Transfer Learning Results

This is a recurring report of the transfer learning experiment, ten in total, including **TCA, JDA, Baseline: ResNet50, Baseline: Digit Network, Self-ensemble Visual Domain Adapt Master, DANN, ADA, MCD_UDA, MADA and ResNet+Wasserstein.**

~~The above line is just to support the display of mathematical formulas(may be it doesn't appear)~~, it is recommended to open with **Chrome**.

Data set (downloaded from all over, and integrate by my. The extraction code: zhgy)

You can use `python main.py --model='xxx' --source='xxx' --target='xxx'` to run them. And other parser you can find in `python main.py --h`

黄晨晰 Chenxi Huang

Computer and science majors

Email: mrsempress98@gmail.com / hcx_98@163.com

Web: http://mrsempress.top

Preface

The machine parameters used are as follows:

CPU	内存	显卡
i7-5930K (6/12)	125G	GTX 1080 * 4

I also read codes of other algorithm. And some codes try them myself, others are not.

Each one gives ways to execution, as well as attention. At the same time, while giving the results, the analysis was carried out, including the comparison with the previous algorithm, paper, and the analysis of different data sets (You can see the part of `Compare`).

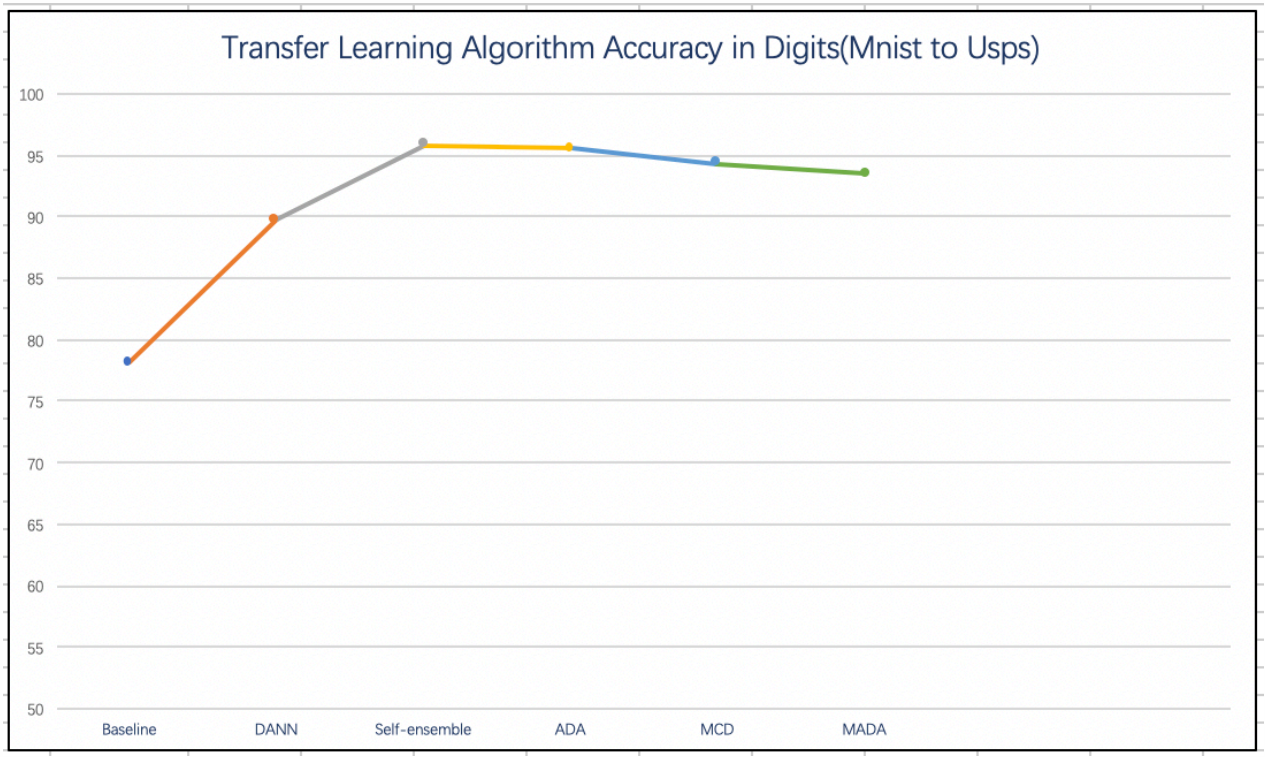
Attention:

1. **[Record]** Due to space limitations, the experimental group released by each algorithm is limited. If you want to see more results of each group, see the `./results_image/*.png` .
2. **[Use log]** Since the way the data was previously recorded is directly output, then the screenshot shows. But the later the algorithm, the larger the amount of data, the more iterations, so it is not convenient for screenshots. I want to record by log, but it takes time to retest it. So I only give the code, if you want to get the data, please run it again. And you can find the log in `./results/model_name/source_to_target/model_name.csv`.
3. **[Input]** If you still have problems with the input, you can open the `main.py` file and use a basic example on each model.
4. **[Gpu]** Note that the default value of my gpu here is `3`, if you use the machine GPU does not have 4 blocks, you need to specify when entering the command.

Summary

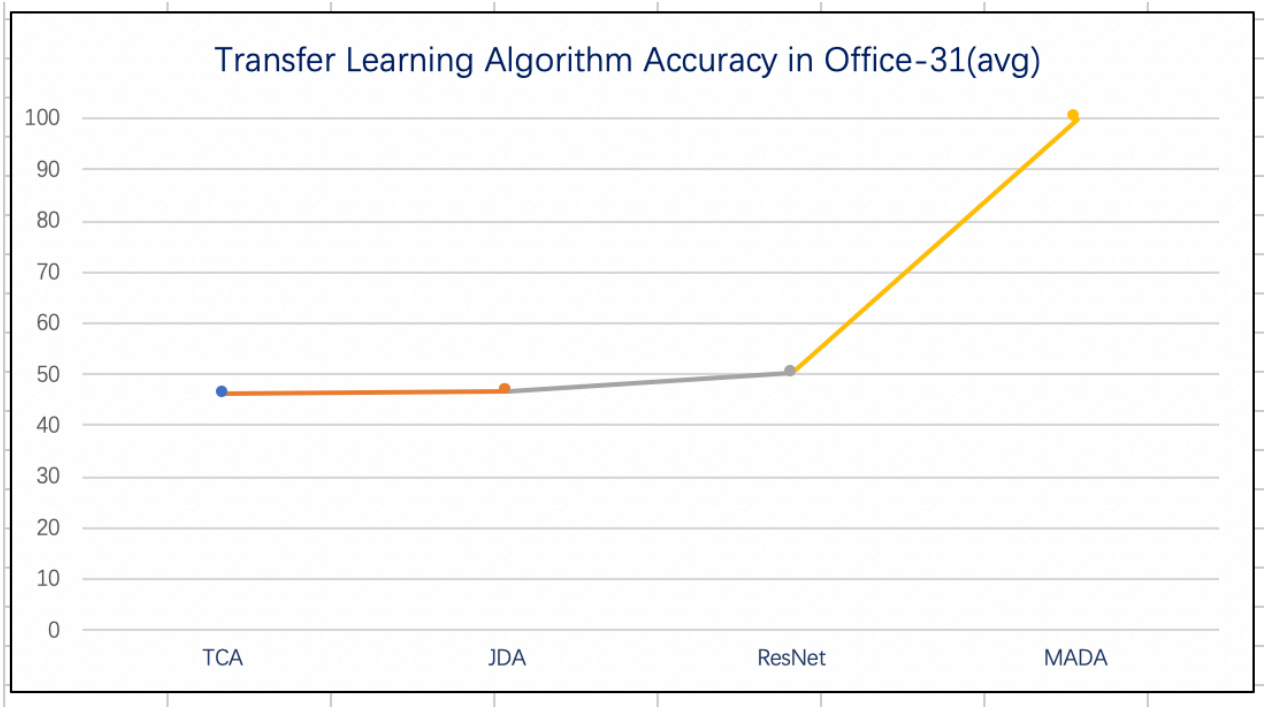
Transfer Learning Algorithm Accuracy in Digits(Mnist to Usps):

(Baseline's network is only used for specific MNIST to USPS according to Appendix D of "Self-ensembling Visual Domain Adapt Master", so the last comparison used Mnist to USPS without average.)



Transfer Learning Algorithm Accuracy in Office-31(avg):

(Beacause Office-31 more large, fewer days I have to experiment than others and more kinds of experiments I do, so only test four algorithm in Office-31.)



The experiment is as same trend as the theory. And details show in `Compare`, you can see the results of each set of comparisons

Non-deep transfer learning

TCA

Download the data set of Office-31, the input format is `.mat` (if it is not, please use Matlab to convert it).

~~You can see the result with the command: `python TCA.py`.~~

Since I integrated the code, please use `python main.py --model='TCA'` and you can set other parameters.

The results is as follows:

(**The accuracy is up to 91.08**%)

`Webcam to Dslr`

```
[root@71b4a3d0c1b1:/home/huangchenxi# python TCA.py
TCA:  data = Office-31 lambda = 1 src: data/Office-31/webcam.mat tar: data/Office-31/dslr.mat
0.910828025477707
root@71b4a3d0c1b1:/home/huangchenxi#
```

After the unified framework, the test by default (changed a set of source and target domains, but the results are still in the normal range):

`Amazon to Webcam`

```
root at 60a5497e4409 in /home/huangchenxi/transfer-learning/transfer-learning
$ python main.py
TCA:  data = Office-31 lambda = 1 src: data/Office-31/amazon.mat tar: data/Offi
ce-31/webcam.mat
0.4
```

Compare

Since the paper's experiments were based on `Cross-domain WiFi Localization` and `Cross-domain Text Classification`, so I used another paper "Transfer Feature Learning with Joint Distribution Adaptation" to verify:

(Because I **add fine-tune**, so the result will **be better than paper**. You can see more detailed information in `/results_image/TCA-x-y.png` .)

Dataset	**In paper(%)**	**In my test (%)**
C->A	44.47	**45.62**
C->W	34.24	**39.32**
C->D	43.31	**45.86**
A->C	37.58	**42.03**

A->W	33.90	**40.00**
A->D	26.11	**35.67**
W->C	29.83	**31.52**
W->A	30.27	**30.48**
W->D	87.26	**91.08**
D->C	28.50	**32.95**
D->A	27.56	**32.78**
D->W	85.42	**87.46**

JDA

I used the data set provided by the author Long Mingsheng source code.

10 common categories are picked out from 4 datasets of object images: `caltech, amazon, webcam, and dslr`. Then follow the previously reported protocols for preparing features, i.e., **extracting SURF features** and quantizing them into an 800-bin histogram with codebooks computed via K-means on a subset of images from amazon. The file name: `*_SURF_L10.mat` is about features and labels.

I write two version of the JDA.

Refer to Long's

One is the code of *Long Mingsheng*'s' MATLAB version of the code. According to the structure of the paper, after the reappearance, the results are as follows:

(**The best accuracy is 79.62%,** about webcam and dslr; but **the worst accuracy is 24.76%**, about webcam and caltech. And except `A->W` and `A->D`, all are better than 1 NN.)

```
root@71b4a3d0c1b1:/home/huangchenxi/test# python JDA.py
[src is ./data/JDA/Caltech10_SURF_L10.mat, tar is ./data/JDA/amazon_SURF_L10.mat
[NN =  0.24217118997912318
 ==============================Iteration [0]==============================
[JDA iteration [1/10]: Acc: 0.2777
 ==============================Iteration [1]==============================
 JDA iteration [2/10]: Acc: 0.2777
 ==============================Iteration [2]==============================
 JDA iteration [3/10]: Acc: 0.2777
 ==============================Iteration [3]==============================
 JDA iteration [4/10]: Acc: 0.2777
 ==============================Iteration [4]==============================
 JDA iteration [5/10]: Acc: 0.2777
 ==============================Iteration [5]==============================
 JDA iteration [6/10]: Acc: 0.2777
 ==============================Iteration [6]==============================
 JDA iteration [7/10]: Acc: 0.2777
 ==============================Iteration [7]==============================
 JDA iteration [8/10]: Acc: 0.2777
 ==============================Iteration [8]==============================
 JDA iteration [9/10]: Acc: 0.2777
 ==============================Iteration [9]==============================
 JDA iteration [10/10]: Acc: 0.2777



 src is ./data/JDA/Caltech10_SURF_L10.mat, tar is ./data/JDA/webcam_SURF_L10.mat
 NN =  0.28135593220338984
 ==============================Iteration [0]==============================
 JDA iteration [1/10]: Acc: 0.2847
 ==============================Iteration [1]==============================
 JDA iteration [2/10]: Acc: 0.2847
 ==============================Iteration [2]==============================
 JDA iteration [3/10]: Acc: 0.2847
 ==============================Iteration [3]==============================
 JDA iteration [4/10]: Acc: 0.2847
 ==============================Iteration [4]==============================
 JDA iteration [5/10]: Acc: 0.2847
 ==============================Iteration [5]==============================
 JDA iteration [6/10]: Acc: 0.2847
 ==============================Iteration [6]==============================
 JDA iteration [7/10]: Acc: 0.2847
 ==============================Iteration [7]==============================
 JDA iteration [8/10]: Acc: 0.2847
 ==============================Iteration [8]==============================
 JDA iteration [9/10]: Acc: 0.2847
 ==============================Iteration [9]==============================
 JDA iteration [10/10]: Acc: 0.2847



 src is ./data/JDA/Caltech10_SURF_L10.mat, tar is ./data/JDA/dslr_SURF_L10.mat
 NN =  0.28662420382165604
 ==============================Iteration [0]==============================
 JDA iteration [1/10]: Acc: 0.3694
 ==============================Iteration [1]==============================
 JDA iteration [2/10]: Acc: 0.3694
 ==============================Iteration [2]==============================
 JDA iteration [3/10]: Acc: 0.3694
 ==============================Iteration [3]==============================
 JDA iteration [4/10]: Acc: 0.3694
 ==============================Iteration [4]==============================
 JDA iteration [5/10]: Acc: 0.3694
 ==============================Iteration [5]==============================
 JDA iteration [6/10]: Acc: 0.3694
 ==============================Iteration [6]==============================
 JDA iteration [7/10]: Acc: 0.3694
 ==============================Iteration [7]==============================
 JDA iteration [8/10]: Acc: 0.3694
 ==============================Iteration [8]==============================
 JDA iteration [9/10]: Acc: 0.3694
 ==============================Iteration [9]==============================
 JDA iteration [10/10]: Acc: 0.3694
```

```
src is ./data/JDA/amazon_SURF_L10.mat, tar is ./data/JDA/Caltech10_SURF_L10.mat
NN =  0.15761353517364202
==============================Iteration [0]==============================
JDA iteration [1/10]: Acc: 0.2680
==============================Iteration [1]==============================
JDA iteration [2/10]: Acc: 0.2680
==============================Iteration [2]==============================
JDA iteration [3/10]: Acc: 0.2680
==============================Iteration [3]==============================
JDA iteration [4/10]: Acc: 0.2680
==============================Iteration [4]==============================
JDA iteration [5/10]: Acc: 0.2680
==============================Iteration [5]==============================
JDA iteration [6/10]: Acc: 0.2680
==============================Iteration [6]==============================
JDA iteration [7/10]: Acc: 0.2680
==============================Iteration [7]==============================
JDA iteration [8/10]: Acc: 0.2680
==============================Iteration [8]==============================
JDA iteration [9/10]: Acc: 0.2680
==============================Iteration [9]==============================
JDA iteration [10/10]: Acc: 0.2680



src is ./data/JDA/amazon_SURF_L10.mat, tar is ./data/JDA/webcam_SURF_L10.mat
NN =  0.3152542372881356
==============================Iteration [0]==============================
JDA iteration [1/10]: Acc: 0.3085
==============================Iteration [1]==============================
JDA iteration [2/10]: Acc: 0.3085
==============================Iteration [2]==============================
JDA iteration [3/10]: Acc: 0.3085
==============================Iteration [3]==============================
JDA iteration [4/10]: Acc: 0.3085
==============================Iteration [4]==============================
JDA iteration [5/10]: Acc: 0.3085
==============================Iteration [5]==============================
JDA iteration [6/10]: Acc: 0.3085
==============================Iteration [6]==============================
JDA iteration [7/10]: Acc: 0.3085
==============================Iteration [7]==============================
JDA iteration [8/10]: Acc: 0.3085
==============================Iteration [8]==============================
JDA iteration [9/10]: Acc: 0.3085
==============================Iteration [9]==============================
JDA iteration [10/10]: Acc: 0.3085



src is ./data/JDA/amazon_SURF_L10.mat, tar is ./data/JDA/dslr_SURF_L10.mat
NN =  0.29936305732484075
==============================Iteration [0]==============================
JDA iteration [1/10]: Acc: 0.2866
==============================Iteration [1]==============================
JDA iteration [2/10]: Acc: 0.2866
==============================Iteration [2]==============================
JDA iteration [3/10]: Acc: 0.2866
==============================Iteration [3]==============================
JDA iteration [4/10]: Acc: 0.2866
==============================Iteration [4]==============================
JDA iteration [5/10]: Acc: 0.2866
==============================Iteration [5]==============================
JDA iteration [6/10]: Acc: 0.2866
==============================Iteration [6]==============================
JDA iteration [7/10]: Acc: 0.2866
==============================Iteration [7]==============================
JDA iteration [8/10]: Acc: 0.2866
==============================Iteration [8]==============================
JDA iteration [9/10]: Acc: 0.2866
==============================Iteration [9]==============================
JDA iteration [10/10]: Acc: 0.2866
```

```
src is ./data/JDA/webcam_SURF_L10.mat, tar is ./data/JDA/Caltech10_SURF_L10.mat
NN =  0.1772039180765806
==============================Iteration [0]==============================
JDA iteration [1/10]: Acc: 0.2476
==============================Iteration [1]==============================
JDA iteration [2/10]: Acc: 0.2476
==============================Iteration [2]==============================
JDA iteration [3/10]: Acc: 0.2476
==============================Iteration [3]==============================
JDA iteration [4/10]: Acc: 0.2476
==============================Iteration [4]==============================
JDA iteration [5/10]: Acc: 0.2476
==============================Iteration [5]==============================
JDA iteration [6/10]: Acc: 0.2476
==============================Iteration [6]==============================
JDA iteration [7/10]: Acc: 0.2476
==============================Iteration [7]==============================
JDA iteration [8/10]: Acc: 0.2476
==============================Iteration [8]==============================
JDA iteration [9/10]: Acc: 0.2476
==============================Iteration [9]==============================
JDA iteration [10/10]: Acc: 0.2476



src is ./data/JDA/webcam_SURF_L10.mat, tar is ./data/JDA/amazon_SURF_L10.mat
NN =  0.17118997912317327
==============================Iteration [0]==============================
JDA iteration [1/10]: Acc: 0.2777
==============================Iteration [1]==============================
JDA iteration [2/10]: Acc: 0.2777
==============================Iteration [2]==============================
JDA iteration [3/10]: Acc: 0.2777
==============================Iteration [3]==============================
JDA iteration [4/10]: Acc: 0.2777
==============================Iteration [4]==============================
JDA iteration [5/10]: Acc: 0.2777
==============================Iteration [5]==============================
JDA iteration [6/10]: Acc: 0.2777
==============================Iteration [6]==============================
JDA iteration [7/10]: Acc: 0.2777
==============================Iteration [7]==============================
JDA iteration [8/10]: Acc: 0.2777
==============================Iteration [8]==============================
JDA iteration [9/10]: Acc: 0.2777
==============================Iteration [9]==============================
JDA iteration [10/10]: Acc: 0.2777



src is ./data/JDA/webcam_SURF_L10.mat, tar is ./data/JDA/dslr_SURF_L10.mat
NN =  0.6242038216560509
==============================Iteration [0]==============================
JDA iteration [1/10]: Acc: 0.7962
==============================Iteration [1]==============================
JDA iteration [2/10]: Acc: 0.7962
==============================Iteration [2]==============================
JDA iteration [3/10]: Acc: 0.7962
==============================Iteration [3]==============================
JDA iteration [4/10]: Acc: 0.7962
==============================Iteration [4]==============================
JDA iteration [5/10]: Acc: 0.7962
==============================Iteration [5]==============================
JDA iteration [6/10]: Acc: 0.7962
==============================Iteration [6]==============================
JDA iteration [7/10]: Acc: 0.7962
==============================Iteration [7]==============================
JDA iteration [8/10]: Acc: 0.7962
==============================Iteration [8]==============================
JDA iteration [9/10]: Acc: 0.7962
==============================Iteration [9]==============================
JDA iteration [10/10]: Acc: 0.7962
```

```
src is ./data/JDA/dslr_SURF_L10.mat, tar is ./data/JDA/Caltech10_SURF_L10.mat
NN =  0.08370436331255565
==============================Iteration [0]==============================
JDA iteration [1/10]: Acc: 0.2850
==============================Iteration [1]==============================
JDA iteration [2/10]: Acc: 0.2850
==============================Iteration [2]==============================
JDA iteration [3/10]: Acc: 0.2850
==============================Iteration [3]==============================
JDA iteration [4/10]: Acc: 0.2850
==============================Iteration [4]==============================
JDA iteration [5/10]: Acc: 0.2850
==============================Iteration [5]==============================
JDA iteration [6/10]: Acc: 0.2850
==============================Iteration [6]==============================
JDA iteration [7/10]: Acc: 0.2850
==============================Iteration [7]==============================
JDA iteration [8/10]: Acc: 0.2850
==============================Iteration [8]==============================
nvidiJDA iteration [9/10]: Acc: 0.2850
==============================Iteration [9]==============================
JDA iteration [10/10]: Acc: 0.2850



src is ./data/JDA/dslr_SURF_L10.mat, tar is ./data/JDA/amazon_SURF_L10.mat
NN =  0.10438413361169102
==============================Iteration [0]==============================
JDA iteration [1/10]: Acc: 0.2766
==============================Iteration [1]==============================
JDA iteration [2/10]: Acc: 0.2766
==============================Iteration [2]==============================
JDA iteration [3/10]: Acc: 0.2766
==============================Iteration [3]==============================
JDA iteration [4/10]: Acc: 0.2766
==============================Iteration [4]==============================
JDA iteration [5/10]: Acc: 0.2766
==============================Iteration [5]==============================
JDA iteration [6/10]: Acc: 0.2766
==============================Iteration [6]==============================
JDA iteration [7/10]: Acc: 0.2766
==============================Iteration [7]==============================
JDA iteration [8/10]: Acc: 0.2766
==============================Iteration [8]==============================
JDA iteration [9/10]: Acc: 0.2766
==============================Iteration [9]==============================
JDA iteration [10/10]: Acc: 0.2766



src is ./data/JDA/dslr_SURF_L10.mat, tar is ./data/JDA/webcam_SURF_L10.mat
NN =  0.10847457627118644
==============================Iteration [0]==============================
JDA iteration [1/10]: Acc: 0.7356
==============================Iteration [1]==============================
JDA iteration [2/10]: Acc: 0.7356
==============================Iteration [2]==============================
JDA iteration [3/10]: Acc: 0.7356
==============================Iteration [3]==============================
JDA iteration [4/10]: Acc: 0.7356
==============================Iteration [4]==============================
JDA iteration [5/10]: Acc: 0.7356
==============================Iteration [5]==============================
JDA iteration [6/10]: Acc: 0.7356
==============================Iteration [6]==============================
JDA iteration [7/10]: Acc: 0.7356
==============================Iteration [7]==============================
JDA iteration [8/10]: Acc: 0.7356
==============================Iteration [8]==============================
JDA iteration [9/10]: Acc: 0.7356
==============================Iteration [9]==============================
JDA iteration [10/10]: Acc: 0.7356
```

```
root@71b4a3d0c1b1:/home/huangchenxi/test#
```

