Introduction to Machine Learning. Project.

Final Submission Report

Done by Vladislav Savchuk DS-01
Based on MCD[1] paper
Link with code: https://github.com/qvviko/domain-adaptation

Abstract

In this assignment we were tasked to implement a method that would perform domain adaptation. This is a field in machine learning related to transfer learning, which aims to remove the dissimilarity gap between different domain by reusing already gained knowledge in a source domain to learn a target domain. Our setup was an unsupervised domain adaptation (which means that we don't know the true labels for the target domain), where we had SVHN as a source dataset and MNIST as a target dataset. Many solutions for this problem exists and after reading many of them and trying to implement some, one of best was implemented and will presented in this report: MCD [1]. During the project I've learned a lot of new techniques (e.g. latent space visualization, ablation study etc.) and found them useful when trying to understand how the domain adaptation happens and can be performed. The code with the results can be found here (trained locally, results saved as an output of notebook).

Introduction

In previous submission we have created a baseline model, which was a simple CNN, that wasn't doing any domain adaptation and tested how it would perform on the test data. In this report, I'll present a model which tries to remove a gap between the domain and target images and outperforms the baseline model because of it.

The report consists of the following parts:

- Overall Idea of the method (intuition on how it works and why it works)
- Architecture
- Description of the training process
- Experiments with implementation (hyperparameter tuning, ablation study, latent space visualization)
- Evaluation of the method
- Conclusion (with suggestion for improvements)

Related Works

Many solutions for the unsupervised domain adaptation was already proposed and researched, some of them uses some kind of improved GAN in their architecture (e.g. PixelDA[2], TarGan[3]), however complexity of those methods can make it hard to use them

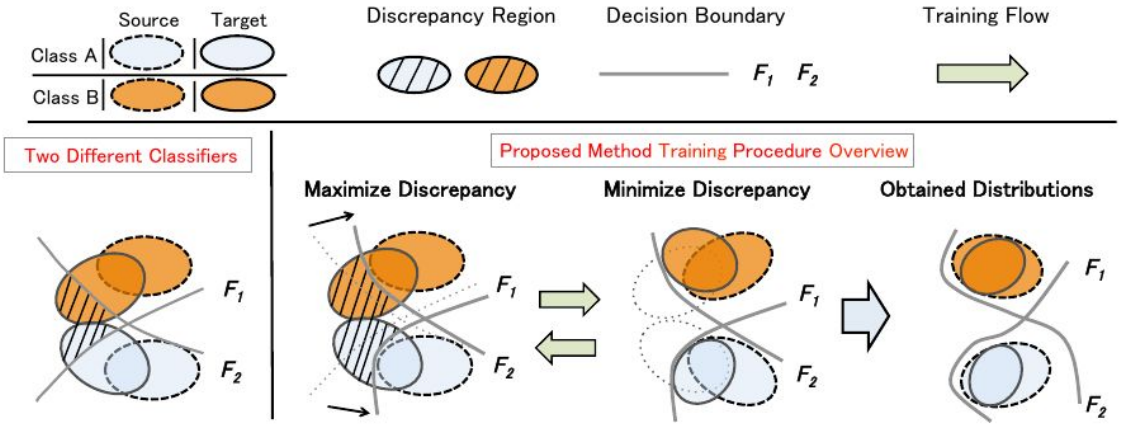
in some problems. Some models try to improve it by using only part of this architecture (e.g. discriminator in TripNet[4]).
The similar idea in those solution is having some kind of a encoder network and try to push it to generate features from target domain, that will be similar to the features of source domain, which is one of the ideas used in the paper that I've implemented.

Method

Overall Idea

In our problem we have a labeled set of images X_s (source) and set of unlabeled images X_t (target). The architecture contains one encoder network which is trained to extract the features of the images and two classifiers, which classify the pictures to their respective classes. Those networks are trained on source images and perform poorly on predicting the target images.

The goal of the method proposed by the paper is to align the features of target images produced by encoder to the features of source images, so that the encoder produces features from target images that are already known to classifiers.



