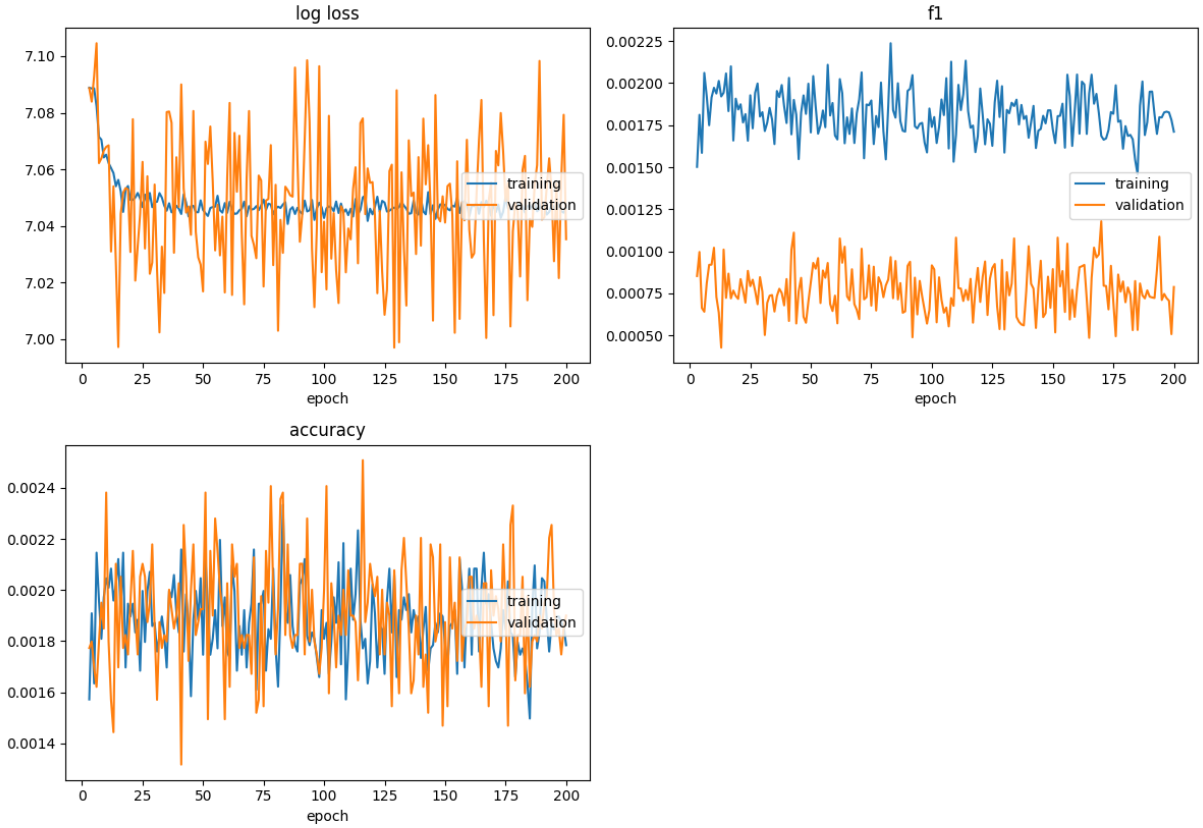


Figure 8: Loss Plots for classifier trained on Unspecified Latent Representation

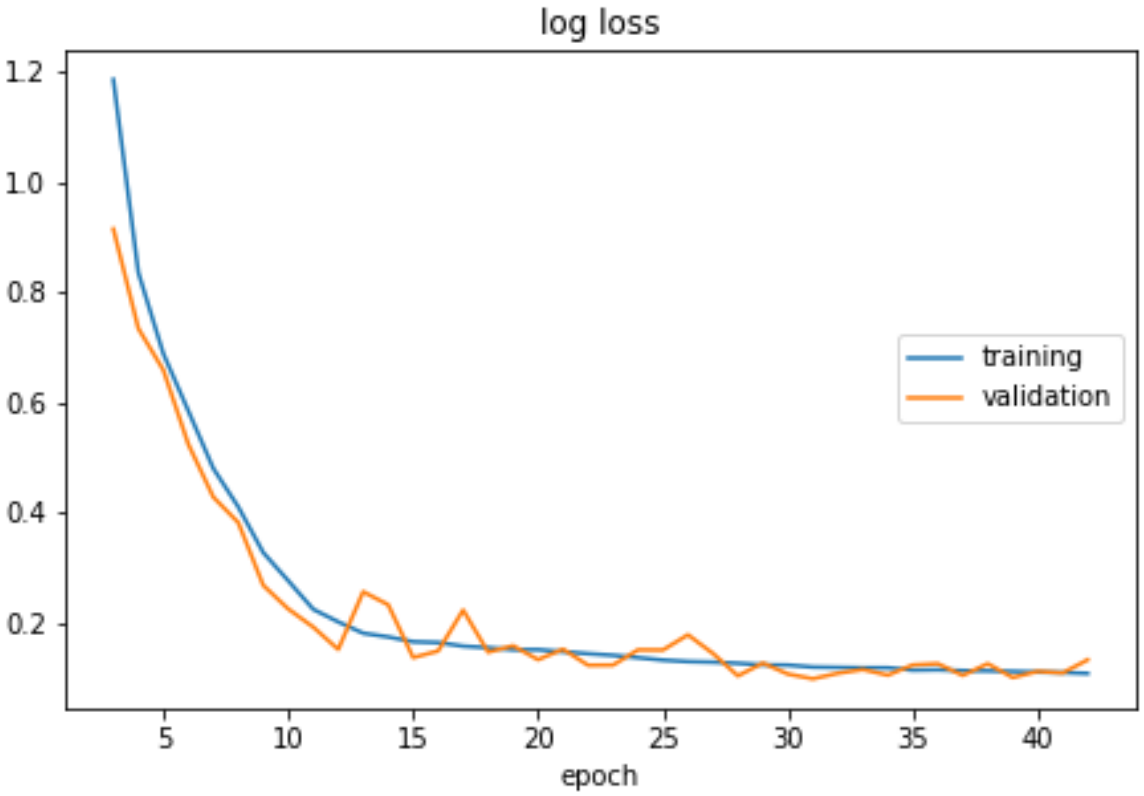