Compare

First, I think the low accuracy due to my programming, but after I see the paper result, it may because the data or algorithm.

The best in paper is `D->W = 89.49`, and in my test is `W->D = 79.62`(and `D->W =73.85%`).

The worst in paper is `W->C = 31.17%`, and in my test is `W->C = 24.76%`.

Table 3. Accuracy (%) on 4 types of cross-domain image datasets.

Dataset	Standard Learning		Transfer Learning			
	NN	PCA	GFK	TCA	TSL	JDA
USPS vs MNIST	44.7	44.95	46.45	51.05	53.75	**59.65**
MNIST vs USPS	65.94	66.22	67.22	56.28	66.06	**67.28**
COIL1 vs COIL2	83.61	84.72	72.50	88.47	88.06	**89.31**
COIL2 vs COIL1	82.78	84.03	74.17	85.83	87.92	**88.47**
PIE1 vs PIE2 (1)	26.09	24.80	26.15	40.76	44.08	**58.81**
PIE1 vs PIE3 (2)	26.59	25.18	27.27	41.79	47.49	**54.23**
PIE1 vs PIE4 (3)	30.67	29.26	31.15	59.63	62.78	**84.50**
PIE1 vs PIE5 (4)	16.67	16.30	17.59	29.35	36.15	**49.75**
PIE2 vs PIE1 (5)	24.49	24.22	25.24	41.81	46.28	**57.62**
PIE2 vs PIE3 (6)	46.63	45.53	47.37	51.47	57.60	**62.93**
PIE2 vs PIE4 (7)	54.07	53.35	54.25	64.73	71.43	**75.82**
PIE2 vs PIE5 (8)	26.53	25.43	27.08	33.70	35.66	**39.89**
PIE3 vs PIE1 (9)	21.37	20.95	21.82	34.69	36.94	**50.96**
PIE3 vs PIE2 (10)	41.01	40.45	43.16	47.70	47.02	**57.95**
PIE3 vs PIE4 (11)	46.53	46.14	46.41	56.23	59.45	**68.45**
PIE3 vs PIE5 (12)	26.23	25.31	26.78	33.15	36.34	**39.95**
PIE4 vs PIE1 (13)	32.95	31.96	34.24	55.64	63.66	**80.58**
PIE4 vs PIE2 (14)	62.68	60.96	62.92	67.83	72.68	**82.63**
PIE4 vs PIE3 (15)	73.22	72.18	73.35	75.86	83.52	**87.25**
PIE4 vs PIE5 (16)	37.19	35.11	37.38	40.26	44.79	**54.66**
PIE5 vs PIE1 (17)	18.49	18.85	20.35	26.98	33.28	**46.46**
PIE5 vs PIE2 (18)	24.19	23.39	24.62	29.90	34.13	**42.05**
PIE5 vs PIE3 (19)	28.31	27.21	28.49	29.90	36.58	**53.31**
PIE5 vs PIE4 (20)	31.24	30.34	31.33	33.64	38.75	**57.01**
C→A (1)	23.70	36.95	41.02	38.20	44.47	**44.78**
C→W (2)	25.76	32.54	40.68	38.64	34.24	**41.69**
C→D (3)	25.48	38.22	38.85	41.40	43.31	**45.22**
A→C (4)	26.00	34.73	**40.25**	37.76	37.58	39.36
A→W (5)	29.83	35.59	**38.98**	37.63	33.90	37.97
A→D (6)	25.48	27.39	36.31	33.12	26.11	**39.49**
W→C (7)	19.86	26.36	30.72	29.30	29.83	**31.17**
W→A (8)	22.96	31.00	29.75	30.06	30.27	**32.78**
W→D (9)	59.24	77.07	80.89	87.26	87.26	**89.17**
D→C (10)	26.27	29.65	30.28	31.70	28.50	**31.52**
D→A (11)	28.50	32.05	32.05	32.15	27.56	**33.09**
D→W (12)	63.39	75.93	75.59	86.10	85.42	**89.49**
Average	37.46	39.84	41.19	47.22	49.80	**57.37**

	In my test(NN)	In paper(NN)	In my test(JDA)	In paper(JDA)
C->A	24.22	23.70	27.77	**44.78**
C->W	28.14	25.76	28.47	**41.69**
C->D	28.66	25.48	36.94	**45.22**
A->C	15.76	26.00	26.80	**39.36**
A->W	31.53	29.83	30.85	**37.97**
A->D	29.94	25.48	28.66	**39.49**
W->C	17.72	19.86	24.76	**31.17**
W->A	17.12	22.96	27.77	**32.78**
W->D	62.42	59.24	79.62	**89.17**
D->C	8.37	26.27	28.50	**31.52**
D->A	10.44	28.50	27.66	**33.09**
D->W	10.85	63.39	73.56	**89.49**

It seems that this reappearance is not good enough, and the gap is a bit big. The first reason is that the data itself is processed and loses some information; Another reason is that matlab is converted to python, and may not use a better function.

Refer to Wang's

The other is *Wang Jindong*'s Python version of the code. He changed the structure of the code, reduced the function call, converted to a loop, used the data already obtained in the previous loop, avoiding the recalculation, and add fine-tune. Therefore, the error is also reduced and **is similiar with that in paper**. The results are as follows:

(**The accuracy is 46.56**% about caltech and amazon. And in last version is only 27.78%, in paper is 44.78%)

```
[root@71b4a3d0c1b1:/home/huangchenxi# python JDA_2.py
JDA:  data = Office-31 lambda = 1 src: data/Office-31/caltech.mat tar: data/Office-31/amazon.mat
JDA iteration [1/10]: Acc: 0.4666
JDA iteration [2/10]: Acc: 0.4593
JDA iteration [3/10]: Acc: 0.4656
JDA iteration [4/10]: Acc: 0.4624
JDA iteration [5/10]: Acc: 0.4666
JDA iteration [6/10]: Acc: 0.4666
JDA iteration [7/10]: Acc: 0.4656
JDA iteration [8/10]: Acc: 0.4656
JDA iteration [9/10]: Acc: 0.4656
JDA iteration [10/10]: Acc: 0.4656
0.46555323590814196
root@71b4a3d0c1b1:/home/huangchenxi#
```

Compare

And the summary in tables:

	In my test	In paper
C->A	**46.56**	44.78
C->W	39.32	**41.69**
C->D	**47.13**	45.22
A->C	**40.16**	39.36
A->W	**38.30**	37.97
A->D	36.94	**39.49**
W->C	**32.41**	31.17
W->A	**32.99**	32.78
W->D	**92.36**	89.17
D->C	**32.06**	31.52
D->A	**33.82**	33.09
D->W	**89.83**	89.49

Compare with TCA

It can be seen that **JDA is basically better than TCA** (it is good in the paper, probably because the degree of fine-tune is different).

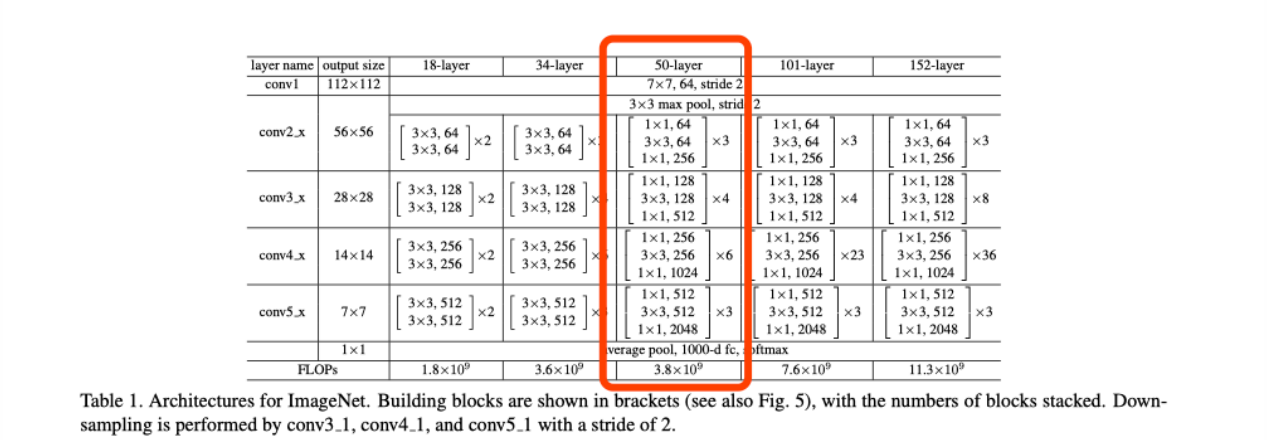
	In my test(JDA)	In paper(JDA)	In my test(TCA)	In paper(TCA)
C->A	**46.56**	44.78	45.62	44.47
C->W	39.32	**41.69**	39.32	34.24
C->D	**47.13**	45.22	45.86	43.31
A->C	40.16	39.36	**42.03**	37.58
A->W	38.30	37.97	**40.00**	33.90
A->D	36.94	**39.49**	35.67	26.11
W->C	**32.41**	31.17	31.52	29.83
W->A	**32.99**	32.78	30.48	30.27
W->D	**92.36**	89.17	91.08	87.26
D->C	32.06	31.52	**32.95**	28.50
D->A	**33.82**	33.09	32.78	27.56
D->W	**89.83**	89.49	87.46	85.42

Deep transfer learning

ResNet50

The datasets are the same.

The structure of ResNet50 is as follows:



layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix} \times$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix} \times 3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix} \times$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix} \times 8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix} \times$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix} \times 6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix} \times 23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix} \times 36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix} \times$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix} \times 3$
	1×1	verage pool, 1000-d fc, oftmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Table 1. Architectures for ImageNet. Building blocks are shown in brackets (see also Fig. 5), with the numbers of blocks stacked. Down-sampling is performed by conv3_1, conv4_1, and conv5_1 with a stride of 2.

We can build it layer by layer according to its structure; it can also be used `torchvision.models.resnet50(pretrained=False, progress=True, **kwargs)`. I choose the latter, and use the pertained parameters (the parameters are in `./codes/ResNet50/resnet50-19c8e357.pth`, you can use it).

But the images is so large and I first set the `batches_size = 256`. So the docker stop, and then I forced the container size as 8 GB and decreased the batches size to 100.

And the part of the result as follows:

```
root@a28dd28ff56a:/home/huangchenxi/transfer-learning/ResNet50# python Fine
-tune.py
TCA:  data = Office-31 lambda = 1 src: webcam.mat tar: dslr.mat
3288 822
Learning rate: 0.00010000
Learning rate: 0.00100000
Epoch: [01/100]---src, loss: 1.926839, acc: 0.5620
Epoch: [01/100]---val, loss: 0.978189, acc: 0.6873
Epoch: [01/100]---tar, loss: 0.990008, acc: 0.6854

Learning rate: 0.00010000
Learning rate: 0.00100000
Epoch: [02/100]---src, loss: 0.746554, acc: 0.7509
Epoch: [02/100]---val, loss: 0.626346, acc: 0.7032
Epoch: [02/100]---tar, loss: 0.631439, acc: 0.6981

Learning rate: 0.00010000
Learning rate: 0.00100000
Epoch: [03/100]---src, loss: 0.503822, acc: 0.7859
Epoch: [03/100]---val, loss: 0.416240, acc: 0.8698
Epoch: [03/100]---tar, loss: 0.402277, acc: 0.8832

Learning rate: 0.00010000
Learning rate: 0.00100000
Epoch: [04/100]---src, loss: 0.364801, acc: 0.8863
Epoch: [04/100]---val, loss: 0.316410, acc: 0.9136
Epoch: [04/100]---tar, loss: 0.302642, acc: 0.9358

Learning rate: 0.00010000
Learning rate: 0.00100000
Epoch: [05/100]---src, loss: 0.287266, acc: 0.9389
Epoch: [05/100]---val, loss: 0.273787, acc: 0.9294
Epoch: [05/100]---tar, loss: 0.244674, acc: 0.9564

Learning rate: 0.00010000
Learning rate: 0.00100000
Epoch: [06/100]---src, loss: 0.244297, acc: 0.9519
Epoch: [06/100]---val, loss: 0.230979, acc: 0.9574
Epoch: [06/100]---tar, loss: 0.212978, acc: 0.9676
```

Obviously, the accuracy is increase. And only after 4 iteration, the target accuracy can **be up to 93.58**%. So, the network is good. And you can set up the batches size as 10, the loss can even lower.

Compare

Since this part directly calls the function, it does not look at the paper that presented it. So no comparison with paper is made. But with the accuracy of non-deep networks, ResNet has a good performance.