(Figure taken from the paper)

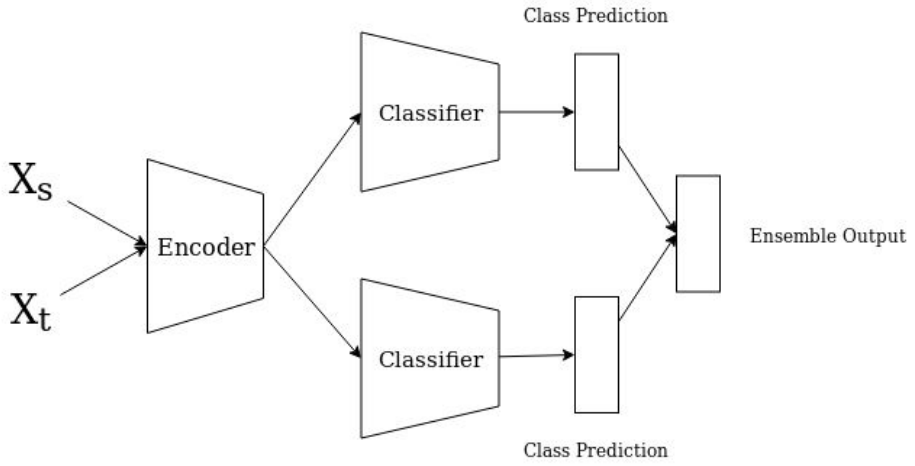
The intuition behind the idea, is that target samples will be outside the support of the source and will be likely classified differently by the classifiers (See the leftmost side of the figure above, the region denoted by black lines is the one which is outside the support of the source). We then can measure the disagreement between two classifiers and train encoder to minimize it, so that the it will try to avoid generating features that are out of the source support. To measure the disagreement we will use the difference between two probabilistic outputs of the classifiers, which shows how much they disagree (this metric can be also called discrepancy).

In order to obtain a encoder which minimizes dissimilarity between two classifiers outputs, we firstly try to train classifiers to maximize their discrepancy (see second image from left to right from the figure), so that our classifier are not very similar in the way they perform their predictions and can detect target images outside the source support. We then train the encoder to minimize discrepancy and encourage it to generate features inside the source support (third image). After those steps the distribution of target features are inside the

support of the source and can be properly classified. This is how the domain adaptation is performed (last image).

Architecture

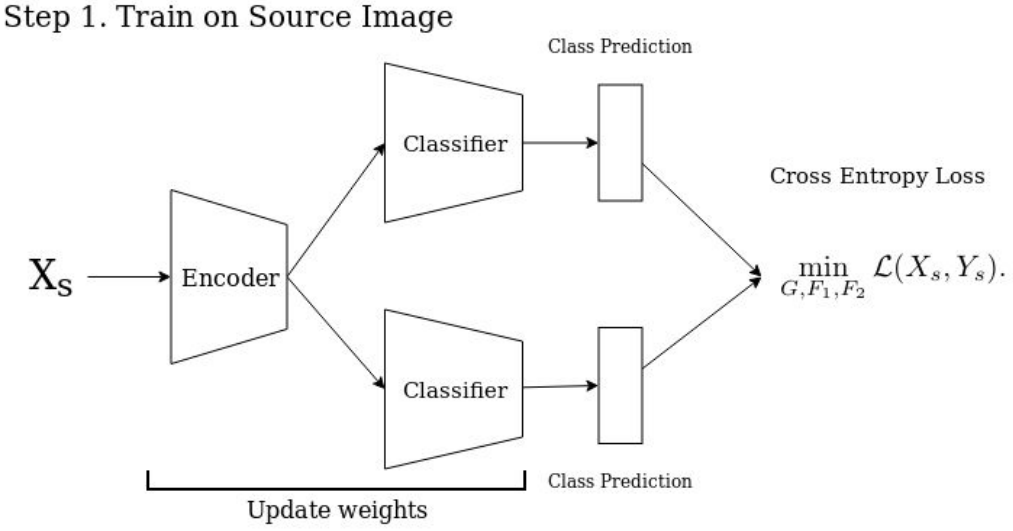
The overall result and architecture is very different from the baseline model, which was a simple CNN. In this model, we don't only have a much higher accuracy (increase in 30%), but also have a total of three networks: one encoder and two classifiers.
The architecture can be drawn as follows:



The image from source and target domain goes through an encoder which extracts their features, then they goes through a classifiers networks and output a probabilistic distribution to which class the sample belongs to.