Figure 9: Loss plot for going from $z \rightarrow s'$

(a) Original Batch - $Decoder(z, s)$

(b) Reconstructed Batch - $Decoder(z, s')$

Figure 10: Examples from $z \rightarrow s'$

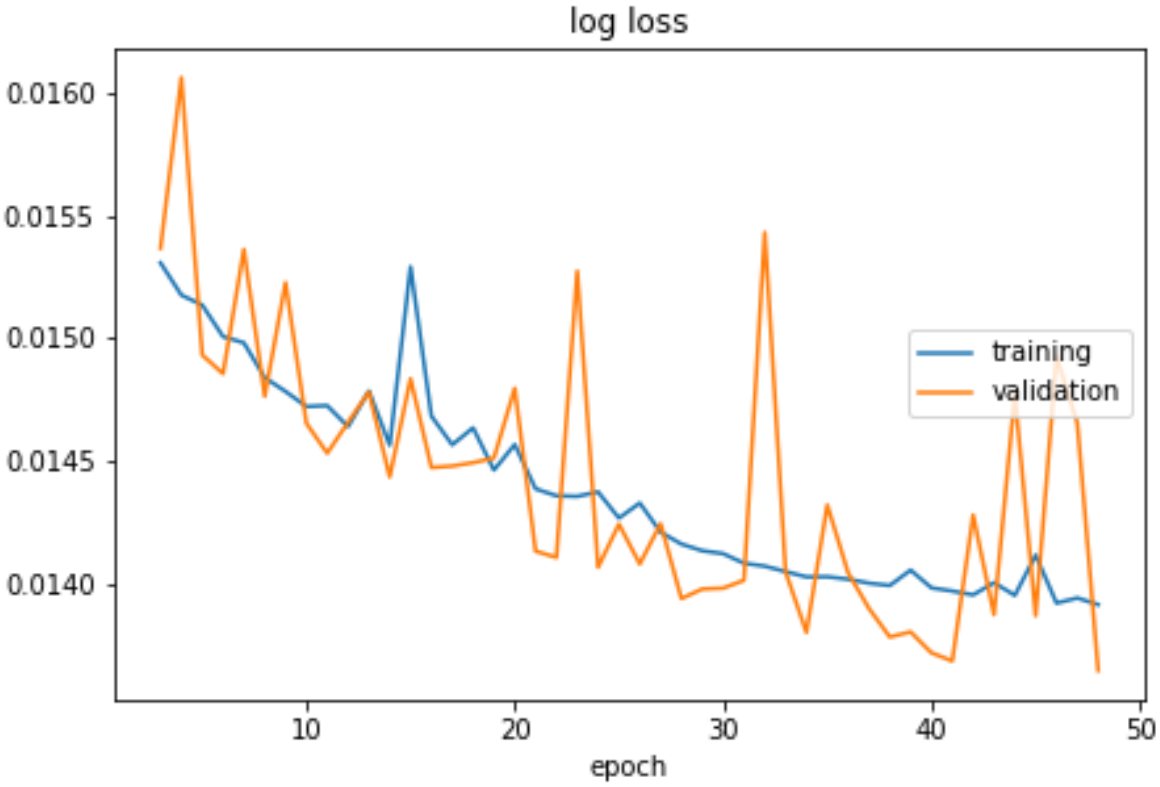


Figure 11: Loss plot for going from $s \rightarrow z'$

(a) Original Batch - $Decoder(z, s)$

(b) Reconstructed Batch - $Decoder(z', s)$