	In my test
C->A	49.27
C->W	38.51
C->D	49.39
A->C	39.33

A->W	38.24
A->D	38.36
W->C	38.30
W->A	38.51
W->D	96.76
D->C	38.39
D->A	38.39
D->W	98.42

With Wasserstein

Replaced the metrics and replaced MMD with Wasserstein.

Compared with the original form of the original GAN, WGAN only changed four points:

- The last layer of the discriminator removes sigmoid
- The loss of the generator and discriminator does not take the log
- Each time the parameters of the discriminator are updated, their absolute values are truncated to no more than a fixed constant c
- Do not use momentum-based optimization algorithms (including momentum and Adam), recommend RMSProp, SGD.

The results as follows:

```
$ python Baseline.py
Epoch: [01/25]---src, loss: 0.342123, acc: 0.9092
Epoch: [01/25]---val, loss: 0.333888, acc: 0.9200
Epoch: [01/25]---tar, loss: 0.362404, acc: 0.8432

Epoch: [02/25]---src, loss: 0.330124, acc: 0.9036
Epoch: [02/25]---val, loss: 0.333967, acc: 0.9036
Epoch: [02/25]---tar, loss: 0.365839, acc: 0.8653

Epoch: [03/25]---src, loss: 0.330052, acc: 0.9015
Epoch: [03/25]---val, loss: 0.333897, acc: 0.8848
Epoch: [03/25]---tar, loss: 0.364945, acc: 0.8436

Epoch: [04/25]---src, loss: 0.329205, acc: 0.9012
Epoch: [04/25]---val, loss: 0.339296, acc: 0.9222
Epoch: [04/25]---tar, loss: 0.376061, acc: 0.9002

Epoch: [05/25]---src, loss: 0.329224, acc: 0.8982
Epoch: [05/25]---val, loss: 0.333891, acc: 0.8990
Epoch: [05/25]---tar, loss: 0.359992, acc: 0.9098

Epoch: [06/25]---src, loss: 0.328851, acc: 0.8983
Epoch: [06/25]---val, loss: 0.334494, acc: 0.9108
Epoch: [06/25]---tar, loss: 0.357882, acc: 0.9240

Epoch: [07/25]---src, loss: 0.330282, acc: 0.8969
Epoch: [07/25]---val, loss: 0.334287, acc: 0.9284
Epoch: [07/25]---tar, loss: 0.362293, acc: 0.9247

Epoch: [08/25]---src, loss: 0.329142, acc: 0.8970
Epoch: [08/25]---val, loss: 0.345426, acc: 0.9111
Epoch: [08/25]---tar, loss: 0.378538, acc: 0.9237

Epoch: [09/25]---src, loss: 0.328856, acc: 0.9002
Epoch: [09/25]---val, loss: 0.341286, acc: 0.8779
Epoch: [09/25]---tar, loss: 0.362307, acc: 0.7775

Epoch: [10/25]---src, loss: 0.328975, acc: 0.8982
Epoch: [10/25]---val, loss: 0.337882, acc: 0.8985
Epoch: [10/25]---tar, loss: 0.362407, acc: 0.8837

Epoch: [11/25]---src, loss: 0.329133, acc: 0.8991
```

Relatively speaking, the rate is very fast at first, but after many iterations, the accuracy is little lower(May be the dataset is too small).

After reading Appendix D of "Self-Ensembling for Visual Domain Adaptation". I began to reproduce his digital migration network. I chose the simplest one. That is, the transfer learning from MNIST to USPS is realized. Its network structure is set as follows:

Description	Shape
28 × 28 Mono image	28 × 28 × 1
Conv 5 × 5 × 32, batch norm	24 × 24 × 32
Max-pool, 2x2	12 × 12 × 32
Conv 3 × 3 × 64, batch norm	10 × 10 × 64
Conv 3 × 3 × 64, batch norm	8 × 8 × 64
Max-pool, 2x2	4 × 4 × 64
Dropout, 50%	4 × 4 × 64
Fully connected, 256 units	256
Fully connected, 10 units, softmax	10

Table 6: MNIST ↔ USPS architecture

The code of network are as follows:

```
def __init__(self, n_classes):
        self.conv1_1 = nn.Conv2d(1, 32, (5, 5))
        self.conv1_1_bn = nn.BatchNorm2d(32)

        self.pool1 = nn.MaxPool2d((2, 2))

        self.conv2_1 = nn.Conv2d(32, 64, (3, 3))
        self.conv2_1_bn = nn.BatchNorm2d(64)

        self.conv2_2 = nn.Conv2d(64, 64, (3, 3))
        self.conv2_2_bn = nn.BatchNorm2d(64)

        self.pool2 = nn.MaxPool2d((2, 2))

        self.drop1 = nn.Dropout()

        self.fc3 = nn.Linear(1024, 256)

        self.fc4 = nn.Linear(256, n_classes)
```

And the result is:

```
batch size: 256 number workers: 4 learning rate=0.001 epochs: 25
```

```
root@a28dd28ff56a:/home/huangchenxi/transfer-learning/Mnist2Usps# python Baseline.
Epoch: [01/25]---src, loss: 0.192084, acc: 0.9433
Epoch: [01/25]---val, loss: 0.051905, acc: 0.9825
Epoch: [01/25]---tar, loss: 1.194448, acc: 0.7043

Epoch: [02/25]---src, loss: 0.060604, acc: 0.9806
Epoch: [02/25]---val, loss: 0.046370, acc: 0.9858
Epoch: [02/25]---tar, loss: 1.519606, acc: 0.6295

Epoch: [03/25]---src, loss: 0.047869, acc: 0.9850
Epoch: [03/25]---val, loss: 0.028589, acc: 0.9905
Epoch: [03/25]---tar, loss: 1.143164, acc: 0.7022

Epoch: [04/25]---src, loss: 0.038444, acc: 0.9876
Epoch: [04/25]---val, loss: 0.021601, acc: 0.9928
Epoch: [04/25]---tar, loss: 0.903657, acc: 0.7644

Epoch: [05/25]---src, loss: 0.032450, acc: 0.9901
Epoch: [05/25]---val, loss: 0.039623, acc: 0.9874
Epoch: [05/25]---tar, loss: 1.234952, acc: 0.7335

Epoch: [06/25]---src, loss: 0.028346, acc: 0.9912
Epoch: [06/25]---val, loss: 0.018120, acc: 0.9939
Epoch: [06/25]---tar, loss: 0.971560, acc: 0.7553

Epoch: [07/25]---src, loss: 0.025108, acc: 0.9915
Epoch: [07/25]---val, loss: 0.021287, acc: 0.9933
Epoch: [07/25]---tar, loss: 0.861253, acc: 0.7670

Epoch: [08/25]---src, loss: 0.024066, acc: 0.9922
Epoch: [08/25]---val, loss: 0.021791, acc: 0.9929
Epoch: [08/25]---tar, loss: 1.067999, acc: 0.7469

Epoch: [09/25]---src, loss: 0.020962, acc: 0.9934
Epoch: [09/25]---val, loss: 0.017854, acc: 0.9944
Epoch: [09/25]---tar, loss: 0.802467, acc: 0.7923

Epoch: [10/25]---src, loss: 0.017997, acc: 0.9940
Epoch: [10/25]---val, loss: 0.018616, acc: 0.9936
Epoch: [10/25]---tar, loss: 0.971246, acc: 0.7475

Epoch: [11/25]---src, loss: 0.018466, acc: 0.9939
Epoch: [11/25]---val, loss: 0.018829, acc: 0.9948
Epoch: [11/25]---tar, loss: 0.823219, acc: 0.7963

Epoch: [12/25]---src, loss: 0.016613, acc: 0.9945
Epoch: [12/25]---val, loss: 0.017613, acc: 0.9950
Epoch: [12/25]---tar, loss: 0.801601, acc: 0.8118
```

```
Epoch: [13/25]---src, loss: 0.015181, acc: 0.9949
Epoch: [13/25]---val, loss: 0.018595, acc: 0.9939
Epoch: [13/25]---tar, loss: 0.847482, acc: 0.7943

Epoch: [14/25]---src, loss: 0.014008, acc: 0.9954
Epoch: [14/25]---val, loss: 0.017293, acc: 0.9948
Epoch: [14/25]---tar, loss: 0.938279, acc: 0.7726

Epoch: [15/25]---src, loss: 0.012149, acc: 0.9960
Epoch: [15/25]---val, loss: 0.019157, acc: 0.9940
Epoch: [15/25]---tar, loss: 0.856247, acc: 0.7869

Epoch: [16/25]---src, loss: 0.010967, acc: 0.9964
Epoch: [16/25]---val, loss: 0.020897, acc: 0.9937
Epoch: [16/25]---tar, loss: 1.030946, acc: 0.7727

Epoch: [17/25]---src, loss: 0.011991, acc: 0.9960
Epoch: [17/25]---val, loss: 0.018124, acc: 0.9945
Epoch: [17/25]---tar, loss: 0.753240, acc: 0.8066

Epoch: [18/25]---src, loss: 0.012081, acc: 0.9959
Epoch: [18/25]---val, loss: 0.019345, acc: 0.9941
Epoch: [18/25]---tar, loss: 1.005149, acc: 0.7689

Epoch: [19/25]---src, loss: 0.010824, acc: 0.9962
Epoch: [19/25]---val, loss: 0.016516, acc: 0.9945
Epoch: [19/25]---tar, loss: 0.724475, acc: 0.8185

Epoch: [20/25]---src, loss: 0.009383, acc: 0.9966
Epoch: [20/25]---val, loss: 0.024679, acc: 0.9937
Epoch: [20/25]---tar, loss: 0.885000, acc: 0.7923

Epoch: [21/25]---src, loss: 0.009041, acc: 0.9970
Epoch: [21/25]---val, loss: 0.018507, acc: 0.9948
Epoch: [21/25]---tar, loss: 0.968110, acc: 0.7711

Epoch: [22/25]---src, loss: 0.008792, acc: 0.9969
Epoch: [22/25]---val, loss: 0.020892, acc: 0.9944
Epoch: [22/25]---tar, loss: 0.817327, acc: 0.7844

Epoch: [23/25]---src, loss: 0.009098, acc: 0.9969
Epoch: [23/25]---val, loss: 0.021970, acc: 0.9936
Epoch: [23/25]---tar, loss: 1.027080, acc: 0.7816

Epoch: [24/25]---src, loss: 0.008881, acc: 0.9968
Epoch: [24/25]---val, loss: 0.018496, acc: 0.9945
Epoch: [24/25]---tar, loss: 0.681893, acc: 0.8276

Epoch: [25/25]---src, loss: 0.007519, acc: 0.9976
Epoch: [25/25]---val, loss: 0.019360, acc: 0.9945
Epoch: [25/25]---tar, loss: 0.962274, acc: 0.7816
```

In experience, lowering some learning rates, increasing the number of iterations, improving the size of the batch, etc., can improve some accuracy. The experiment also proved.

`batch size: 36 number workers: 2 learning rate=3e-4 epochs: 256`

```
$ python main.py --model='Baseline_Digit' --source='MNIST' --target='USPS' --n_classes=10
Baseline_Digit
Epoch: [01/256]---src, loss: 0.173502, acc: 0.9516
Epoch: [01/256]---val, loss: 0.043602, acc: 0.9847
Epoch: [01/256]---tar, loss: 1.286771, acc: 0.6582

Epoch: [02/256]---src, loss: 0.061344, acc: 0.9817
Epoch: [02/256]---val, loss: 0.030976, acc: 0.9902
Epoch: [02/256]---tar, loss: 1.113913, acc: 0.7166

Epoch: [03/256]---src, loss: 0.046306, acc: 0.9853
Epoch: [03/256]---val, loss: 0.022454, acc: 0.9926
Epoch: [03/256]---tar, loss: 0.853039, acc: 0.7671

Epoch: [04/256]---src, loss: 0.037732, acc: 0.9879
Epoch: [04/256]---val, loss: 0.024954, acc: 0.9917
Epoch: [04/256]---tar, loss: 0.903350, acc: 0.7910

Epoch: [05/256]---src, loss: 0.035376, acc: 0.9887
Epoch: [05/256]---val, loss: 0.021251, acc: 0.9928
Epoch: [05/256]---tar, loss: 1.204596, acc: 0.6985

Epoch: [06/256]---src, loss: 0.028423, acc: 0.9908
Epoch: [06/256]---val, loss: 0.031079, acc: 0.9900
Epoch: [06/256]---tar, loss: 1.043929, acc: 0.7461

Epoch: [07/256]---src, loss: 0.026714, acc: 0.9914
Epoch: [07/256]---val, loss: 0.027179, acc: 0.9911
Epoch: [07/256]---tar, loss: 0.828751, acc: 0.7917

Epoch: [08/256]---src, loss: 0.023795, acc: 0.9925
Epoch: [08/256]---val, loss: 0.018674, acc: 0.9935
Epoch: [08/256]---tar, loss: 0.924846, acc: 0.7823

Epoch: [09/256]---src, loss: 0.020756, acc: 0.9935
Epoch: [09/256]---val, loss: 0.018090, acc: 0.9935
Epoch: [09/256]---tar, loss: 0.860521, acc: 0.7851

Epoch: [10/256]---src, loss: 0.020786, acc: 0.9933
Epoch: [10/256]---val, loss: 0.017851, acc: 0.9944
Epoch: [10/256]---tar, loss: 0.744768, acc: 0.8303

Epoch: [11/256]---src, loss: 0.017645, acc: 0.9943
Epoch: [11/256]---val, loss: 0.017168, acc: 0.9947
```

```
Epoch: [11/256]---src, loss: 0.017645, acc: 0.9943
Epoch: [11/256]---val, loss: 0.017168, acc: 0.9947
Epoch: [11/256]---tar, loss: 0.693371, acc: 0.8479

Epoch: [12/256]---src, loss: 0.016330, acc: 0.9947
Epoch: [12/256]---val, loss: 0.018545, acc: 0.9943
Epoch: [12/256]---tar, loss: 0.561949, acc: 0.8546

Epoch: [13/256]---src, loss: 0.015491, acc: 0.9950
Epoch: [13/256]---val, loss: 0.017385, acc: 0.9951
Epoch: [13/256]---tar, loss: 0.701490, acc: 0.8351

Epoch: [14/256]---src, loss: 0.015387, acc: 0.9948
Epoch: [14/256]---val, loss: 0.018047, acc: 0.9940
Epoch: [14/256]---tar, loss: 0.765911, acc: 0.8305

Epoch: [15/256]---src, loss: 0.013603, acc: 0.9954
Epoch: [15/256]---val, loss: 0.016428, acc: 0.9947
Epoch: [15/256]---tar, loss: 0.556106, acc: 0.8631

Epoch: [16/256]---src, loss: 0.011403, acc: 0.9961
Epoch: [16/256]---val, loss: 0.020806, acc: 0.9941
Epoch: [16/256]---tar, loss: 0.918480, acc: 0.8198

Epoch: [17/256]---src, loss: 0.011906, acc: 0.9959
Epoch: [17/256]---val, loss: 0.015828, acc: 0.9956
Epoch: [17/256]---tar, loss: 0.726540, acc: 0.8553

Epoch: [18/256]---src, loss: 0.011253, acc: 0.9963
Epoch: [18/256]---val, loss: 0.017504, acc: 0.9947
Epoch: [18/256]---tar, loss: 0.460645, acc: 0.8756

Epoch: [19/256]---src, loss: 0.008843, acc: 0.9970
Epoch: [19/256]---val, loss: 0.022262, acc: 0.9926
Epoch: [19/256]---tar, loss: 0.633865, acc: 0.8532

Epoch: [20/256]---src, loss: 0.009553, acc: 0.9966
Epoch: [20/256]---val, loss: 0.020464, acc: 0.9942
Epoch: [20/256]---tar, loss: 0.644801, acc: 0.8441

Epoch: [21/256]---src, loss: 0.009262, acc: 0.9967
Epoch: [21/256]---val, loss: 0.018933, acc: 0.9944
Epoch: [21/256]---tar, loss: 0.674756, acc: 0.8512

Epoch: [22/256]---src, loss: 0.008381, acc: 0.9972
Epoch: [22/256]---val, loss: 0.020346, acc: 0.9939
Epoch: [22/256]---tar, loss: 0.785033, acc: 0.8259

Epoch: [23/256]---src, loss: 0.008080, acc: 0.9972
Epoch: [23/256]---val, loss: 0.017641, acc: 0.9951
Epoch: [23/256]---tar, loss: 0.786440, acc: 0.8244

Epoch: [24/256]---src, loss: 0.008657, acc: 0.9972
Epoch: [24/256]---val, loss: 0.017334, acc: 0.9950
Epoch: [24/256]---tar, loss: 0.732182, acc: 0.8461

Epoch: [25/256]---src, loss: 0.006765, acc: 0.9977
Epoch: [25/256]---val, loss: 0.022630, acc: 0.9949
Epoch: [25/256]---tar, loss: 0.683339, acc: 0.8404

Epoch: [26/256]---src, loss: 0.008382, acc: 0.9973
```

```
Epoch: [26/256]---val, loss: 0.019544, acc: 0.9942
Epoch: [26/256]---tar, loss: 0.679418, acc: 0.8269
```

Compare

In my first 25 iterations, the best is **86.31**%. Since it does not have a separate MT, **MT+CT** is in the range of 88.14 ± -0.34, then we can imagine that only the result of MT must be correct. Experimental results of other data sets are compared in the next heading.