Training Process

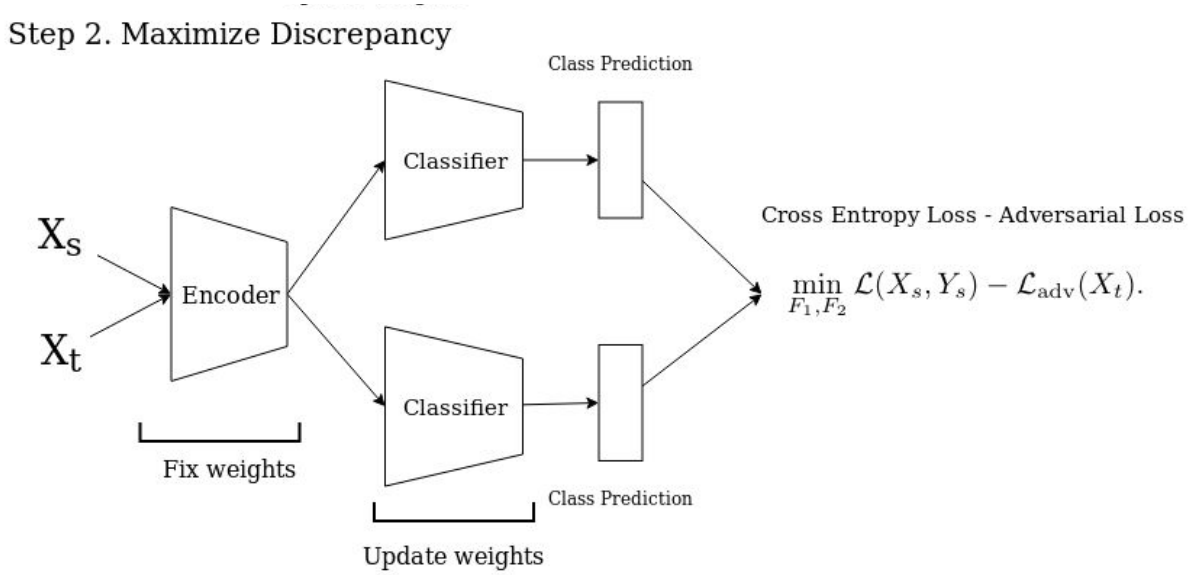
Now let's consider the steps for training of our network.



Step 1. We train our our architecture on Source images, so that it can learn task specific features. The networks is trained to minimize the Cross Entropy Loss:

$$\mathcal{L}(X_s, Y_s) = -\mathbb{E}_{(\mathbf{x_s}, y_s)\sim(X_s, Y_s)} \sum_{k=1}^{K} \mathbb{1}_{[k=y_s]} \log p(\mathbf{y}|\mathbf{x}_s)$$

(Where $p(y|x_s)$ is the notation indicating an distribution output from the classifier)



Step 2. We train our classifiers, while the weights of encoder are fixed. We train them to increase the discrepancy, so that we can find target feature outside the support of the source. The objective is to minimize the following loss:

$$\min_{F_1,F_2} \mathcal{L}(X_s, Y_s) - \mathcal{L}_{\text{adv}}(X_t)$$

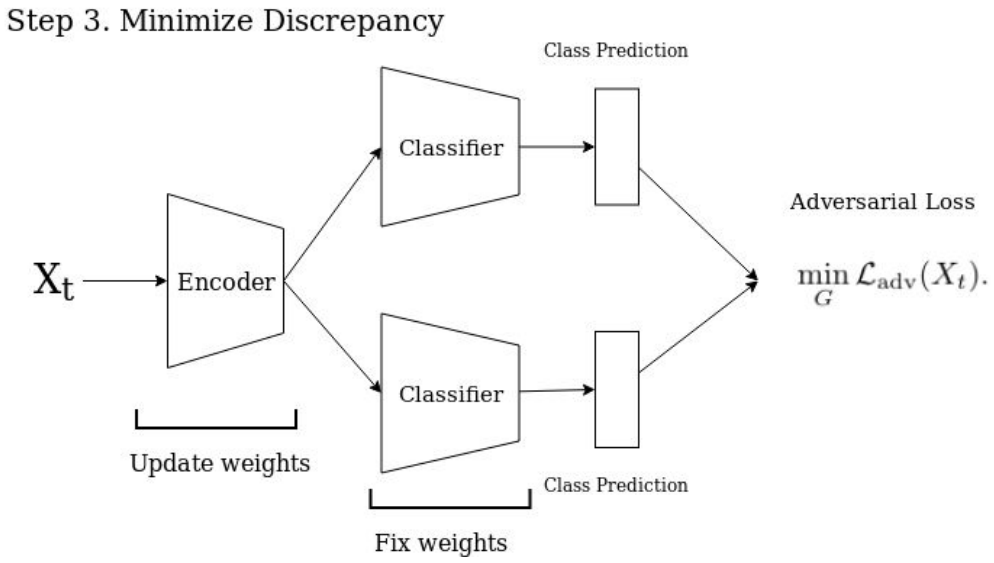
There is an addition of classification loss on the source, as without it the performance drops significantly (check ablation study in experiments)
Where adversarial loss is following:

$$\mathcal{L}_{\text{adv}}(X_t) = \mathbb{E}_{\mathbf{x_t}\sim X_t}[d(p_1(\mathbf{y}|\mathbf{x_t}), p_2(\mathbf{y}|\mathbf{x_t}))]$$

(Basically a discrepancy loss between output of two classifiers)

The discrepancy function that have been chosen is an absolute value of the difference between the two classifiers probabilistic outputs:

$$d(p_1, p_2) = \frac{1}{K} \sum_{k=1}^{K} |p_{1k} - p_{2k}|$$



Step 3. Finally, we train our encoder to minimize the discrepancy, while the classifiers are fixed, so it can generate features that will be inside the support of the source. Therefore, we minimize the following loss:

$$\min_G \mathcal{L}_{\text{adv}}(X_t)$$

It was found out that we can perform this step multiple times for the same batch, which can sometimes increase the performance on some types of problem. The amount of times we repeat this step is a hyperparameter of an algorithm.

To gain mathematical insights about the work of the this method, you can check the paper itself, as it is better described there.

Experiments and Results

For the experimentation and evaluation I've used the following configuration of networks. Encoder consists of the following layers:

```
Encoder(
  (seq1): Sequential(
    (0): Conv2d(1, 32, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
    (4): Conv2d(32, 32, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (5): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): ReLU()
    (7): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
    (8): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (9): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): ReLU()
  )
  (seq2): Sequential(
    (0): Linear(in_features=4096, out_features=2048, bias=True)
    (1): BatchNorm1d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): Dropout(p=0.5, inplace=False)
  )
)
```

Both classifiers consists of the following layers:

```
Classifier(
  (seq): Sequential(
    (0): Linear(in_features=2048, out_features=1024, bias=True)
    (1): BatchNorm1d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): Linear(in_features=1024, out_features=10, bias=True)
  )
)
```

The implementation was done by referring to the architecture of the paper [1], the idea how to construct some of them was taken by source code[6] for MCD that can be found in the paper and some improvements (like pre training the models for some epochs before doing the main steps of domain adaptation or introducing learning rate schedulers) were made by myself.

We can extract a number of hyperparameters that could be tuned for our model:

- number of epochs
- number of times we repeat the third step (described previously) during domain adaptation
- the structure of networks
- the learning rate.

After several runs, it was evident that the networks don't overfit on large amount of epochs (around 50), so it was decided to leave it as 50. On the other hand, the accuracy on all datasets seemed to jumping to different accuracies during the end of training, which showed that the learning rate was too high, so to fix it I created **learning rate scheduler**, which was decreasing the learning rate at later epochs, so that algorithm could converge to one specific accuracy.