Figure 12: Examples from $s \rightarrow z'$

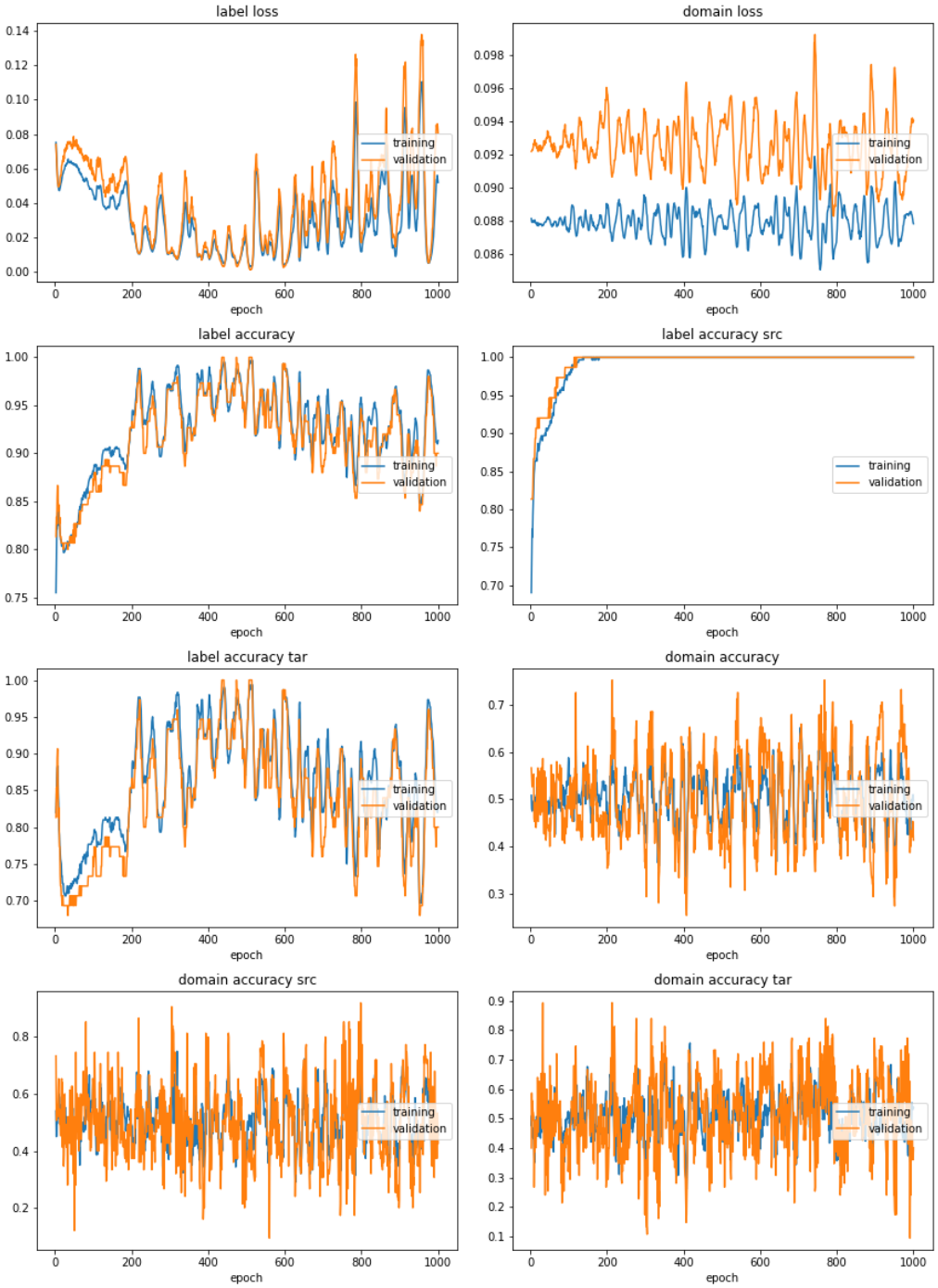


Figure 13: Training Plots for Shallow DANN

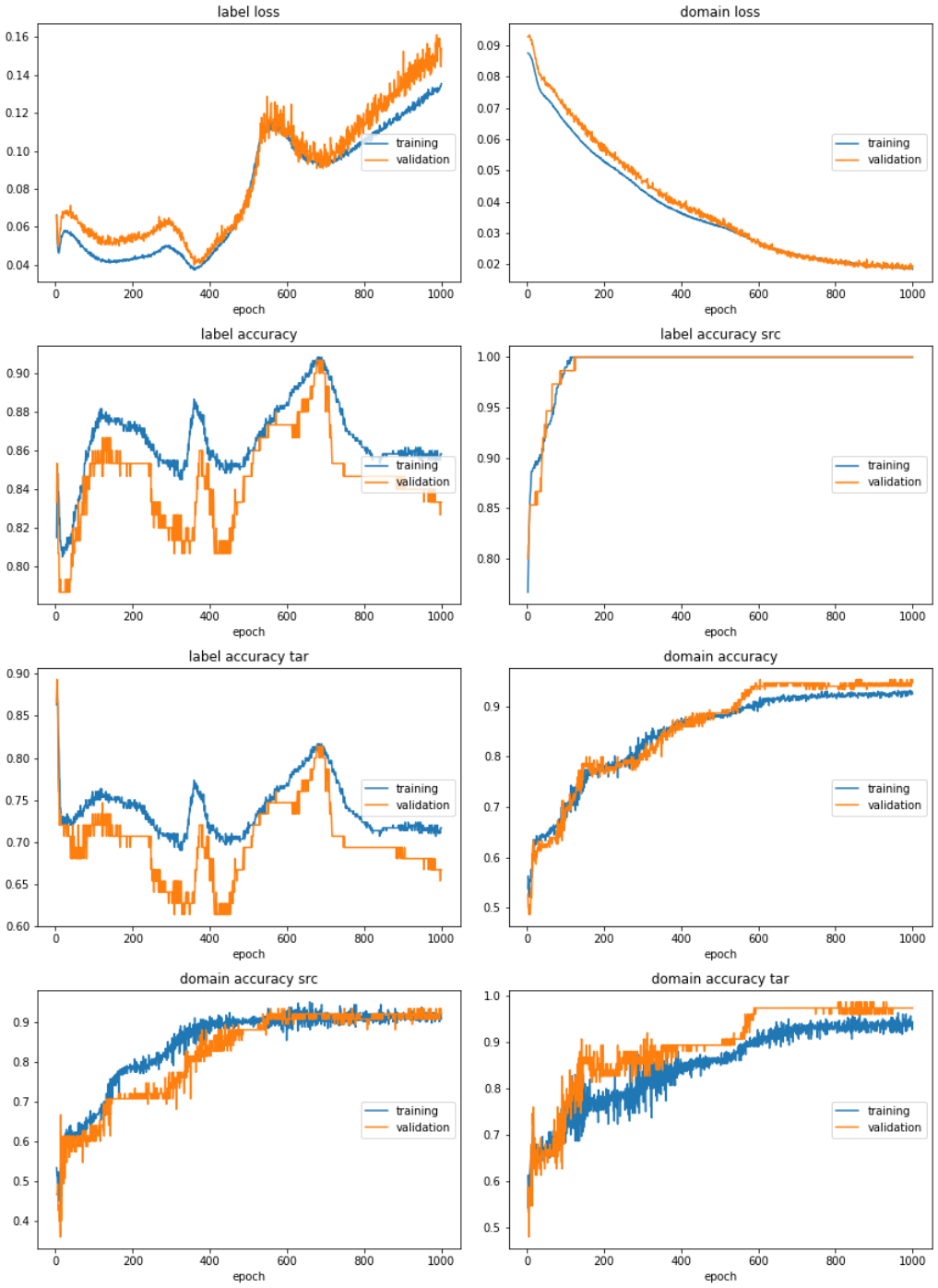