	USPS – MNIST	MNIST – USPS	SVHN – MNIST	MNIST – SVHN	CIFAR – STL	STL – CIFAR	Syn Digits – SVHN	Syn Signs – GTSRB
TRAIN ON SOURCE								
SupSrc*	77.55 ±0.8	82.03 ±1.16	66.5 ±1.93	25.44 ±2.8	72.84 ±0.61	51.88 ±1.44	86.86 ±0.86	96.95 ±0.36
SupSrc+TF	77.53 ±4.63	95.39 ±0.93	68.65 ±1.5	24.86 ±3.29	75.2 ±0.28	59.06 ±1.02	87.45 ±0.65	97.3 ±0.16
SupSrc+TFA	91.97 ±2.15	96.25 ±0.54	71.73 ±5.73	28.69 ±1.59	75.18 ±0.76	59.38 ±0.58	87.16 ±0.85	98.02 ±0.20
Specific aug.[b]	–	–	–	61.99 ±3.9	–	–	–	–
RevGrad[a] [1]	74.01	91.11	73.91	35.67	66.12	56.91	91.09	88.65
DCRN [2]	73.67	91.8	81.97	40.05	66.37	58.65	–	–
G2A [3]	90.8	92.5	84.70	36.4	–	–	–	–
ADDA [4]	90.1	89.4	76.00	–	–	–	–	–
ATT [5]	–	–	86.20	52.8	–	–	93.1	96.2
SBADA-GAN [6]	97.60	95.04	76.14	61.08	–	–	–	–
ADA [7]	–	–	97.6	–	–	–	91.86	97.66
OUR RESULTS								
MT+TF	98.07 ±2.82	**98.26** ±0.11	99.18 ±0.12	13.96[c] ±4.41	80.08 ±0.25	18.3 ±9.03	15.94 ±0.0	98.63 ±0.09
MT+CT*	92.35 ±8.61	88.14 ±0.34	93.33 ±5.88	33.87[c] ±4.02	77.53 ±0.11	71.65 ±0.67	96.01 ±0.08	98.53 ±0.15
MT+CT+TF	97.28 ±2.74	98.13 ±0.17	98.64 ±0.42	34.15[c] ±3.56	79.73 ±0.45	**74.24** ±0.46	96.51 ±0.08	98.66 ±0.12
MT+CT+TFA	**99.54** ±0.04	98.23 ±0.13	**99.26** ±0.05	37.49[c] ±2.44	**80.09** ±0.31	69.86 ±1.97	**97.11** ±0.04	**99.37** ±0.09
Specific aug.[b]	–	–	–	**97.0**[c] ±0.06	–	–	–	–
TRAIN ON TARGET								
SupTgt*	99.53 ±0.02	97.29 ±0.2	99.59 ±0.08	95.7 ±0.13	67.75 ±2.23	88.86 ±0.38	95.62 ±0.2	98.49 ±0.32
SupTgt+TF	99.62 ±0.04	97.65 ±0.17	99.61 ±0.04	96.19 ±0.1	70.98 ±0.79	89.83 ±0.39	96.18 ±0.09	98.64 ±0.09
SupTgt+TFA	99.62 ±0.03	97.83 ±0.17	99.59 ±0.06	96.65 ±0.11	70.03 ±1.13	90.44 ±0.38	96.59 ±0.09	99.22 ±0.22
Specific aug.[b]	–	–	–	97.16 ±0.05	–	–	–	–

[1] Ganin & Lempitsky (2015), [2] Ghifary et al. (2016), [3] Sankaranarayanan et al. (2017), [4] Tzeng et al. (2017), [5] Saito et al. (2017a), [6] Russo et al. (2017), [7] Haeusser et al. (2017)

[a] RevGrad results were available in both Ganin & Lempitsky (2015) and Ghifary et al. (2016); we drew results from both papers to obtain results for all of the experiments shown.

[b] MNIST → SVHN specific intensity augmentation as described in Section 4.1.

[c] MNIST → SVHN experiments used class balance loss.

Table 1: Small image benchmark classification accuracy; each result is presented as *mean* ± *standard deviation*, computed from 5 independent runs. The abbreviations for components of our models are as follows: MT = mean teacher, CT = confidence thresholding, TF = translation and horizontal flip augmentation, TFA = translation, horizontal flip and affine augmentation, * indicates minimal augmentation.

Self-ensemble Visual Domain Adapt Master

First you need to install some packages, such as batchup, skimage.

- When installing batchup, if you use `pip install batchup`. if the version is too high to prevent installation, you can use`pip install batchup==0.1.0.`
- When installing skimage, the command is `pip install scikit-image`; If the problem of timeout still occurs, modify the image source.

The main highlight of the code is the addition of **Confidence threshodling** and **class balance loss**, and the addition of **data enhancements** based on Mean Teacher and Temporal Ensembling. Where `TRAIN clf loss`is the loss of training classification, and `unsup(tgt)` is the loss of the target under unsupervised conditions. You can see that the test results **are close to the supervised** way.

The input information as follows:

```
root at 641aa73707ea in /home/huangchenxi/transfer-learning/Self_ensemble
$ python self_ensemble.py
Loading MNIST...
MNIST:
train: X.shape=(60000, 1, 28, 28), y.shape=(60000,),
val: X.shape=(0, 1, 28, 28), y.shape=(0,),
test: X.shape=(10000, 1, 28, 28), y.shape=(10000,)
MNIST: train: X.min=0.0, X.max=1.0
Loading USPS...
USPS: train:
X.shape=(7291, 1, 28, 28), y.shape=(7291,),
val: X.shape=(0, 1, 28, 28), y.shape=(0,),
test: X.shape=(2007, 1, 28, 28), y.shape=(2007,)
USPS: train: X.min=0.0, X.max=1.0
===========Loaded data==========
Dataset:
SOURCE Train: X.shape=(60000, 1, 28, 28), y.shape=(60000,)
SOURCE Test: X.shape=(10000, 1, 28, 28), y.shape=(10000,)
TARGET Train: X.shape=(7291, 1, 28, 28)
TARGET Test: X.shape=(2007, 1, 28, 28), y.shape=(2007,)
===========Built network=========
```

And the training result as follows:

```
==========Training==========
*** Epoch 0 took 13.36s:
TRAIN clf loss=0.678767, unsup (tgt) loss=0.000006, conf mask=0.028%, unsup mask=0.028%; SRC TEST ERR=2.910%,
TGT TEST student err=17.838%, TGT TEST teacher err=21.574%
*** Epoch 1 took 11.91s:
TRAIN clf loss=0.213565, unsup (tgt) loss=0.003726, conf mask=16.330%, unsup mask=16.330%; SRC TEST ERR=2.480%,
TGT TEST student err=16.791%, TGT TEST teacher err=15.247%
*** Epoch 2 took 11.85s:
TRAIN clf loss=0.198579, unsup (tgt) loss=0.010548, conf mask=44.331%, unsup mask=44.331%; SRC TEST ERR=3.450%,
TGT TEST student err=10.065%, TGT TEST teacher err=10.912%
*** Epoch 3 took 11.91s:
TRAIN clf loss=0.165477, unsup (tgt) loss=0.014070, conf mask=56.748%, unsup mask=56.748%; SRC TEST ERR=2.000%,
TGT TEST student err=8.271%, TGT TEST teacher err=7.773%
*** Epoch 4 took 11.73s:
TRAIN clf loss=0.141033, unsup (tgt) loss=0.014656, conf mask=65.030%, unsup mask=65.030%; SRC TEST ERR=1.810%,
TGT TEST student err=6.527%, TGT TEST teacher err=5.580%
*** Epoch 5 took 12.22s:
TRAIN clf loss=0.114963, unsup (tgt) loss=0.015976, conf mask=70.810%, unsup mask=70.810%; SRC TEST ERR=1.350%,
TGT TEST student err=6.677%, TGT TEST teacher err=4.833%
*** Epoch 6 took 12.36s:
TRAIN clf loss=0.112600, unsup (tgt) loss=0.016339, conf mask=75.512%, unsup mask=75.512%; SRC TEST ERR=2.650%,
TGT TEST student err=5.082%, TGT TEST teacher err=4.584%
*** Epoch 7 took 11.68s:
TRAIN clf loss=0.103892, unsup (tgt) loss=0.017079, conf mask=77.959%, unsup mask=77.959%; SRC TEST ERR=0.930%,
TGT TEST student err=5.381%, TGT TEST teacher err=4.385%
*** Epoch 8 took 12.13s:
TRAIN clf loss=0.087609, unsup (tgt) loss=0.016692, conf mask=79.148%, unsup mask=79.148%; SRC TEST ERR=1.260%,
TGT TEST student err=4.385%, TGT TEST teacher err=4.235%
*** Epoch 9 took 12.24s:
TRAIN clf loss=0.096784, unsup (tgt) loss=0.016795, conf mask=81.139%, unsup mask=81.139%; SRC TEST ERR=1.070%,
TGT TEST student err=5.531%, TGT TEST teacher err=3.837%
*** Epoch 10 took 12.13s:
TRAIN clf loss=0.103647, unsup (tgt) loss=0.017304, conf mask=82.826%, unsup mask=82.826%; SRC TEST ERR=1.280%,
TGT TEST student err=4.634%, TGT TEST teacher err=3.538%
Epoch 11 took 12.16s:
TRAIN clf loss=0.080447, unsup (tgt) loss=0.016555, conf mask=81.817%, unsup mask=81.817%; SRC TEST ERR=1.210%,
TGT TEST student err=4.634%, TGT TEST teacher err=3.538%
Epoch 12 took 12.07s:
TRAIN clf loss=0.078208, unsup (tgt) loss=0.016775, conf mask=82.647%, unsup mask=82.647%; SRC TEST ERR=2.050%,
TGT TEST student err=4.136%, TGT TEST teacher err=3.438%
*** Epoch 13 took 12.00s:
TRAIN clf loss=0.079053, unsup (tgt) loss=0.017073, conf mask=84.403%, unsup mask=84.403%; SRC TEST ERR=0.870%,
TGT TEST student err=3.737%, TGT TEST teacher err=3.438%
*** Epoch 14 took 11.93s:
TRAIN clf loss=0.071268, unsup (tgt) loss=0.017110, conf mask=85.246%, unsup mask=85.246%; SRC TEST ERR=1.150%,
TGT TEST student err=4.086%, TGT TEST teacher err=2.840%
```

And so on. The numbers are fluctuating, but the overall results are good

```
TRAIN clf loss=0.024455, unsup (tgt) loss=0.017092, conf mask=96.184%, unsup mask=96.184%; SRC TEST ERR=0.640%,
TGT TEST student err=2.242%, TGT TEST teacher err=1.794%
Epoch 185 took 12.51s:
TRAIN clf loss=0.023293, unsup (tgt) loss=0.017368, conf mask=96.114%, unsup mask=96.114%; SRC TEST ERR=0.540%,
TGT TEST student err=1.844%, TGT TEST teacher err=1.794%
*** Epoch 186 took 12.30s:
TRAIN clf loss=0.024465, unsup (tgt) loss=0.017586, conf mask=96.695%, unsup mask=96.695%; SRC TEST ERR=0.600%,
TGT TEST student err=1.943%, TGT TEST teacher err=1.744%
Epoch 187 took 12.53s:
TRAIN clf loss=0.026973, unsup (tgt) loss=0.017184, conf mask=95.534%, unsup mask=95.534%; SRC TEST ERR=0.660%,
TGT TEST student err=1.993%, TGT TEST teacher err=1.744%
Epoch 188 took 12.43s:
TRAIN clf loss=0.018069, unsup (tgt) loss=0.017135, conf mask=95.700%, unsup mask=95.700%; SRC TEST ERR=0.630%,
TGT TEST student err=1.993%, TGT TEST teacher err=1.644%
Epoch 189 took 12.51s:
TRAIN clf loss=0.018568, unsup (tgt) loss=0.017056, conf mask=95.755%, unsup mask=95.755%; SRC TEST ERR=0.490%,
TGT TEST student err=2.093%, TGT TEST teacher err=1.794%
Epoch 190 took 12.44s:
TRAIN clf loss=0.030494, unsup (tgt) loss=0.017192, conf mask=96.128%, unsup mask=96.128%; SRC TEST ERR=0.560%,
TGT TEST student err=1.993%, TGT TEST teacher err=1.694%
Epoch 191 took 12.43s:
TRAIN clf loss=0.015902, unsup (tgt) loss=0.017317, conf mask=96.142%, unsup mask=96.142%; SRC TEST ERR=0.460%,
TGT TEST student err=1.943%, TGT TEST teacher err=1.545%
Epoch 192 took 12.61s:
TRAIN clf loss=0.024231, unsup (tgt) loss=0.017064, conf mask=96.253%, unsup mask=96.253%; SRC TEST ERR=0.520%,
TGT TEST student err=1.993%, TGT TEST teacher err=1.594%
Epoch 193 took 12.49s:
TRAIN clf loss=0.030299, unsup (tgt) loss=0.017117, conf mask=95.783%, unsup mask=95.783%; SRC TEST ERR=0.520%,
TGT TEST student err=1.744%, TGT TEST teacher err=1.545%
Epoch 194 took 12.63s:
TRAIN clf loss=0.017378, unsup (tgt) loss=0.017203, conf mask=95.437%, unsup mask=95.437%; SRC TEST ERR=0.440%,
TGT TEST student err=1.893%, TGT TEST teacher err=1.594%
Epoch 195 took 12.44s:
TRAIN clf loss=0.018870, unsup (tgt) loss=0.017045, conf mask=96.142%, unsup mask=96.142%; SRC TEST ERR=0.680%,
TGT TEST student err=1.943%, TGT TEST teacher err=1.545%
Epoch 196 took 12.35s:
TRAIN clf loss=0.022651, unsup (tgt) loss=0.017567, conf mask=96.225%, unsup mask=96.225%; SRC TEST ERR=0.530%,
TGT TEST student err=1.844%, TGT TEST teacher err=1.744%
Epoch 197 took 12.72s:
TRAIN clf loss=0.023108, unsup (tgt) loss=0.017201, conf mask=96.336%, unsup mask=96.336%; SRC TEST ERR=0.590%,
TGT TEST student err=2.043%, TGT TEST teacher err=1.844%
Epoch 198 took 12.31s:
TRAIN clf loss=0.022707, unsup (tgt) loss=0.017238, conf mask=95.893%, unsup mask=95.893%; SRC TEST ERR=0.530%,
TGT TEST student err=1.893%, TGT TEST teacher err=1.594%
Epoch 199 took 12.51s:
TRAIN clf loss=0.016236, unsup (tgt) loss=0.017005, conf mask=95.796%, unsup mask=95.796%; SRC TEST ERR=0.680%,
TGT TEST student err=1.993%, TGT TEST teacher err=1.744%
Closing remaining open files:/root/.batchup/data/mnist.h5...done/root/.batchup/data/usps.h5...done
```

After 200 epochs, the result is respectively perfect.

Obviously, the loss is smaller, and you can see the target loss in unsupervised is so small that can even **be same with supervised.**

Compare

The results of the paper is:

	USPS – MNIST	MNIST – USPS	SVHN – MNIST	MNIST – SVHN	CIFAR – STL	STL – CIFAR	Syn Digits – SVHN	Syn Signs – GTSRB
TRAIN ON SOURCE								
SupSrc*	77.55 ±0.8	82.03 ±1.16	66.5 ±1.93	25.44 ±2.8	72.84 ±0.61	51.88 ±1.44	86.86 ±0.86	96.95 ±0.36
SupSrc+TF	77.53 ±4.63	95.39 ±0.93	68.65 ±1.5	24.86 ±3.29	75.2 ±0.28	59.06 ±1.02	87.45 ±0.65	97.3 ±0.16
SupSrc+TFA	91.97 ±2.15	96.25 ±0.54	71.73 ±5.73	28.69 ±1.59	75.18 ±0.76	59.38 ±0.58	87.16 ±0.85	98.02 ±0.20
Specific aug.[b]	–	–	–	61.99 ±3.9	–	–	–	–
RevGrad[a] [1]	74.01	91.11	73.91	35.67	66.12	56.91	91.09	88.65
DCRN [2]	73.67	91.8	81.97	40.05	66.37	58.65	–	–
G2A [3]	90.8	92.5	84.70	36.4	–	–	–	–
ADDA [4]	90.1	89.4	76.00	–	–	–	–	–
ATT [5]	–	–	86.20	52.8	–	–	93.1	96.2
SBADA-GAN [6]	97.60	95.04	76.14	61.08	–	–	–	–
ADA [7]	–	–	97.6	–	–	–	91.86	97.66
OUR RESULTS								
MT+TF	98.07 ±2.82	**98.26** ±0.11	99.18 ±0.12	13.96[c] ±4.41	80.08 ±0.25	18.3 ±9.03	15.94 ±0.0	98.63 ±0.09
MT+CT*	92.35 ±8.61	88.14 ±0.34	93.33 ±5.88	33.87[c] ±4.02	77.53 ±0.11	71.65 ±0.67	96.01 ±0.08	98.53 ±0.15
MT+CT+TF	97.28 ±2.74	98.13 ±0.17	98.64 ±0.42	34.15[c] ±3.56	79.73 ±0.45	**74.24** ±0.46	96.51 ±0.08	98.66 ±0.12
MT+CT+TFA	**99.54** ±0.04	98.23 ±0.13	**99.26** ±0.05	37.49[c] ±2.44	**80.09** ±0.31	69.86 ±1.97	**97.11** ±0.04	**99.37** ±0.09
Specific aug.[b]	–	–	–	**97.0**[c] ±0.06	–	–	–	–
TRAIN ON TARGET								
SupTgt*	99.53 ±0.02	97.29 ±0.2	99.59 ±0.08	95.7 ±0.13	67.75 ±2.23	88.86 ±0.38	95.62 ±0.2	98.49 ±0.32
SupTgt+TF	99.62 ±0.04	97.65 ±0.17	99.61 ±0.04	96.19 ±0.1	70.98 ±0.79	89.83 ±0.39	96.18 ±0.09	98.64 ±0.09
SupTgt+TFA	99.62 ±0.03	97.83 ±0.17	99.59 ±0.06	96.65 ±0.11	70.03 ±1.13	90.44 ±0.38	96.59 ±0.09	99.22 ±0.22
Specific aug.[b]	–	–	–	97.16 ±0.05	–	–	–	–

[1] Ganin & Lempitsky (2015), [2] Ghifary et al. (2016), [3] Sankaranarayanan et al. (2017), [4] Tzeng et al. (2017), [5] Saito et al. (2017a), [6] Russo et al. (2017), [7] Haeusser et al. (2017)

[a] RevGrad results were available in both Ganin & Lempitsky (2015) and Ghifary et al. (2016); we drew results from both papers to obtain results for all of the experiments shown.

[b] MNIST → SVHN specific intensity augmentation as described in Section 4.1.

[c] MNIST → SVHN experiments used class balance loss.

Table 1: Small image benchmark classification accuracy; each result is presented as *mean* ± *standard deviation*, computed from 5 independent runs. The abbreviations for components of our models are as follows: MT = mean teacher, CT = confidence thresholding, TF = translation and horizontal flip augmentation, TFA = translation, horizontal flip and affine augmentation, * indicates minimal augmentation.