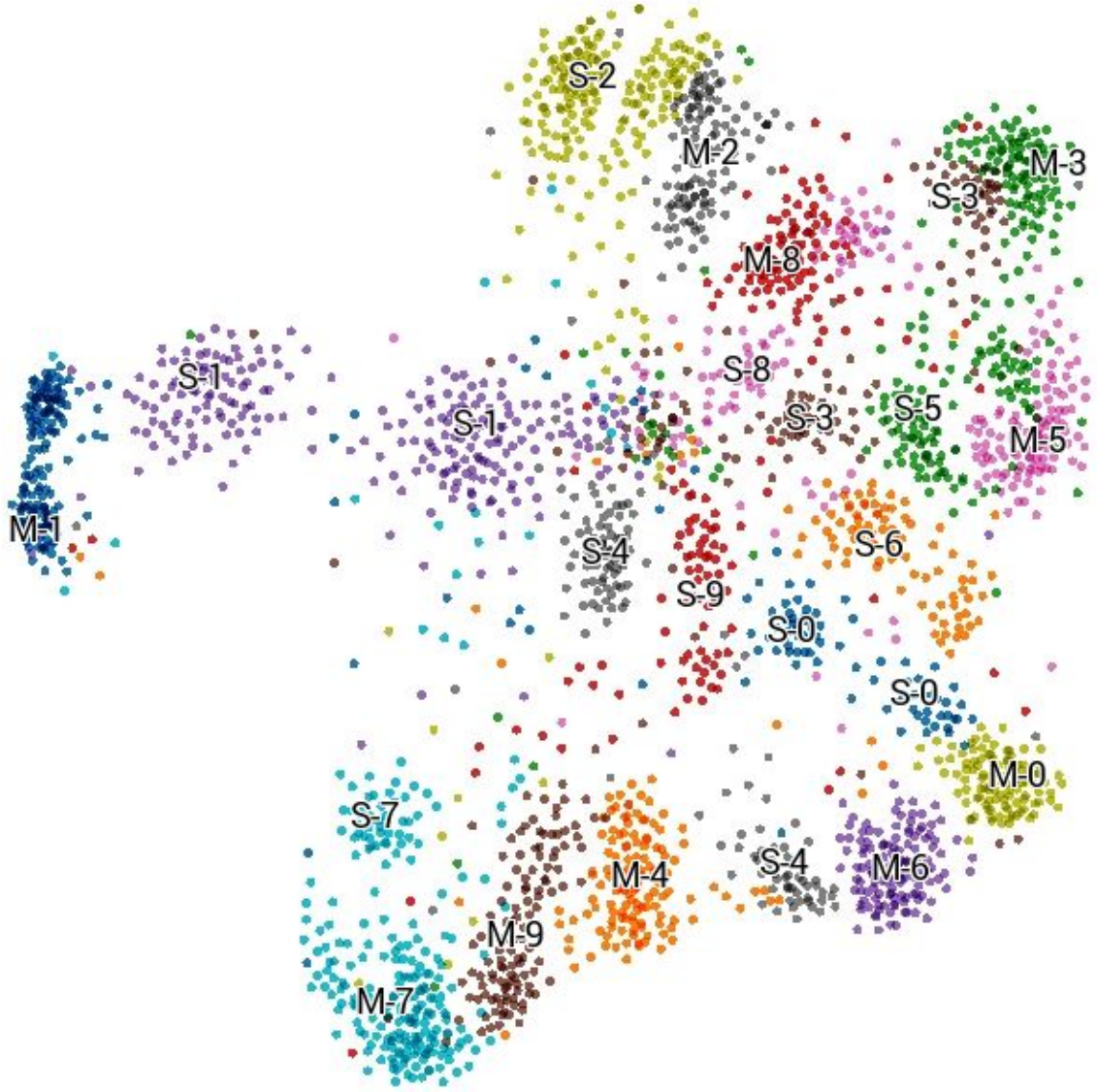
To find the good learning rate I've used Tensorboard with a kind of a Grid Search, where I've been training my model and saving results to the Tensorboard with respective hyperparameter values. Later I've checked the graphs of accuracy and choose the one that suited the most (which was 4 for number of time we repeat the third step, 0.0002 for learning rate and 0.0005 weight decay in Adam Optimizer).
During the tuning it was also found out that a randomness of data loaders can lower the final accuracy. To lower the probability of that happening we learn the network only on source a couple of times before doing the training