Figure 14: Training Plots for simple NN

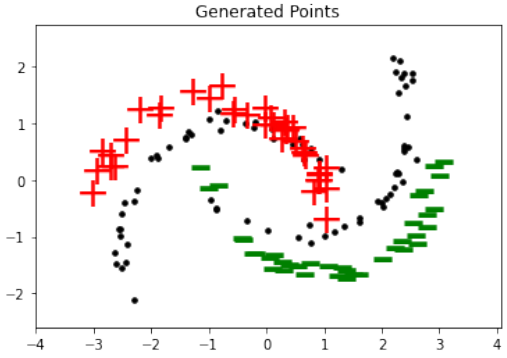
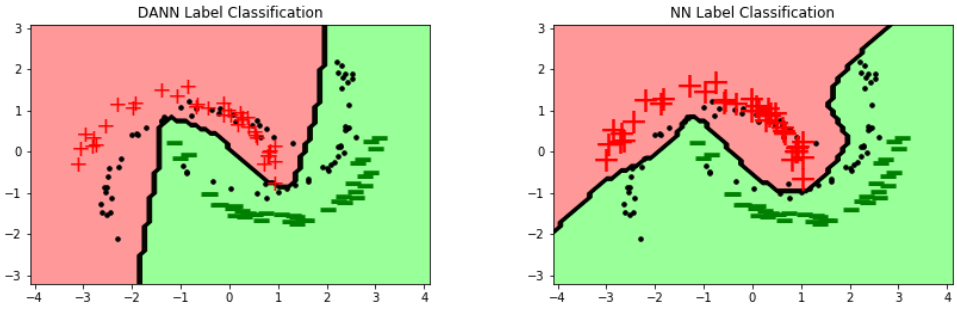
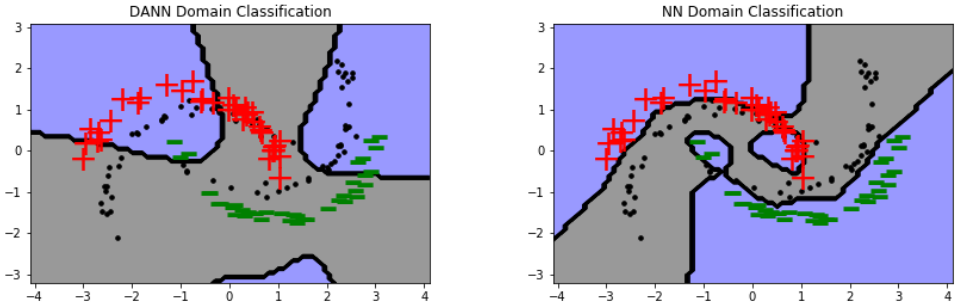