I can see that **my error rate is about 1.9%**, that is, **the accuracy rate is about 98%**, which is the same as in the paper. (But my runtime is so long. Maybe something bad I ignore.)

	In paper	In my test
U->M	92.35 ± 8.61	89.75
M->U	88.14 ± 0.34	95.79
S->M	95.33 ± 5.88	91.97
M->S	33.87 ± 4.02	63.23

Other dataset combination tests is in `results_image/Self-ensemble-X-Y.png`. The kinds are follows:

```
if exp == 'svhn_mnist':
    d_source = Data_transform.load_svhn(zero_centre=False, greyscale=True)
    d_target = Data_transform.load_mnist(invert=False, zero_centre=False, pad32=True, val=False)
elif exp == 'mnist_svhn':
    d_source = Data_transform.load_mnist(invert=False, zero_centre=False, pad32=True)
    d_target = Data_transform.load_svhn(zero_centre=False, greyscale=True, val=False)
elif exp == 'svhn_mnist_rgb':
    d_source = Data_transform.load_svhn(zero_centre=False, greyscale=False)
    d_target = Data_transform.load_mnist(invert=False, zero_centre=False, pad32=True, val=False, rgb=True)
elif exp == 'mnist_svhn_rgb':
    d_source = Data_transform.load_mnist(invert=False, zero_centre=False, pad32=True, rgb=True)
    d_target = Data_transform.load_svhn(zero_centre=False, greyscale=False, val=False)
elif exp == 'cifar_stl':
    d_source = Data_transform.load_cifar10(range_01=False)
    d_target = Data_transform.load_stl(zero_centre=False, val=False)
elif exp == 'stl_cifar':
    d_source = Data_transform.load_stl(zero_centre=False)
    d_target = Data_transform.load_cifar10(range_01=False, val=False)
elif exp == 'mnist_usps':
    d_source = Data_transform.load_mnist(zero_centre=False)
    d_target = Data_transform.load_usps(zero_centre=False, scale28=True, val=False)
elif exp == 'usps_mnist':
    d_source = Data_transform.load_usps(zero_centre=False, scale28=True)
    d_target = Data_transform.load_mnist(zero_centre=False, val=False)
elif exp == 'syndigits_svhn':
    d_source = Data_transform.load_syn_digits(zero_centre=False)
    d_target = Data_transform.load_svhn(zero_centre=False, val=False)
elif exp == 'synsigns_gtsrb':
    d_source = Data_transform.load_syn_signs(zero_centre=False)
    d_target = Data_transform.load_gtsrb(zero_centre=False, val=False)
```

Domain Adversarial Training of Neural Networks

The data set should first convert and preprocessed by the SVMlight toolkit . According to the algorithm of the shallow NN, I did `5.1.5 Proxy Distance-Figure 3`, tested the four combinations:

- whether there is with adversarial
- whether there is with mSDA

The Shallow NN is:

> Algorithm: Shallow DANN – Stochastic training update

Input：

1. Samples

$S = (x_i, y_i)_{i=1}^{n}, T = (x_i)_{i=1}^{n'}$

2. hidden layer soze$D$
3. input layer parameterλ
4. learning rateμ

Input： CNN$\{W, V, b, c\}$

$W, V \leftarrow \text{random\_init}(D)$

$b, c, u, d \leftarrow 0$

while stopping criterion is not met do

for i from 1 to n do

// forward propagation

$G_f(x_i) \leftarrow sigm(b + Wx_i)$

$G_y(G_f(x_i)) \leftarrow softmax(VG_f(x_i) + c)$

// backward propagation

$\Delta_c \leftarrow (e(y_i) - G_y(G_f(x_i)))$

$\Delta_V \leftarrow \Delta_c G_f(x_i)^T$

$\Delta_b \leftarrow (V^T \Delta_c) \odot G_f(x_i) \odot (1 - G_f(x_i))$

$\Delta W \leftarrow \Delta_b \cdot (x_i)^T$

//from now domain, regularization

$G_d(G_f(x_i)) \leftarrow sigm(d + u^T G_f(x_i))$

$\Delta_d \leftarrow \lambda(1 - G_d(G_f(x_i)))$

$\Delta_u \leftarrow \lambda(1 - G_d(G_f(x_i)))G_f(x_i)$

$\text{tmp} \leftarrow \lambda(1 - G_d(G_f(x_i))) \times u \odot G_f(x_i) \odot (1 - G_f(x_i))$

$\Delta_b \leftarrow \Delta_b + tmp$

$\Delta_W \leftarrow \Delta_W + tmp \cdot (x_i)^T$

//from other domain, regularization

$j \leftarrow \text{uniform\_integer}(1, \ldots, n')$

$G_f(x_j) \leftarrow sigm(b + Wx_j)$

$G_d(G_f(x_j)) \leftarrow sigm(d + u^T G_f(x_j))$

$\Delta_d \leftarrow \Delta_d - \lambda G_d(G_f(x_j))$

$\Delta_u \leftarrow \Delta_u - \lambda G_d(G_f(x_j))G_f(x_j)$

$\text{tmp} \leftarrow -\lambda G_d(G_f(x_j)) \times u \odot G_f(x_j) \odot (1 - G_f(x_j))$

$\Delta_b \leftarrow \Delta_b + tmp$

$\Delta_W \leftarrow \Delta_W + tmp \cdot (x_j)^T$

//update CNN inner parameters

$W \leftarrow W - \mu\Delta_W$

$V \leftarrow V - \mu\Delta_V$

$b \leftarrow b - \mu\Delta_b$

$c \leftarrow c - \mu\Delta_c$

//update domain classifier

$u \leftarrow u - \mu\Delta u$

$d \leftarrow d - \mu\Delta d$

end for

end while

And the code for the shallow NN is in `DANN.DANN.fit()`.

The data set is `Office-31-Amazon`, the source domain is `DVD`, the target The domain is `electronics`. At the same time, each group has a set of comparisons, whether the `PAD-agent distance method` is used. Three types of training loss, verification loss, and test loss were calculated.

The results are as follows:

- With mSDA
  - Without adversarial

```
Loading data...
source file: ../data/Office-31/Amazon/dvd_train.svmlight
target file: ../data/Office-31/Amazon/electronics_train.svmlight
test file:   ../data/Office-31/Amazon/electronics_test.svmlight
Computing mSDA representations...
Fit...
[DANN parameters] {'hidden_layer_size': 50, 'maxiter': 200, 'lambd
a_adapt': 0.0, 'epsilon_init': None, 'learning_rate': 0.0001, 'adv
ersarial_representation': False, 'seed': 12342, 'verbose': True}
[DANN best valid risk so far] 0.255000 (iter 0)
[DANN best valid risk so far] 0.240000 (iter 1)
[DANN best valid risk so far] 0.230000 (iter 2)
[DANN best valid risk so far] 0.215000 (iter 3)
[DANN best valid risk so far] 0.210000 (iter 4)
[DANN best valid risk so far] 0.210000 (iter 5)
[DANN best valid risk so far] 0.210000 (iter 6)
[DANN best valid risk so far] 0.210000 (iter 7)
[DANN best valid risk so far] 0.210000 (iter 8)
[DANN best valid risk so far] 0.210000 (iter 9)
[DANN best valid risk so far] 0.210000 (iter 10)
[DANN best valid risk so far] 0.210000 (iter 12)
[DANN best valid risk so far] 0.210000 (iter 13)
[DANN best valid risk so far] 0.210000 (iter 14)
[DANN best valid risk so far] 0.210000 (iter 15)
[DANN best valid risk so far] 0.210000 (iter 16)
[DANN best valid risk so far] 0.210000 (iter 19)
[DANN early stop] iter 31
Predict...
Training Risk   = 0.124444
Validation Risk = 0.210000
Test Risk       = 0.231121
================================================================================
```

```
================================================================
Computing PAD on DANN representation...
PAD on (1800, 2000) examples
[ PAD C = 0.000010 ] train risk: 0.473684  test risk: 0.473684
[ PAD C = 0.000100 ] train risk: 0.473684  test risk: 0.473684
[ PAD C = 0.001000 ] train risk: 0.473684  test risk: 0.473684
[ PAD C = 0.010000 ] train risk: 0.337368  test risk: 0.351053
[ PAD C = 0.100000 ] train risk: 0.146842  test risk: 0.157895
[ PAD C = 1.000000 ] train risk: 0.115789  test risk: 0.122632
[ PAD C = 10.000000 ] train risk: 0.110526  test risk: 0.123684
[ PAD C = 100.000000 ] train risk: 0.112632  test risk: 0.127368
[ PAD C = 1000.000000 ] train risk: 0.108947  test risk: 0.126842
[ PAD C = 10000.000000 ] train risk: 0.108947  test risk: 0.126316
PAD on DANN representation = 1.509474
================================================================
Computing PAD on original data...
PAD on (1800, 2000) examples
[ PAD C = 0.000010 ] train risk: 0.051053  test risk: 0.060526
[ PAD C = 0.000100 ] train risk: 0.017895  test risk: 0.026842
[ PAD C = 0.001000 ] train risk: 0.012632  test risk: 0.018421
[ PAD C = 0.010000 ] train risk: 0.001053  test risk: 0.021579
[ PAD C = 0.100000 ] train risk: 0.000000  test risk: 0.027895
[ PAD C = 1.000000 ] train risk: 0.000000  test risk: 0.027895
[ PAD C = 10.000000 ] train risk: 0.000000  test risk: 0.027895
[ PAD C = 100.000000 ] train risk: 0.000000  test risk: 0.027895
[ PAD C = 1000.000000 ] train risk: 0.000000  test risk: 0.027895
[ PAD C = 10000.000000 ] train risk: 0.000000  test risk: 0.027895
PAD on original data = 1.926316
```

- With adversarial

```
Loading data...
source file: ../data/Office-31/Amazon/dvd_train.svmlight
target file: ../data/Office-31/Amazon/electronics_train.svmlight
test file:   ../data/Office-31/Amazon/electronics_test.svmlight
mSDA representations loaded from disk
Fit...
[DANN parameters] {'hidden_layer_size': 50, 'maxiter': 200, 'lambda_
adapt': 0.1, 'epsilon_init': None, 'learning_rate': 0.0001, 'adversa
rial_representation': True, 'seed': 12342, 'verbose': True}
[DANN best valid risk so far] 0.255000 (iter 0)
[DANN best valid risk so far] 0.240000 (iter 1)
[DANN best valid risk so far] 0.230000 (iter 2)
[DANN best valid risk so far] 0.215000 (iter 3)
[DANN best valid risk so far] 0.210000 (iter 4)
[DANN best valid risk so far] 0.210000 (iter 5)
[DANN best valid risk so far] 0.210000 (iter 6)
[DANN best valid risk so far] 0.210000 (iter 7)
[DANN best valid risk so far] 0.210000 (iter 8)
[DANN best valid risk so far] 0.210000 (iter 9)
[DANN best valid risk so far] 0.210000 (iter 10)
[DANN best valid risk so far] 0.210000 (iter 12)
[DANN best valid risk so far] 0.210000 (iter 13)
[DANN best valid risk so far] 0.210000 (iter 14)
[DANN best valid risk so far] 0.210000 (iter 15)
[DANN best valid risk so far] 0.210000 (iter 16)
[DANN best valid risk so far] 0.210000 (iter 19)
[DANN early stop] iter 31
Predict...
Training Risk   = 0.125000
Validation Risk = 0.210000
Test Risk       = 0.223552
======================================================================
```

```
Computing PAD on DANN representation...
PAD on (1800, 2000) examples
[ PAD C = 0.000010 ] train risk: 0.473684  test risk: 0.473684
[ PAD C = 0.000100 ] train risk: 0.473684  test risk: 0.473684
[ PAD C = 0.001000 ] train risk: 0.473684  test risk: 0.473684
[ PAD C = 0.010000 ] train risk: 0.362632  test risk: 0.365789
[ PAD C = 0.100000 ] train risk: 0.207368  test risk: 0.221579
[ PAD C = 1.000000 ] train risk: 0.153684  test risk: 0.162105
[ PAD C = 10.000000 ] train risk: 0.144211  test risk: 0.157895
[ PAD C = 100.000000 ] train risk: 0.138947  test risk: 0.154737
[ PAD C = 1000.000000 ] train risk: 0.141579  test risk: 0.154737
[ PAD C = 10000.000000 ] train risk: 0.143158  test risk: 0.157895
PAD on DANN representation = 1.381053
======================================================================
Computing PAD on original data...
PAD on (1800, 2000) examples
[ PAD C = 0.000010 ] train risk: 0.051053  test risk: 0.060526
[ PAD C = 0.000100 ] train risk: 0.017895  test risk: 0.026842
[ PAD C = 0.001000 ] train risk: 0.012632  test risk: 0.018421
[ PAD C = 0.010000 ] train risk: 0.001053  test risk: 0.021579
[ PAD C = 0.100000 ] train risk: 0.000000  test risk: 0.027895
[ PAD C = 1.000000 ] train risk: 0.000000  test risk: 0.027895
[ PAD C = 10.000000 ] train risk: 0.000000  test risk: 0.027895
[ PAD C = 100.000000 ] train risk: 0.000000  test risk: 0.027895
[ PAD C = 1000.000000 ] train risk: 0.000000  test risk: 0.027895
[ PAD C = 10000.000000 ] train risk: 0.000000  test risk: 0.027895
PAD on original data = 1.926316
```

- Without mSDA
  - Without adversarial

```
Loading data...
source file: ../data/Office-31/Amazon/dvd_train.svmlight
target file: ../data/Office-31/Amazon/electronics_train.svmlight
test file:   ../data/Office-31/Amazon/electronics_test.svmlight
Fit...
[DANN parameters] {'hidden_layer_size': 50, 'maxiter': 200, 'lambda_adapt': 0.0, 'epsilo
n_init': None, 'learning_rate': 0.001, 'adversarial_representation': False, 'seed': 1234
2, 'verbose': True}
[DANN best valid risk so far] 0.365000 (iter 0)
[DANN best valid risk so far] 0.320000 (iter 1)
[DANN best valid risk so far] 0.275000 (iter 2)
[DANN best valid risk so far] 0.260000 (iter 3)
[DANN best valid risk so far] 0.240000 (iter 4)
[DANN best valid risk so far] 0.235000 (iter 5)
[DANN best valid risk so far] 0.220000 (iter 6)
[DANN best valid risk so far] 0.210000 (iter 7)
[DANN best valid risk so far] 0.210000 (iter 8)
[DANN best valid risk so far] 0.195000 (iter 9)
[DANN best valid risk so far] 0.195000 (iter 10)
[DANN best valid risk so far] 0.195000 (iter 12)
[DANN best valid risk so far] 0.195000 (iter 13)
[DANN best valid risk so far] 0.195000 (iter 16)
[DANN best valid risk so far] 0.195000 (iter 17)
[DANN best valid risk so far] 0.195000 (iter 18)
[DANN best valid risk so far] 0.195000 (iter 19)
[DANN best valid risk so far] 0.195000 (iter 21)
[DANN best valid risk so far] 0.195000 (iter 22)
[DANN best valid risk so far] 0.195000 (iter 23)
[DANN best valid risk so far] 0.195000 (iter 24)
[DANN best valid risk so far] 0.195000 (iter 25)
[DANN best valid risk so far] 0.195000 (iter 26)
[DANN best valid risk so far] 0.195000 (iter 27)
[DANN best valid risk so far] 0.195000 (iter 28)
[DANN best valid risk so far] 0.195000 (iter 43)
[DANN best valid risk so far] 0.195000 (iter 44)
[DANN best valid risk so far] 0.195000 (iter 45)
[DANN best valid risk so far] 0.195000 (iter 46)
[DANN best valid risk so far] 0.195000 (iter 70)
[DANN best valid risk so far] 0.195000 (iter 71)
[DANN best valid risk so far] 0.195000 (iter 72)
[DANN best valid risk so far] 0.195000 (iter 87)
```

```
[DANN best valid risk so far] 0.190000 (iter 104)
[DANN best valid risk so far] 0.190000 (iter 151)
[DANN best valid risk so far] 0.190000 (iter 152)
[DANN best valid risk so far] 0.190000 (iter 153)
[DANN best valid risk so far] 0.190000 (iter 154)
[DANN best valid risk so far] 0.190000 (iter 155)
[DANN best valid risk so far] 0.190000 (iter 156)
[DANN best valid risk so far] 0.190000 (iter 157)
[DANN best valid risk so far] 0.190000 (iter 158)
Predict...
Training Risk   = 0.000000
Validation Risk = 0.190000
Test Risk       = 0.265798
================================================================
Computing PAD on DANN representation...
PAD on (1800, 2000) examples
[ PAD C = 0.000010 ] train risk: 0.473684  test risk: 0.473684
[ PAD C = 0.000100 ] train risk: 0.473684  test risk: 0.473684
[ PAD C = 0.001000 ] train risk: 0.473684  test risk: 0.473684
[ PAD C = 0.010000 ] train risk: 0.471053  test risk: 0.467368
[ PAD C = 0.100000 ] train risk: 0.337368  test risk: 0.337368
[ PAD C = 1.000000 ] train risk: 0.274737  test risk: 0.287895
[ PAD C = 10.000000 ] train risk: 0.268947  test risk: 0.285789
[ PAD C = 100.000000 ] train risk: 0.273684  test risk: 0.287368
[ PAD C = 1000.000000 ] train risk: 0.272632  test risk: 0.288421
[ PAD C = 10000.000000 ] train risk: 0.273684  test risk: 0.287368
PAD on DANN representation = 0.856842
================================================================
Computing PAD on original data...
PAD on (1800, 2000) examples
[ PAD C = 0.000010 ] train risk: 0.473684  test risk: 0.473684
[ PAD C = 0.000100 ] train risk: 0.366316  test risk: 0.350526
[ PAD C = 0.001000 ] train risk: 0.104737  test risk: 0.125789
[ PAD C = 0.010000 ] train risk: 0.024737  test risk: 0.052105
[ PAD C = 0.100000 ] train risk: 0.000526  test risk: 0.036842
[ PAD C = 1.000000 ] train risk: 0.000000  test risk: 0.041053
[ PAD C = 10.000000 ] train risk: 0.000000  test risk: 0.041053
[ PAD C = 100.000000 ] train risk: 0.000000  test risk: 0.041053
[ PAD C = 1000.000000 ] train risk: 0.000000  test risk: 0.041053
[ PAD C = 10000.000000 ] train risk: 0.000000  test risk: 0.041053
PAD on original data = 1.852632
```