Visualization of Latent Space

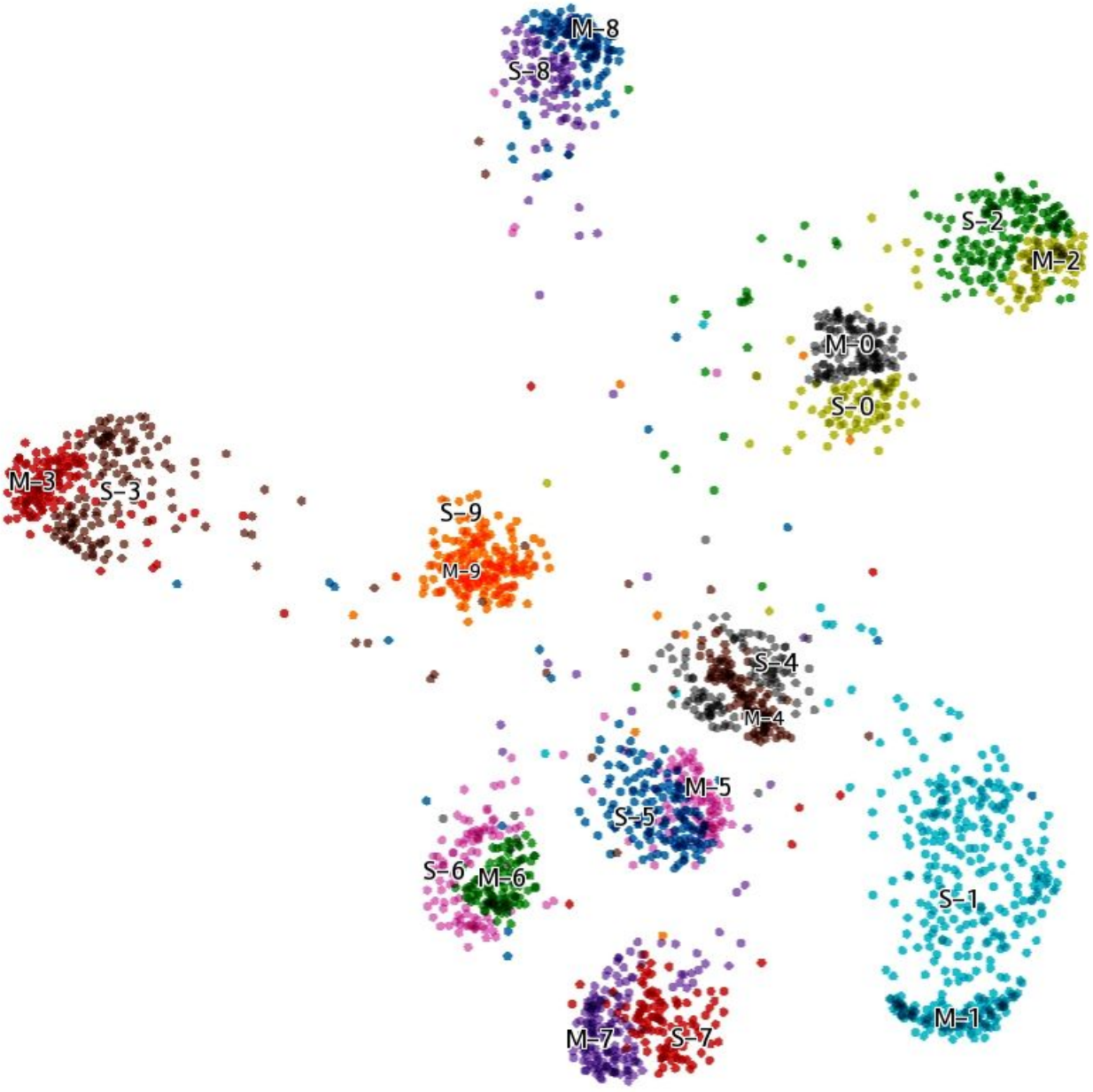
Let's visualize the Latent Space (features we get from Encoder). For visualization I've been using Tensorboard embeddings and T-SNE. I will use notation M-i, S-j, where M - MNIST images and S - SVHN images with i, j meaning the number they represent

1. Here we can see the *Latent Space without any training* (with random weights). There is no clear structure and it is only possible to see MNIST images somewhat clearly, the SVHN images doesn't even make clusters (They are in the left upper corner and it is impossible to mark any of them). It is also possible to see the domain gap, as the clusters of SVHN and MNIST are far from each other.