Figure 15: Original Points for Twinning moons dataset. (Black points are the Target domain)



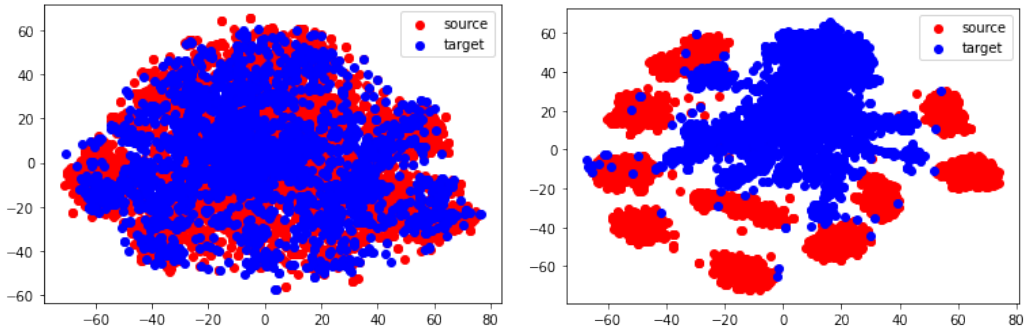
(a) Shallow DANN (b) Simple NN

Figure 16: Label Classification



(a) Shallow DANN (b) Simple NN

Figure 17: Domain Classification



(a) Adapted (DANN) (b) Non-Adapted (Source only)