- With adversarial

```
Loading data...
source file: ../data/Office-31/Amazon/dvd_train.svmlight
target file: ../data/Office-31/Amazon/electronics_train.svmlight
test file:   ../data/Office-31/Amazon/electronics_test.svmlight
Fit...
[DANN parameters] {'hidden_layer_size': 50, 'maxiter': 200, 'lambd
a_adapt': 0.1, 'epsilon_init': None, 'learning_rate': 0.001, 'adve
rsarial_representation': True, 'seed': 12342, 'verbose': True}
[DANN best valid risk so far] 0.365000 (iter 0)
[DANN best valid risk so far] 0.320000 (iter 1)
[DANN best valid risk so far] 0.275000 (iter 2)
[DANN best valid risk so far] 0.260000 (iter 3)
[DANN best valid risk so far] 0.240000 (iter 4)
[DANN best valid risk so far] 0.235000 (iter 5)
[DANN best valid risk so far] 0.220000 (iter 6)
[DANN best valid risk so far] 0.210000 (iter 7)
[DANN best valid risk so far] 0.210000 (iter 8)
[DANN best valid risk so far] 0.205000 (iter 9)
[DANN best valid risk so far] 0.195000 (iter 10)
[DANN best valid risk so far] 0.195000 (iter 12)
[DANN best valid risk so far] 0.195000 (iter 13)
[DANN best valid risk so far] 0.195000 (iter 15)
[DANN best valid risk so far] 0.195000 (iter 16)
[DANN best valid risk so far] 0.195000 (iter 17)
[DANN best valid risk so far] 0.195000 (iter 23)
[DANN best valid risk so far] 0.195000 (iter 24)
[DANN best valid risk so far] 0.195000 (iter 25)
[DANN best valid risk so far] 0.195000 (iter 26)
[DANN best valid risk so far] 0.195000 (iter 27)
[DANN best valid risk so far] 0.195000 (iter 28)
[DANN best valid risk so far] 0.195000 (iter 29)
[DANN best valid risk so far] 0.195000 (iter 42)
[DANN early stop] iter 64
Predict...
Training Risk   = 0.026667
Validation Risk = 0.195000
Test Risk       = 0.254357
========================================================================
```

```
================================================================
Computing PAD on DANN representation...
PAD on (1800, 2000) examples
[ PAD C = 0.000010 ] train risk: 0.473684  test risk: 0.473684
[ PAD C = 0.000100 ] train risk: 0.473684  test risk: 0.473684
[ PAD C = 0.001000 ] train risk: 0.473684  test risk: 0.473684
[ PAD C = 0.010000 ] train risk: 0.473684  test risk: 0.473684
[ PAD C = 0.100000 ] train risk: 0.372105  test risk: 0.379474
[ PAD C = 1.000000 ] train risk: 0.265263  test risk: 0.285263
[ PAD C = 10.000000 ] train risk: 0.265263  test risk: 0.279474
[ PAD C = 100.000000 ] train risk: 0.265789  test risk: 0.277895
[ PAD C = 1000.000000 ] train risk: 0.264737  test risk: 0.276842
[ PAD C = 10000.000000 ] train risk: 0.263158  test risk: 0.277368
PAD on DANN representation = 0.892632
================================================================
Computing PAD on original data...
PAD on (1800, 2000) examples
[ PAD C = 0.000010 ] train risk: 0.473684  test risk: 0.473684
[ PAD C = 0.000100 ] train risk: 0.366316  test risk: 0.350526
[ PAD C = 0.001000 ] train risk: 0.104737  test risk: 0.125789
[ PAD C = 0.010000 ] train risk: 0.024737  test risk: 0.052105
[ PAD C = 0.100000 ] train risk: 0.000526  test risk: 0.036842
[ PAD C = 1.000000 ] train risk: 0.000000  test risk: 0.041053
[ PAD C = 10.000000 ] train risk: 0.000000  test risk: 0.041053
[ PAD C = 100.000000 ] train risk: 0.000000  test risk: 0.041053
[ PAD C = 1000.000000 ] train risk: 0.000000  test risk: 0.041053
[ PAD C = 10000.000000 ] train risk: 0.000000  test risk: 0.041053
PAD on original data = 1.852632
```

Compare

Let's look at the results in tabular form for comparison.

		Without adversarial	**With adversarial**
With mSDA	Training Risk	0.124444	0.125000
	Validation Risk	0.210000	0.210000
	Test Risk	0.231121	**e**
	PAD on DANN	1.509474	1.381053
	PAD on original data	1.926316	1.926316
	Iteration numbers	**19**	**19**
Without mSDA	Training Risk	**0.000000**	0.026667
	Validation Risk	**0.190000**	0.195000
	Test Risk	0.265798	0.254357
	PAD on DANN	0.856842	0.892632
	PAD on original data	1.852632	1.852632
	Iteration numbers	158	42

> In paper: D->E
>
> DANN on Original data: ≈ 1.25
>
> DANN & NN with 100 hidden neurons: ≈ 0.6
>
> DANN on mSDA representation: ≈ 1.0

First we can see that mSDA can accelerate the process of convergence and also can increase PAD. (PAD is a metric estimating the **similarity** of the source and the target representations.)

In the absence of mSDA and no adversarial, although there is no error in the training set, in the test set, the error is relatively large and there is a tendency to overfitting.

In these four cases, **when there is both mSDA and confrontation,** the effect is best. The convergence speed is fast and the test results are good.

But at the same time, we can also see that there is no difference of more than 0.02 between the test results in the four cases. Therefore, it also illustrates **the generalization of DANN**. And its design is simple and can be attached to many previous models to improve performance.

Compared with datasets

	In my test
U->M	77.64
M->U	89.74
S->M	94.77

Compared with data in Amazon

	In paper	In my test
Dvd -> Books	72.3	79.0
Dvd -> Electronics	75.4	77.64
Dvd -> Kitchen	78.3	79.0
Electronics -> Books	71.3	79.0
Electronics -> Dvd	73.8	79.0
Electronics -> Kitchen	85.4	79.0
Books -> Dvd	78.4	79.6
Books -> Electronics	73.3	75.2
Books -> kitchen	77.9	80.3
Kitchen -> Books	70.9	72.3
Kitchen -> Dvd	74.0	75.5
Kitchen -> Electronics	84.3	86.9

ADA

Most important is the `associated loss`.

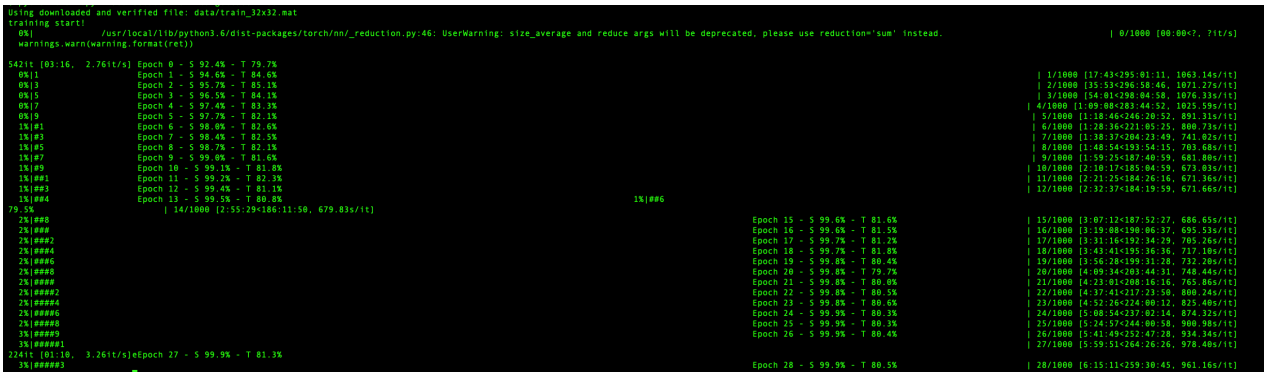
```
class WalkerLoss(nn.Module):

    def forward(self, Psts, y):
        equality_matrix = torch.eq(y.clone().view(-1, 1), y).float()
        p_target = equality_matrix / equality_matrix.sum(dim=1, keepdim=True)
```

```
        p_target.requires_grad = False

        L_walker = F.kl_div(torch.log(1e-8 + Psts), p_target,
size_average=False)
        L_walker /= p_target.size()[0]

        return L_walker


class VisitLoss(nn.Module):

    def forward(self, Pt):
        p_visit = torch.ones([1, Pt.size()[1]]) / float(Pt.size()[1])
        p_visit.requires_grad = False
        if Pt.is_cuda: p_visit = p_visit.cuda()
        L_visit = F.kl_div(torch.log(1e-8 + Pt), p_visit, size_average=False)
        L_visit /= p_visit.size()[0]

        return L_visit
```

Due to the space problem, the setting of"`source domain 100, target domain 1000`"is not used, instead is"`source domain 20, target domain 200`".

The result as follows:

`mnist->svhn: (not int paper)`



In the `first iteration`, you can get more than `90%` in the training set and nearly `80%` in the test set. Obviously, it's little overfitting. But the speed and accuracy is good, respectively. Maybe in the setting of "`source domain 100, target domain 1000`", the result will be good.

Compare

Method	Domains (source → target)			
	MNIST → MNIST-M	Syn. Digits → SVHN	SVHN → MNIST	Syn. Signs → GTSRB
Transf. Repr. [22]	13.30	-	21.20	-
SA [8]	43.10	13.56	40.68	18.35
CORAL [24]	42.30	14.80	36.90	13.10
ADDA [30]	-	-	24.00	-
DANN [9]	23.33 (55.87 %)	8.91 (79.67 %)	26.15 (42.57 %)	11.35 (46.39 %)
DSN w/ DANN [3]	16.80 (63.18 %)	8.80 (78.95 %)	17.30 (58.31 %)	6.90 (54.42 %)
DSN w/ MMD [3]	19.50 (56.77 %)	11.50 (31.58 %)	27.80 (32.26 %)	7.40 (51.02 %)
MMD [15]	23.10	12.00	28.90	8.90
DA_{MMD}	22.90	19.14	28.48	10.69
Ours (DA_{assoc} fixed params†)	**10.47 ± 0.28**	8.70 ± 0.2	4.32 ± 1.54	17.20 ± 1.32
Ours (DA_{assoc})	10.53 (85.94 %)	**8.14 (87.78 %)**	**2.40 (93.71 %)**	**2.34 (81.23)**
Source only	35.96	15.68	30.71	4.59
Target only	6.37	7.09	0.50	1.82

Since I used the data in `torchvision.datasets`, there are only svhn and mnist, so I didn't experiment with other datasets. It only needs to convert the data, but because of the long test time and big space of one data set in this experiment, so if time permits, I will add them.

After use other dataset and transform it, get `Mnist to Usps` is 95.79.

MCD_UDA

The author used a `three-layer fully-joined network` with a `batch size set to 32` and optimized the model `using a SGD` with a `learning rate of 1e-3`. And report accuracy after every ten iterations

The author gives the MNIST data.

~~If you run an experiment on adaptation from svhn to mnist, where num_k indicates the number of update for generator.~~

```
python main.py --source svhn --target mnist --num_k 3
```

~~If you want to run an experiment **using gradient reversal layer,** simply add option --one_step when running this code.~~

```
python main.py --source svhn --target mnist --one_step
```

Since I integrated the code, please use `python main.py --model='MCD_UDA'` and You can set other parameters, as explained above.

I chose two sets of experiments, one is `mnist to usps`; the other is `svhn to mnist`. And for the former to do a `three-layer fully connected network` and `using gradient reversal layer` comparison; for the latter to do a `three-layer`and`four-layer` full-join network comparison. Due to the high number of iterations, only the first and last two result images are released here. The complete results are in `results_image/MCD_UDA_record/`.

Svhn to mnist

Three layer fully connected network

```
$ python main.py --source svhn --target mnist --num_k 3
Namespace(all_use='no', batch_size=128, checkpoint_dir='checkpoint', cuda=True, eval_only=False, lr=0.0002, max_epoch=200, no_cuda=False, num_k=3, one_step=False, optimizer='adam', resume_epoch=100, save_epoch=10, save_model=False, seed=1, source='svhn', target='mnist', use_abs_diff=False)
dataset loading
(55000, 3, 32, 32)
/usr/local/lib/python3.6/dist-packages/torchvision/transforms/transforms.py:208: UserWarning: The use of the transforms.Scale transform is deprecated, please use transforms.Resize instead.
  "please use transforms.Resize instead.")
load finished!
/home/huangchenxi/transfer-learning/MCD_UDA/solver.py:90: UserWarning: Implicit dimension choice for softmax has been deprecated. Change the call to include dim=X as an argument.
  return torch.mean(torch.abs(F.softmax(out1) - F.softmax(out2)))
Train Epoch: 0 [0/100 (0%)]     Loss1: 1.911990  Loss2: 1.88967  Discrepancy: 0.021790
Train Epoch: 0 [100/100 (0%)]   Loss1: 0.557463  Loss2: 0.56178  Discrepancy: 0.008060
Train Epoch: 0 [200/100 (0%)]   Loss1: 0.406073  Loss2: 0.39611  Discrepancy: 0.008883
Train Epoch: 0 [300/100 (0%)]   Loss1: 0.305152  Loss2: 0.29400  Discrepancy: 0.011216
Train Epoch: 0 [400/100 (1%)]   Loss1: 0.346556  Loss2: 0.34085  Discrepancy: 0.009077
/home/huangchenxi/transfer-learning/MCD_UDA/solver.py:229: UserWarning: volatile was removed and now has no effect. Use `with torch.no_grad():` instead.
  img, label = Variable(img, volatile=True), Variable(label)

Test set: Average loss: -0.0371, Accuracy C1: 6837/10000 (68%) Accuracy C2: 6838/10000 (68%) Accuracy Ensemble: 6861/10000 (68%)

recording %s record/svhn_mnist_k_3_onestep_False_1_test.txt
Train Epoch: 1 [0/100 (0%)]     Loss1: 0.309628  Loss2: 0.30610  Discrepancy: 0.011315
Train Epoch: 1 [100/100 (0%)]   Loss1: 0.303234  Loss2: 0.31203  Discrepancy: 0.012625
Train Epoch: 1 [200/100 (0%)]   Loss1: 0.270616  Loss2: 0.28401  Discrepancy: 0.008394
Train Epoch: 1 [300/100 (0%)]   Loss1: 0.375139  Loss2: 0.38299  Discrepancy: 0.009320
Train Epoch: 1 [400/100 (1%)]   Loss1: 0.458935  Loss2: 0.45941  Discrepancy: 0.009006

Test set: Average loss: -0.0412, Accuracy C1: 7630/10000 (76%) Accuracy C2: 7587/10000 (75%) Accuracy Ensemble: 7607/10000 (76%)

recording %s record/svhn_mnist_k_3_onestep_False_1_test.txt
Train Epoch: 2 [0/100 (0%)]     Loss1: 0.237049  Loss2: 0.24444  Discrepancy: 0.007628
Train Epoch: 2 [100/100 (0%)]   Loss1: 0.331287  Loss2: 0.325654          Discrepancy: 0.007541
Train Epoch: 2 [200/100 (0%)]   Loss1: 0.459615  Loss2: 0.463207          Discrepancy: 0.009048
Train Epoch: 2 [300/100 (0%)]   Loss1: 0.256926  Loss2: 0.246781          Discrepancy: 0.009858
Train Epoch: 2 [400/100 (1%)]   Loss1: 0.259559  Loss2: 0.258995          Discrepancy: 0.009695

Test set: Average loss: -0.0394, Accuracy C1: 7923/10000 (79%) Accuracy C2: 7901/10000 (79%) Accuracy Ensemble: 7911/10000 (79%)

recording %s record/svhn_mnist_k_3_onestep_False_1_test.txt
Train Epoch: 3 [0/100 (0%)]     Loss1: 0.179793  Loss2: 0.182681          Discrepancy: 0.007457
Train Epoch: 3 [100/100 (0%)]   Loss1: 0.189167  Loss2: 0.186856          Discrepancy: 0.008367
Train Epoch: 3 [200/100 (0%)]   Loss1: 0.439096  Loss2: 0.449638          Discrepancy: 0.008867
```

```
Train Epoch: 43 [200/100 (0%)]  Loss1: 0.108076  Loss2: 0.105094          Disc
repancy: 0.001884
Train Epoch: 43 [300/100 (0%)]  Loss1: 0.070828  Loss2: 0.069617          Disc
repancy: 0.001500
Train Epoch: 43 [400/100 (1%)]  Loss1: 0.064212  Loss2: 0.066789          Disc
repancy: 0.001651

Test set: Average loss: -0.0773, Accuracy C1: 9591/10000 (95%) Accuracy C2: 95
93/10000 (95%) Accuracy Ensemble: 9588/10000 (95%)

recording %s record/svhn_mnist_k_3_onestep_False_1_test.txt
Train Epoch: 44 [0/100 (0%)]    Loss1: 0.032637  Loss2: 0.031198          Disc
repancy: 0.001405
Train Epoch: 44 [100/100 (0%)]  Loss1: 0.047018  Loss2: 0.046000          Disc
repancy: 0.002637
Train Epoch: 44 [200/100 (0%)]  Loss1: 0.083661  Loss2: 0.084633          Disc
repancy: 0.002645
Train Epoch: 44 [300/100 (0%)]  Loss1: 0.098382  Loss2: 0.095631          Disc
repancy: 0.002202
Train Epoch: 44 [400/100 (1%)]  Loss1: 0.071495  Loss2: 0.075741          Disc
repancy: 0.002169

Test set: Average loss: -0.0796, Accuracy C1: 9592/10000 (95%) Accuracy C2: 95
77/10000 (95%) Accuracy Ensemble: 9585/10000 (95%)

recording %s record/svhn_mnist_k_3_onestep_False_1_test.txt
Train Epoch: 45 [0/100 (0%)]    Loss1: 0.052420  Loss2: 0.049829          Disc
repancy: 0.001848
Train Epoch: 45 [100/100 (0%)]  Loss1: 0.121637  Loss2: 0.124418          Disc
repancy: 0.001917
Train Epoch: 45 [200/100 (0%)]  Loss1: 0.075262  Loss2: 0.074892          Disc
repancy: 0.002314
Train Epoch: 45 [300/100 (0%)]  Loss1: 0.065439  Loss2: 0.062935          Disc
repancy: 0.001621
Train Epoch: 45 [400/100 (1%)]  Loss1: 0.079050  Loss2: 0.080480          Disc
repancy: 0.001592

Test set: Average loss: -0.0739, Accuracy C1: 9645/10000 (96%) Accuracy C2: 96
43/10000 (96%) Accuracy Ensemble: 9644/10000 (96%)

recording %s record/svhn_mnist_k_3_onestep_False_1_test.txt
Train Epoch: 46 [0/100 (0%)]    Loss1: 0.067377  Loss2: 0.068424          Disc
repancy: 0.001051
Train Epoch: 46 [100/100 (0%)]  Loss1: 0.104238  Loss2: 0.099360          Disc
repancy: 0.001753
Train Epoch: 46 [200/100 (0%)]  Loss1: 0.058262  Loss2: 0.057898          Disc
repancy: 0.001712
Train Epoch: 46 [300/100 (0%)]  Loss1: 0.086384  Loss2: 0.084947          Disc
repancy: 0.001438
Train Epoch: 46 [400/100 (1%)]  Loss1: 0.065899  Loss2: 0.067052          Disc
repancy: 0.001809

Test set: Average loss: -0.0785, Accuracy C1: 9624/10000 (96%) Accuracy C2: 96
50/10000 (96%) Accuracy Ensemble: 9640/10000 (96%)

recording %s record/svhn_mnist_k_3_onestep_False_1_test.txt

root at 641aa73707ea in /home/huangchenxi/transfer-learning/MCD_UDA
$
```