2. *After doing the training on Source Images* we get the following Latent Space.
It is now possible to also see cluster of SVHN images, but they are still can be far from their respective classes from MNIST domain.



3. Finally, *after performing the domain adaptation* we get the following Latent Space.
The same classes now form cluster despite being in different domains, which means that we solved the domain gap between MNIST and SVHN.

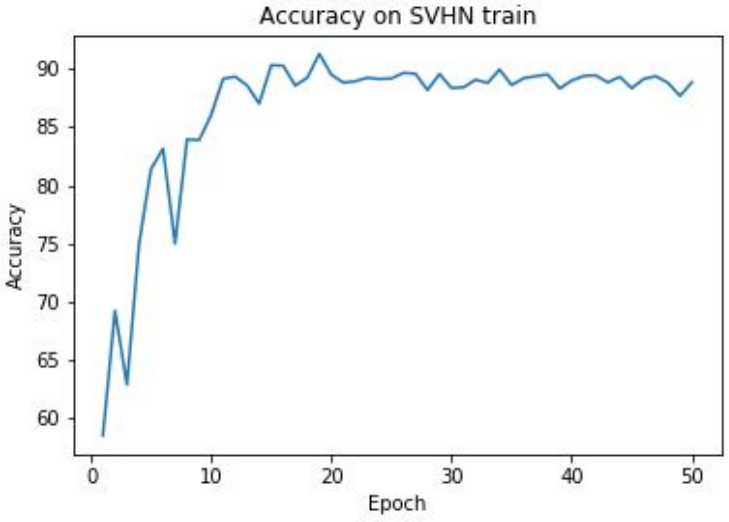


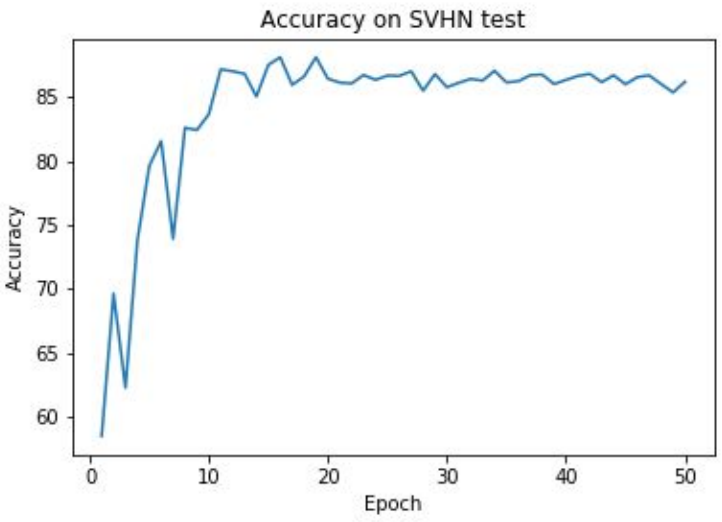
Evaluation

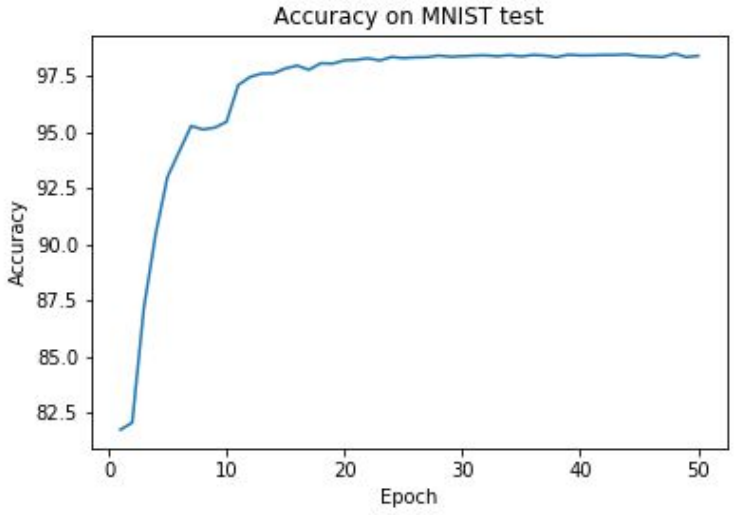
The code of the run can be found here. It also saves information about runs (accuracy on datasets, graphics, latent space) that later can be viewed in tensorboard, so if you run it locally you can see how the process of training is going.
The Final Accuracy after 50 Epochs we got:

	SVHN Train	SVHN Test	MNIST Test
Accuracy (%)	88.82	86.23	98.4

We can plot the graph and see how the accuracy changes over time for all 50 epochs.







Ablation Study

For ablation study we can try to remove some parts of the training process and architecture and see how the model performs.
I've come up with the following modifications:

- Removing last two steps of training (MCD-1Step)
- Removing second step of training (MCD-13Step)
- Removing third step of training (MCD-12Step)
- Removing cross entropy loss in second loss (MCD-OnlyDis)

After training we get the following results:

Model	SVHN Train	SVHN Test	MNIST Test
Original MCD	88.82	86.23	**98.4**
MCD-1Step	**99.51**	**92.82**	64.34
MCD-13Step	86.16	84.72	96.94
MCD-12Step	94.18	88.62	66.87
MCD-OnlyDis	86.09	85.24	88.68

As we can see, removing any of the steps, changing the process of training degrades the performance of the model, therefore we can make a conclusion that all components of training a crucial for our model. We can also notice that MCD-1Step performs better all models predicting the SVHN dataset. This is merely because removing step 2 and step 3 basically transforms the architecture into one encoder and two classifier that simply learn on the source domain. We can also notice that removing the original loss in second step (which was described in the training process) harm the accuracy overall.