Figure 18: Effect of Domain Adaptation on MNIST → MNIST-m

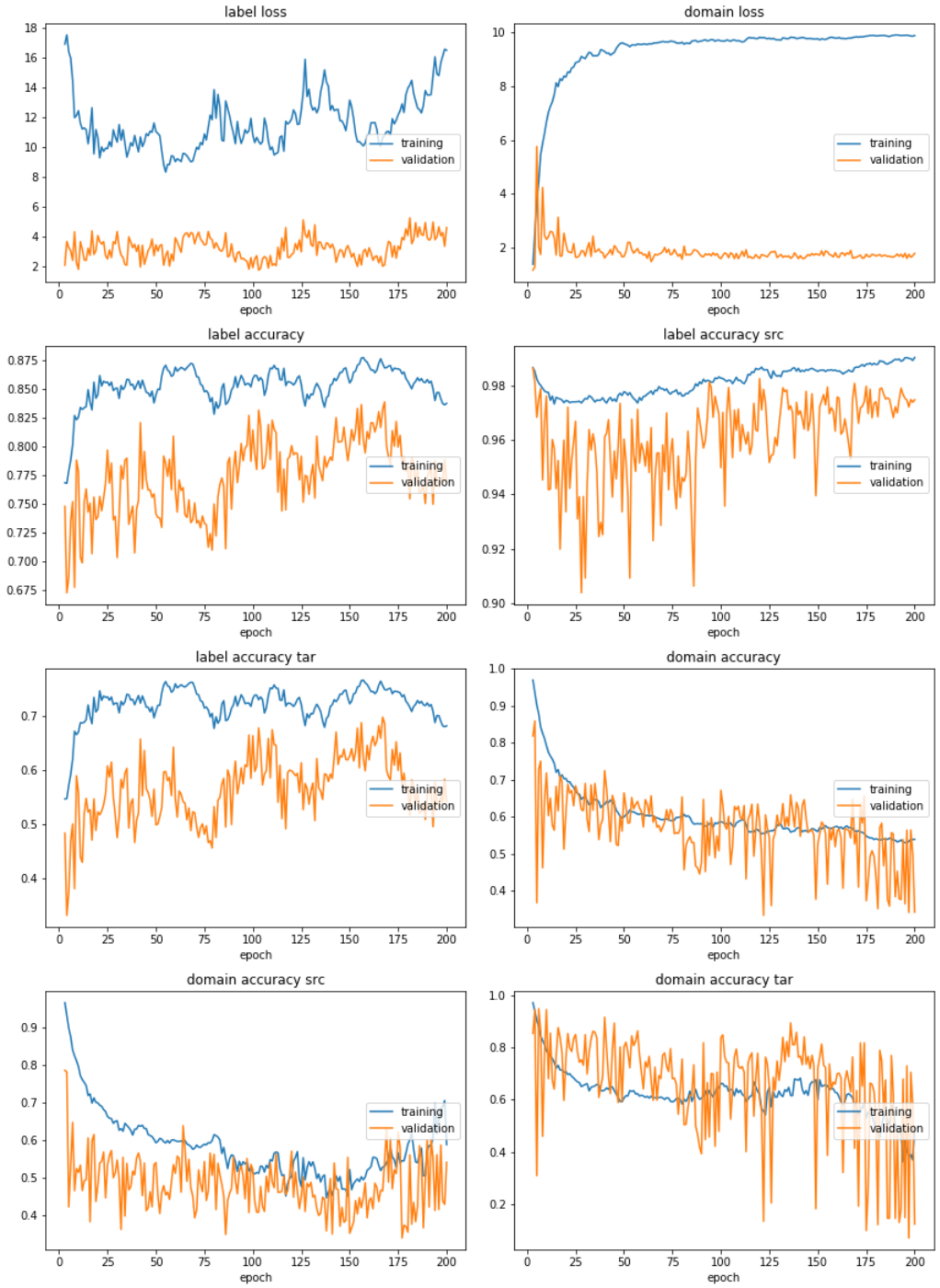


Figure 19: Training Plots for DANN

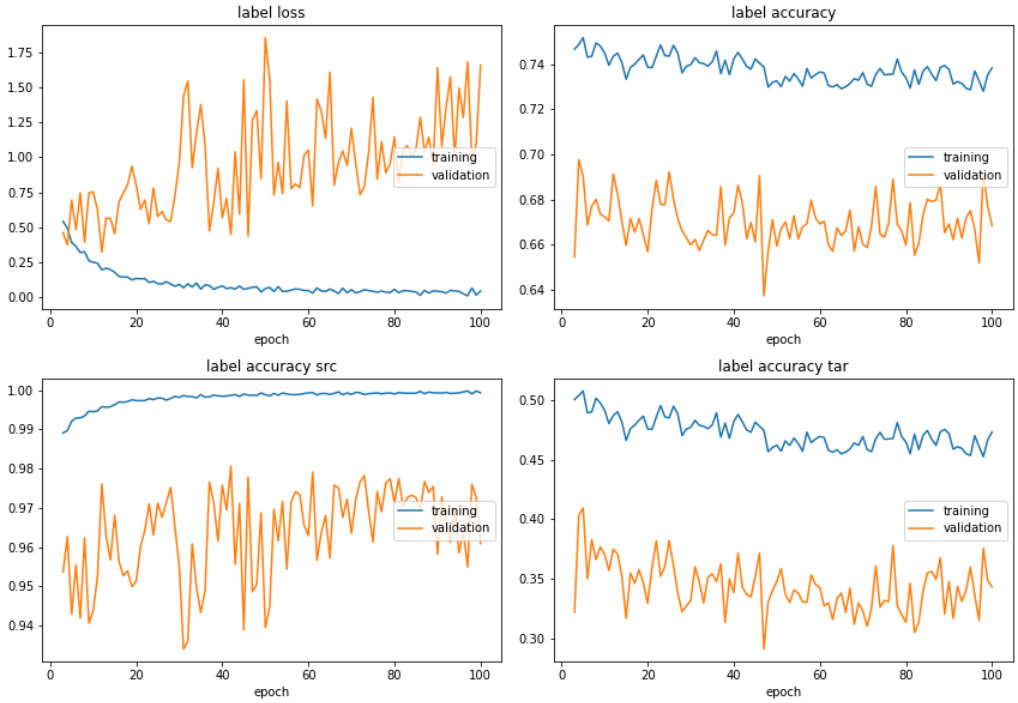


Figure 20: Training Plots for Source-only model