using gradient reversal layer

```
$ python main.py --source svhn --target mnist --one_step
Namespace(all_use='no', batch_size=128, checkpoint_dir='checkpoint', cuda=True, eval_only=False, lr=
0.0002, max_epoch=200, no_cuda=False, num_k=4, one_step=True, optimizer='adam', resume_epoch=100, sa
ve_epoch=10, save_model=False, seed=1, source='svhn', target='mnist', use_abs_diff=False)
dataset loading
(55000, 3, 32, 32)
/usr/local/lib/python3.6/dist-packages/torchvision/transforms/transforms.py:208: UserWarning: The us
e of the transforms.Scale transform is deprecated, please use transforms.Resize instead.
  "please use transforms.Resize instead.")
load finished!
/home/huangchenxi/transfer-learning/MCD_UDA/solver.py:90: UserWarning: Implicit dimension choice for
 softmax has been deprecated. Change the call to include dim=X as an argument.
  return torch.mean(torch.abs(F.softmax(out1) - F.softmax(out2)))
Train Epoch: 0 [0/100 (0%)]     Loss1: 2.268277  Loss2: 2.305896          Discrepancy: -0.024503
Train Epoch: 0 [100/100 (0%)]   Loss1: 0.713227  Loss2: 0.706736          Discrepancy: -0.006221
Train Epoch: 0 [200/100 (0%)]   Loss1: 0.387395  Loss2: 0.38860  Discrepancy: -0.006290
Train Epoch: 0 [300/100 (0%)]   Loss1: 0.397803  Loss2: 0.397506          Discrepancy: -0.005612
Train Epoch: 0 [400/100 (1%)]   Loss1: 0.365004  Loss2: 0.369847          Discrepancy: -0.004369
/home/huangchenxi/transfer-learning/MCD_UDA/solver.py:229: UserWarning: volatile was removed and now
 has no effect. Use `with torch.no_grad():` instead.
  img, label = Variable(img, volatile=True), Variable(label)

Test set: Average loss: -0.0269, Accuracy C1: 5740/10000 (57%) Accuracy C2: 5749/10000 (57%) Accurac
y Ensemble: 5746/10000 (57%)

recording %s record/svhn_mnist_k_4_onestep_True_0_test.txt
Train Epoch: 1 [0/100 (0%)]     Loss1: 0.404227  Loss2: 0.402911          Discrepancy: -0.005601
Train Epoch: 1 [100/100 (0%)]   Loss1: 0.322874  Loss2: 0.324709          Discrepancy: -0.004417
Train Epoch: 1 [200/100 (0%)]   Loss1: 0.313822  Loss2: 0.311150          Discrepancy: -0.004363
Train Epoch: 1 [300/100 (0%)]   Loss1: 0.391723  Loss2: 0.391865          Discrepancy: -0.004365
Train Epoch: 1 [400/100 (1%)]   Loss1: 0.444960  Loss2: 0.441041          Discrepancy: -0.004264

Test set: Average loss: -0.0313, Accuracy C1: 6472/10000 (64%) Accuracy C2: 6459/10000 (64%) Accurac
y Ensemble: 6467/10000 (64%)

recording %s record/svhn_mnist_k_4_onestep_True_0_test.txt
Train Epoch: 2 [0/100 (0%)]     Loss1: 0.235934  Loss2: 0.235478          Discrepancy: -0.004050
Train Epoch: 2 [100/100 (0%)]   Loss1: 0.288127  Loss2: 0.281232          Discrepancy: -0.003681
Train Epoch: 2 [200/100 (0%)]   Loss1: 0.517296  Loss2: 0.519623          Discrepancy: -0.004146
Train Epoch: 2 [300/100 (0%)]   Loss1: 0.278004  Loss2: 0.281306          Discrepancy: -0.002996
Train Epoch: 2 [400/100 (1%)]   Loss1: 0.269612  Loss2: 0.268115          Discrepancy: -0.003091

Test set: Average loss: -0.0313, Accuracy C1: 6145/10000 (61%) Accuracy C2: 6140/10000 (61%) Accurac
y Ensemble: 6139/10000 (61%)

recording %s record/svhn_mnist_k_4_onestep_True_0_test.txt
Train Epoch: 3 [0/100 (0%)]     Loss1: 0.275257  Loss2: 0.278027          Discrepancy: -0.003090
Train Epoch: 3 [100/100 (0%)]   Loss1: 0.162313  Loss2: 0.160310          Discrepancy: -0.004580
Train Epoch: 3 [200/100 (0%)]   Loss1: 0.364433  Loss2: 0.365316          Discrepancy: -0.002920
Train Epoch: 3 [300/100 (0%)]   Loss1: 0.350219  Loss2: 0.351968          Discrepancy: -0.002836
Train Epoch: 3 [400/100 (1%)]   Loss1: 0.125356  Loss2: 0.126044          Discrepancy: -0.003095

Test set: Average loss: -0.0339, Accuracy C1: 6388/10000 (63%) Accuracy C2: 6357/10000 (63%) Accurac
y Ensemble: 6372/10000 (63%)

recording %s record/svhn_mnist_k_4_onestep_True_0_test.txt
Train Epoch: 4 [0/100 (0%)]     Loss1: 0.186891  Loss2: 0.184971          Discrepancy: -0.003328
Train Epoch: 4 [100/100 (0%)]   Loss1: 0.232140  Loss2: 0.236099          Discrepancy: -0.002692
Train Epoch: 4 [200/100 (0%)]   Loss1: 0.248813  Loss2: 0.246535          Discrepancy: -0.002602
Train Epoch: 4 [300/100 (0%)]   Loss1: 0.247648  Loss2: 0.244619          Discrepancy: -0.003100
Train Epoch: 4 [400/100 (1%)]   Loss1: 0.265795  Loss2: 0.269968          Discrepancy: -0.002684

Test set: Average loss: -0.0319, Accuracy C1: 6604/10000 (66%) Accuracy C2: 6607/10000 (66%) Accurac
y Ensemble: 6606/10000 (66%)

recording %s record/svhn_mnist_k_4_onestep_True_0_test.txt
Train Epoch: 5 [0/100 (0%)]     Loss1: 0.315748  Loss2: 0.317907          Discrepancy: -0.002660
Train Epoch: 5 [100/100 (0%)]   Loss1: 0.180223  Loss2: 0.184398          Discrepancy: -0.002916
```

```
Test set: Average loss: -0.0433, Accuracy C1: 6629/10000 (66%) Accuracy C2: 6628/10000 (66%) Accurac
y Ensemble: 6623/10000 (66%)

recording %s record/svhn_mnist_k_4_onestep_True_0_test.txt
Train Epoch: 41 [0/100 (0%)]    Loss1: 0.108207  Loss2: 0.108199          Discrepancy: -0.001499
Train Epoch: 41 [100/100 (0%)]  Loss1: 0.016104  Loss2: 0.016335          Discrepancy: -0.001511
Train Epoch: 41 [200/100 (0%)]  Loss1: 0.032615  Loss2: 0.033705          Discrepancy: -0.001385
Train Epoch: 41 [300/100 (0%)]  Loss1: 0.025081  Loss2: 0.025062          Discrepancy: -0.001392
Train Epoch: 41 [400/100 (1%)]  Loss1: 0.051415  Loss2: 0.052083          Discrepancy: -0.001751

Test set: Average loss: -0.0443, Accuracy C1: 6806/10000 (68%) Accuracy C2: 6801/10000 (68%) Accurac
y Ensemble: 6803/10000 (68%)

recording %s record/svhn_mnist_k_4_onestep_True_0_test.txt
Train Epoch: 42 [0/100 (0%)]    Loss1: 0.015099  Loss2: 0.015437          Discrepancy: -0.001461
Train Epoch: 42 [100/100 (0%)]  Loss1: 0.048697  Loss2: 0.048781          Discrepancy: -0.001620
Train Epoch: 42 [200/100 (0%)]  Loss1: 0.032834  Loss2: 0.031839          Discrepancy: -0.001269
Train Epoch: 42 [300/100 (0%)]  Loss1: 0.037795  Loss2: 0.038357          Discrepancy: -0.001927
Train Epoch: 42 [400/100 (1%)]  Loss1: 0.038690  Loss2: 0.039036          Discrepancy: -0.001380

Test set: Average loss: -0.0498, Accuracy C1: 7128/10000 (71%) Accuracy C2: 7140/10000 (71%) Accurac
y Ensemble: 7136/10000 (71%)

recording %s record/svhn_mnist_k_4_onestep_True_0_test.txt
Train Epoch: 43 [0/100 (0%)]    Loss1: 0.111399  Loss2: 0.112127          Discrepancy: -0.001343
Train Epoch: 43 [100/100 (0%)]  Loss1: 0.039488  Loss2: 0.038493          Discrepancy: -0.001622
Train Epoch: 43 [200/100 (0%)]  Loss1: 0.049160  Loss2: 0.050125          Discrepancy: -0.001262
Train Epoch: 43 [300/100 (0%)]  Loss1: 0.026048  Loss2: 0.025455          Discrepancy: -0.001060
Train Epoch: 43 [400/100 (1%)]  Loss1: 0.066523  Loss2: 0.066126          Discrepancy: -0.001474

Test set: Average loss: -0.0474, Accuracy C1: 6965/10000 (69%) Accuracy C2: 6962/10000 (69%) Accurac
y Ensemble: 6965/10000 (69%)

recording %s record/svhn_mnist_k_4_onestep_True_0_test.txt
Train Epoch: 44 [0/100 (0%)]    Loss1: 0.024393  Loss2: 0.024379          Discrepancy: -0.001475
Train Epoch: 44 [100/100 (0%)]  Loss1: 0.031503  Loss2: 0.031038          Discrepancy: -0.001384
Train Epoch: 44 [200/100 (0%)]  Loss1: 0.053952  Loss2: 0.054666          Discrepancy: -0.001014
Train Epoch: 44 [300/100 (0%)]  Loss1: 0.090659  Loss2: 0.091470          Discrepancy: -0.001779
Train Epoch: 44 [400/100 (1%)]  Loss1: 0.056462  Loss2: 0.058524          Discrepancy: -0.001342

Test set: Average loss: -0.0481, Accuracy C1: 6966/10000 (69%) Accuracy C2: 6966/10000 (69%) Accurac
y Ensemble: 6969/10000 (69%)

recording %s record/svhn_mnist_k_4_onestep_True_0_test.txt
Train Epoch: 45 [0/100 (0%)]    Loss1: 0.031661  Loss2: 0.032142          Discrepancy: -0.001419
Train Epoch: 45 [100/100 (0%)]  Loss1: 0.081738  Loss2: 0.082430          Discrepancy: -0.001402
Train Epoch: 45 [200/100 (0%)]  Loss1: 0.050895  Loss2: 0.051133          Discrepancy: -0.001484
Train Epoch: 45 [300/100 (0%)]  Loss1: 0.070765  Loss2: 0.070488          Discrepancy: -0.001254
Train Epoch: 45 [400/100 (1%)]  Loss1: 0.066423  Loss2: 0.065869          Discrepancy: -0.001229

Test set: Average loss: -0.0494, Accuracy C1: 6863/10000 (68%) Accuracy C2: 6853/10000 (68%) Accurac
y Ensemble: 6854/10000 (68%)

recording %s record/svhn_mnist_k_4_onestep_True_0_test.txt
Train Epoch: 46 [0/100 (0%)]    Loss1: 0.057762  Loss2: 0.058632          Discrepancy: -0.001192
Train Epoch: 46 [100/100 (0%)]  Loss1: 0.028099  Loss2: 0.028188          Discrepancy: -0.001630
Train Epoch: 46 [200/100 (0%)]  Loss1: 0.025466  Loss2: 0.025755          Discrepancy: -0.001336
Train Epoch: 46 [300/100 (0%)]  Loss1: 0.033726  Loss2: 0.033338          Discrepancy: -0.001717
Train Epoch: 46 [400/100 (1%)]  Loss1: 0.053387  Loss2: 0.053617          Discrepancy: -0.001337

Test set: Average loss: -0.0467, Accuracy C1: 6762/10000 (67%) Accuracy C2: 6758/10000 (67%) Accurac
y Ensemble: 6759/10000 (67%)

recording %s record/svhn_mnist_k_4_onestep_True_0_test.txt

root at 60a5497e4409 in /home/huangchenxi/transfer-learning/MCD_UDA
$
```