Conclusion

From the results it is evident that not only it is possible to solve domain gap between SVHN and MNIST dataset (though, the accuracy on the domain may be lower), but also that this can be done by using a very simple architecture (simple relative to other solutions for unsupervised domain adaptations) of just three networks. It's also evident that the simple discrepancy function suggested by authors of the paper works good for images, but it is still recommended to use problem-specific function to get a better performance.

The results however, still may be improved by implementing another discrepancy function, which has already been shown to provide a better results (For example, using a Sliced Wasserstein Distance SWD[5]). This improvement is possible because we can measure the disagreement between two classifiers (which is the idea behind the method) in many ways and capture the difference in their output using other metrics, which may describe it more thoroughly.

References

1. Saito, K.; Watanabe, K.; Ushiku, Y.; Harada, T. Maximum Classifier Discrepancy for Unsupervised Domain Adaptation. *arXiv 2017, arXiv:1712.02560.*
2. Bousmalis, K.; Silberman, N.; Dohan, D.; Erhan, D.; Krishnan, D. Unsupervised Pixel-Level Domain Adaptation with Generative Adversarial Networks. *arXiv 2016, arXiv:1612.05424.*
3. Bekkouch, I. E. I.; Youssry, Y.; Gafarov, R.; Khan, A.;Khattak, A. M. Triplet Loss Network for Unsupervised Domain Adaptation. *MDPI 2019*. [Ref]
4. Lv, F.; Zhu, J.; Yang, G.; Duan, L. TarGAN: Generating target data with class labels for unsupervised domain adaptation. *Knowl.-Based Syst. 2019*. [CrossRef]

5. Lee, C.; Batra T.; Baig, M. H.; Ulbricht, D. Sliced Wasserstein Discrepancy for Unsupervised Domain Adaptation. *arXiv 2019, arXiv:1903.04064.*
6. “MCD DA” Source code. https://github.com/mil-tokyo/MCD_DA