Mnist to usps

Three layer fully connected network

```
$ python main.py --source mnist --target usps --num_k 3
Namespace(all_use='no', batch_size=128, checkpoint_dir='checkpoint', cuda=True, eval_only=False, lr=0.0002, max
_epoch=200, no_cuda=False, num_k=3, one_step=False, optimizer='adam', resume_epoch=100, save_epoch=10, save_mod
el=False, seed=1, source='mnist', target='usps', use_abs_diff=False)
dataset loading
(2000, 1, 28, 28)
/usr/local/lib/python3.6/dist-packages/torchvision/transforms/transforms.py:208: UserWarning: The use of the tr
ansforms.Scale transform is deprecated, please use transforms.Resize instead.
  "please use transforms.Resize instead.")
load finished!
/home/huangchenxi/transfer-learning/MCD_UDA/solver.py:90: UserWarning: Implicit dimension choice for softmax ha
s been deprecated. Change the call to include dim=X as an argument.
  return torch.mean(torch.abs(F.softmax(out1) - F.softmax(out2)))
Train Epoch: 0 [0/100 (0%)]     Loss1: 2.382223  Loss2: 2.382314          Discrepancy: 0.036035
/home/huangchenxi/transfer-learning/MCD_UDA/solver.py:229: UserWarning: volatile was removed and now has no eff
ect. Use `with torch.no_grad():` instead.
  img, label = Variable(img, volatile=True), Variable(label)

Test set: Average loss: -0.0002, Accuracy C1: 712/1860 (38%) Accuracy C2: 491/1860 (26%) Accuracy Ensemble: 692
/1860 (37%)

recording %s record/mnist_usps_k_3_alluse_no_onestep_False_0_test.txt
Train Epoch: 1 [0/100 (0%)]     Loss1: 2.216476  Loss2: 2.210045          Discrepancy: 0.034149

Test set: Average loss: -0.0011, Accuracy C1: 968/1860 (52%) Accuracy C2: 693/1860 (37%) Accuracy Ensemble: 829
/1860 (44%)

recording %s record/mnist_usps_k_3_alluse_no_onestep_False_0_test.txt
Train Epoch: 2 [0/100 (0%)]     Loss1: 2.124503  Loss2: 2.066645          Discrepancy: 0.033762

Test set: Average loss: -0.0017, Accuracy C1: 993/1860 (53%) Accuracy C2: 785/1860 (42%) Accuracy Ensemble: 889
/1860 (47%)

recording %s record/mnist_usps_k_3_alluse_no_onestep_False_0_test.txt
Train Epoch: 3 [0/100 (0%)]     Loss1: 1.975672  Loss2: 1.926368          Discrepancy: 0.037194

Test set: Average loss: -0.0025, Accuracy C1: 951/1860 (51%) Accuracy C2: 823/1860 (44%) Accuracy Ensemble: 887
/1860 (47%)

recording %s record/mnist_usps_k_3_alluse_no_onestep_False_0_test.txt
Train Epoch: 4 [0/100 (0%)]     Loss1: 1.919788  Loss2: 1.755128          Discrepancy: 0.039241

Test set: Average loss: -0.0032, Accuracy C1: 988/1860 (53%) Accuracy C2: 924/1860 (49%) Accuracy Ensemble: 962
/1860 (51%)

recording %s record/mnist_usps_k_3_alluse_no_onestep_False_0_test.txt
Train Epoch: 5 [0/100 (0%)]     Loss1: 1.657431  Loss2: 1.705456          Discrepancy: 0.040850

Test set: Average loss: -0.0039, Accuracy C1: 1028/1860 (55%) Accuracy C2: 938/1860 (50%) Accuracy Ensemble: 98
8/1860 (53%)

recording %s record/mnist_usps_k_3_alluse_no_onestep_False_0_test.txt
Train Epoch: 6 [0/100 (0%)]     Loss1: 1.614554  Loss2: 1.560779          Discrepancy: 0.043945

Test set: Average loss: -0.0047, Accuracy C1: 1105/1860 (59%) Accuracy C2: 1037/1860 (55%) Accuracy Ensemble: 1
089/1860 (58%)

recording %s record/mnist_usps_k_3_alluse_no_onestep_False_0_test.txt
Train Epoch: 7 [0/100 (0%)]     Loss1: 1.541687  Loss2: 1.541064          Discrepancy: 0.048636

Test set: Average loss: -0.0052, Accuracy C1: 1054/1860 (56%) Accuracy C2: 1015/1860 (54%) Accuracy Ensemble: 1
037/1860 (55%)

recording %s record/mnist_usps_k_3_alluse_no_onestep_False_0_test.txt
Train Epoch: 8 [0/100 (0%)]     Loss1: 1.494078  Loss2: 1.384127          Discrepancy: 0.050657

Test set: Average loss: -0.0059, Accuracy C1: 1095/1860 (58%) Accuracy C2: 1077/1860 (57%) Accuracy Ensemble: 1
098/1860 (59%)
```

```
Test set: Average loss: -0.0492, Accuracy C1: 1762/1860 (94%) Accuracy C2: 1778/1860 (95%) Accuracy Ensemble: 1
768/1860 (95%)

recording %s record/mnist_usps_k_3_alluse_no_onestep_False_0_test.txt
Train Epoch: 190 [0/100 (0%)]   Loss1: 0.011243  Loss2: 0.010493          Discrepancy: 0.011267

Test set: Average loss: -0.0491, Accuracy C1: 1757/1860 (94%) Accuracy C2: 1774/1860 (95%) Accuracy Ensemble: 1
765/1860 (94%)

recording %s record/mnist_usps_k_3_alluse_no_onestep_False_0_test.txt
Train Epoch: 191 [0/100 (0%)]   Loss1: 0.014260  Loss2: 0.018318          Discrepancy: 0.017557

Test set: Average loss: -0.0497, Accuracy C1: 1746/1860 (93%) Accuracy C2: 1768/1860 (95%) Accuracy Ensemble: 1
761/1860 (94%)

recording %s record/mnist_usps_k_3_alluse_no_onestep_False_0_test.txt
Train Epoch: 192 [0/100 (0%)]   Loss1: 0.007250  Loss2: 0.031433          Discrepancy: 0.012802

Test set: Average loss: -0.0484, Accuracy C1: 1750/1860 (94%) Accuracy C2: 1764/1860 (94%) Accuracy Ensemble: 1
757/1860 (94%)

recording %s record/mnist_usps_k_3_alluse_no_onestep_False_0_test.txt
Train Epoch: 193 [0/100 (0%)]   Loss1: 0.011271  Loss2: 0.007898          Discrepancy: 0.021786

Test set: Average loss: -0.0490, Accuracy C1: 1745/1860 (93%) Accuracy C2: 1769/1860 (95%) Accuracy Ensemble: 1
765/1860 (94%)

recording %s record/mnist_usps_k_3_alluse_no_onestep_False_0_test.txt
Train Epoch: 194 [0/100 (0%)]   Loss1: 0.014166  Loss2: 0.013017          Discrepancy: 0.022087

Test set: Average loss: -0.0483, Accuracy C1: 1762/1860 (94%) Accuracy C2: 1780/1860 (95%) Accuracy Ensemble: 1
770/1860 (95%)

recording %s record/mnist_usps_k_3_alluse_no_onestep_False_0_test.txt
Train Epoch: 195 [0/100 (0%)]   Loss1: 0.024037  Loss2: 0.013002          Discrepancy: 0.014850

Test set: Average loss: -0.0508, Accuracy C1: 1766/1860 (94%) Accuracy C2: 1786/1860 (96%) Accuracy Ensemble: 1
781/1860 (95%)

recording %s record/mnist_usps_k_3_alluse_no_onestep_False_0_test.txt
Train Epoch: 196 [0/100 (0%)]   Loss1: 0.020742  Loss2: 0.007589          Discrepancy: 0.018010

Test set: Average loss: -0.0491, Accuracy C1: 1750/1860 (94%) Accuracy C2: 1767/1860 (95%) Accuracy Ensemble: 1
765/1860 (94%)

recording %s record/mnist_usps_k_3_alluse_no_onestep_False_0_test.txt
Train Epoch: 197 [0/100 (0%)]   Loss1: 0.013722  Loss2: 0.006397          Discrepancy: 0.014234

Test set: Average loss: -0.0498, Accuracy C1: 1753/1860 (94%) Accuracy C2: 1775/1860 (95%) Accuracy Ensemble: 1
763/1860 (94%)

recording %s record/mnist_usps_k_3_alluse_no_onestep_False_0_test.txt
Train Epoch: 198 [0/100 (0%)]   Loss1: 0.011574  Loss2: 0.016353          Discrepancy: 0.012327

Test set: Average loss: -0.0480, Accuracy C1: 1742/1860 (93%) Accuracy C2: 1763/1860 (94%) Accuracy Ensemble: 1
759/1860 (94%)

recording %s record/mnist_usps_k_3_alluse_no_onestep_False_0_test.txt
Train Epoch: 199 [0/100 (0%)]   Loss1: 0.008313  Loss2: 0.015026          Discrepancy: 0.013231

Test set: Average loss: -0.0473, Accuracy C1: 1747/1860 (93%) Accuracy C2: 1765/1860 (94%) Accuracy Ensemble: 1
755/1860 (94%)

recording %s record/mnist_usps_k_3_alluse_no_onestep_False_0_test.txt

root at 263b4efd5a52 in /home/huangchenxi/transfer-learning/MCD_UDA
$
```

Four layer fully connected network

```
[$ python main.py --source mnist --target usps --num_k 4                                                       ]
Namespace(all_use='no', batch_size=128, checkpoint_dir='checkpoint', cuda=True, eval_only=False, lr=0.0002, max
_epoch=200, no_cuda=False, num_k=4, one_step=False, optimizer='adam', resume_epoch=100, save_epoch=10, save_mod
el=False, seed=1, source='mnist', target='usps', use_abs_diff=False)
dataset loading
(2000, 1, 28, 28)
/usr/local/lib/python3.6/dist-packages/torchvision/transforms/transforms.py:208: UserWarning: The use of the tr
ansforms.Scale transform is deprecated, please use transforms.Resize instead.
  "please use transforms.Resize instead.")
load finished!
/home/huangchenxi/transfer-learning/MCD_UDA/solver.py:90: UserWarning: Implicit dimension choice for softmax ha
s been deprecated. Change the call to include dim=X as an argument.
  return torch.mean(torch.abs(F.softmax(out1) - F.softmax(out2)))
Train Epoch: 0 [0/100 (0%)]     Loss1: 2.343074  Loss2: 2.354578          Discrepancy: 0.034134
/home/huangchenxi/transfer-learning/MCD_UDA/solver.py:229: UserWarning: volatile was removed and now has no eff
ect. Use `with torch.no_grad():` instead.
  img, label = Variable(img, volatile=True), Variable(label)

Test set: Average loss: -0.0003, Accuracy C1: 641/1860 (34%) Accuracy C2: 490/1860 (26%) Accuracy Ensemble: 719
/1860 (38%)

recording %s record/mnist_usps_k_4_alluse_no_onestep_False_0_test.txt
Train Epoch: 1 [0/100 (0%)]     Loss1: 2.217272  Loss2: 2.172659          Discrepancy: 0.033076

Test set: Average loss: -0.0011, Accuracy C1: 988/1860 (53%) Accuracy C2: 758/1860 (40%) Accuracy Ensemble: 915
/1860 (49%)

recording %s record/mnist_usps_k_4_alluse_no_onestep_False_0_test.txt
Train Epoch: 2 [0/100 (0%)]     Loss1: 2.168787  Loss2: 2.101144          Discrepancy: 0.033594

Test set: Average loss: -0.0019, Accuracy C1: 1033/1860 (55%) Accuracy C2: 889/1860 (47%) Accuracy Ensemble: 98
7/1860 (53%)

recording %s record/mnist_usps_k_4_alluse_no_onestep_False_0_test.txt
Train Epoch: 3 [0/100 (0%)]     Loss1: 1.990340  Loss2: 2.051481          Discrepancy: 0.035275

Test set: Average loss: -0.0027, Accuracy C1: 1093/1860 (58%) Accuracy C2: 982/1860 (52%) Accuracy Ensemble: 10
65/1860 (57%)

recording %s record/mnist_usps_k_4_alluse_no_onestep_False_0_test.txt
Train Epoch: 4 [0/100 (0%)]     Loss1: 1.879270  Loss2: 1.876694          Discrepancy: 0.036969

Test set: Average loss: -0.0035, Accuracy C1: 1062/1860 (57%) Accuracy C2: 1005/1860 (54%) Accuracy Ensemble: 1
056/1860 (56%)

recording %s record/mnist_usps_k_4_alluse_no_onestep_False_0_test.txt
Train Epoch: 5 [0/100 (0%)]     Loss1: 1.660003  Loss2: 1.677380          Discrepancy: 0.039429

Test set: Average loss: -0.0044, Accuracy C1: 1131/1860 (60%) Accuracy C2: 1094/1860 (58%) Accuracy Ensemble: 1
141/1860 (61%)

recording %s record/mnist_usps_k_4_alluse_no_onestep_False_0_test.txt
Train Epoch: 6 [0/100 (0%)]     Loss1: 1.609736  Loss2: 1.597398          Discrepancy: 0.042612

Test set: Average loss: -0.0052, Accuracy C1: 1160/1860 (62%) Accuracy C2: 1131/1860 (60%) Accuracy Ensemble: 1
163/1860 (62%)

recording %s record/mnist_usps_k_4_alluse_no_onestep_False_0_test.txt
Train Epoch: 7 [0/100 (0%)]     Loss1: 1.582161  Loss2: 1.559811          Discrepancy: 0.047276

Test set: Average loss: -0.0060, Accuracy C1: 1222/1860 (65%) Accuracy C2: 1201/1860 (64%) Accuracy Ensemble: 1
236/1860 (66%)

recording %s record/mnist_usps_k_4_alluse_no_onestep_False_0_test.txt
Train Epoch: 8 [0/100 (0%)]     Loss1: 1.454922  Loss2: 1.429422          Discrepancy: 0.048694

Test set: Average loss: -0.0070, Accuracy C1: 1251/1860 (67%) Accuracy C2: 1221/1860 (65%) Accuracy Ensemble: 1
254/1860 (67%)
```

```
Test set: Average loss: -0.0504, Accuracy C1: 1764/1860 (94%) Accuracy C2: 1758/1860 (94%) Accuracy Ensemble: 1
762/1860 (94%)

recording %s record/mnist_usps_k_4_alluse_no_onestep_False_0_test.txt
Train Epoch: 190 [0/100 (0%)]   Loss1: 0.022510  Loss2: 0.009406          Discrepancy: 0.011573

Test set: Average loss: -0.0523, Accuracy C1: 1772/1860 (95%) Accuracy C2: 1757/1860 (94%) Accuracy Ensemble: 1
763/1860 (94%)

recording %s record/mnist_usps_k_4_alluse_no_onestep_False_0_test.txt
Train Epoch: 191 [0/100 (0%)]   Loss1: 0.015358  Loss2: 0.005861          Discrepancy: 0.012024

Test set: Average loss: -0.0498, Accuracy C1: 1761/1860 (94%) Accuracy C2: 1753/1860 (94%) Accuracy Ensemble: 1
760/1860 (94%)

recording %s record/mnist_usps_k_4_alluse_no_onestep_False_0_test.txt
Train Epoch: 192 [0/100 (0%)]   Loss1: 0.046938  Loss2: 0.011934          Discrepancy: 0.009163

Test set: Average loss: -0.0497, Accuracy C1: 1756/1860 (94%) Accuracy C2: 1733/1860 (93%) Accuracy Ensemble: 1
746/1860 (93%)

recording %s record/mnist_usps_k_4_alluse_no_onestep_False_0_test.txt
Train Epoch: 193 [0/100 (0%)]   Loss1: 0.012432  Loss2: 0.011339          Discrepancy: 0.014111

Test set: Average loss: -0.0510, Accuracy C1: 1776/1860 (95%) Accuracy C2: 1762/1860 (94%) Accuracy Ensemble: 1
770/1860 (95%)

recording %s record/mnist_usps_k_4_alluse_no_onestep_False_0_test.txt
Train Epoch: 194 [0/100 (0%)]   Loss1: 0.011661  Loss2: 0.008523          Discrepancy: 0.013203

Test set: Average loss: -0.0503, Accuracy C1: 1768/1860 (95%) Accuracy C2: 1758/1860 (94%) Accuracy Ensemble: 1
765/1860 (94%)

recording %s record/mnist_usps_k_4_alluse_no_onestep_False_0_test.txt
Train Epoch: 195 [0/100 (0%)]   Loss1: 0.010646  Loss2: 0.021519          Discrepancy: 0.013082

Test set: Average loss: -0.0501, Accuracy C1: 1768/1860 (95%) Accuracy C2: 1751/1860 (94%) Accuracy Ensemble: 1
761/1860 (94%)

recording %s record/mnist_usps_k_4_alluse_no_onestep_False_0_test.txt
Train Epoch: 196 [0/100 (0%)]   Loss1: 0.018338  Loss2: 0.008945          Discrepancy: 0.010631

Test set: Average loss: -0.0515, Accuracy C1: 1786/1860 (96%) Accuracy C2: 1774/1860 (95%) Accuracy Ensemble: 1
777/1860 (95%)

recording %s record/mnist_usps_k_4_alluse_no_onestep_False_0_test.txt
Train Epoch: 197 [0/100 (0%)]   Loss1: 0.010997  Loss2: 0.012398          Discrepancy: 0.008331

Test set: Average loss: -0.0487, Accuracy C1: 1764/1860 (94%) Accuracy C2: 1744/1860 (93%) Accuracy Ensemble: 1
754/1860 (94%)

recording %s record/mnist_usps_k_4_alluse_no_onestep_False_0_test.txt
Train Epoch: 198 [0/100 (0%)]   Loss1: 0.014923  Loss2: 0.025930          Discrepancy: 0.014242

Test set: Average loss: -0.0509, Accuracy C1: 1771/1860 (95%) Accuracy C2: 1762/1860 (94%) Accuracy Ensemble: 1
773/1860 (95%)

recording %s record/mnist_usps_k_4_alluse_no_onestep_False_0_test.txt
Train Epoch: 199 [0/100 (0%)]   Loss1: 0.005333  Loss2: 0.007897          Discrepancy: 0.008177

Test set: Average loss: -0.0518, Accuracy C1: 1779/1860 (95%) Accuracy C2: 1765/1860 (94%) Accuracy Ensemble: 1
779/1860 (95%)

recording %s record/mnist_usps_k_4_alluse_no_onestep_False_0_test.txt

root at 263b4efd5a52 in /home/huangchenxi/transfer-learning/MCD_UDA
$
```

Compare

ACC	M2U-3	M2U-4	S2M-3	S2M-3-one-step
In my test	94.35%	95.65%	96.4%	67.59%
In paper	93.8 ± 0.8	94.2 ± 0.7	95.9 ± 0.5	

The result **is consistent with** that in paper, which also shows that the MCD_UDA algorithm is indeed more accurate. Using one step, the accuracy is also the expected to be poor.

	In paper	In my test
U->M	91.8 ± 0.9	95.15
U->S		44.47
M->U	93.8 ± 0.8	94.35
M->S		11.27
S->U		80.00
S->M	95.9 ± 0.4	96.40

MADA

Because there only codes in caffe, that it is difficult to recurrent it. So first I read paper again, then I write the network of MADA. After that, I use the architecture that I used before to integrate it.

The network architecture as follows:

```
def __init__(self, n_classes, convnet=None, classifier=None):
        super().__init__()

        self._n_classes = n_classes

        self._convnet = convnet or ConvNet()
        self._classifier = classifier or Classifier(n_classes, 12544)
        self._grl = GRL(factor=-1)
        self._domain_classifiers = [  # k classifiers
            Classifier(1, 12544)
            for _ in range(n_classes)
        ]
```

Compare

The result shows the algorithm is excellent. Because I don't have much time to do the remain work, so I don't compare it.

```
root at 61cebca9a3a1 in /home/huangchenxi/transfer-learning/transfer-learning
[$ python main.py --model='MADA' --source='amazon' --target='dslr' --gpu=1 --num_w]
orkers=4 --num_workers=0 --batch_size=32
MADA
Data size = 2817 , corrects = 3
Initial Train Loss: 0.0000 Acc: 0.0011
Data size = 498 , corrects = 2
Initial Test Loss: 0.0000 Acc: 0.0011

Epoch 0/255
iteration : 0
Data size = 512 , corrects = 480
Alpha =  0.0

Train Loss: 21.7576 Acc: 0.9375
Data size = 498 , corrects = 498
Test Loss: 0.0000 Acc: 1.0000
Current Best Test Acc : 0.937500    Current Best Test Loss : 0.000000  Cur lr : 0
.000300

Epoch 1/255
iteration : 16
Data size = 512 , corrects = 512
Alpha =  0.019528766852031865

Train Loss: 21.4984 Acc: 1.0000
Data size = 498 , corrects = 498
Test Loss: 0.0000 Acc: 1.0000
Current Best Test Acc : 1.000000    Current Best Test Loss : 0.000000  Cur lr : 0
.000292

Epoch 2/255
iteration : 32
Data size = 512 , corrects = 512
Alpha =  0.03904264390418577

Train Loss: 21.4917 Acc: 1.0000
Data size = 498 , corrects = 498
Test Loss: 0.0000 Acc: 1.0000
Current Best Test Acc : 1.000000    Current Best Test Loss : 0.000000  Cur lr : 0
.000284

Epoch 3/255
iteration : 48
Data size = 512 , corrects = 512
Alpha =  0.05852678673317713

Train Loss: 21.4991 Acc: 1.0000
Data size = 498 , corrects = 498
Test Loss: 0.0000 Acc: 1.0000
Current Best Test Acc : 1.000000    Current Best Test Loss : 0.000000  Cur lr : 0
.000276

Epoch 4/255
iteration : 64

Data size = 512 , corrects = 512
Alpha =  0.07796644137536823
```

Of course, this is a bit strange. In the second epoch, the accuracy rate has been 100%. In the paper, the effect of A->D is not so good, although in W->D, it can indeed reach 100%.

Table 1: Accuracy (%) on *Office-31* for unsupervised domain adaptation (AlexNet and ResNet)

Method	A → W	D → W	W → D	A → D	D → A	W → A	Avg
AlexNet (Krizhevsky, Sutskever, and Hinton 2012)	60.6±0.4	95.4±0.2	99.0±0.1	64.2±0.3	45.5±0.5	48.3±0.5	68.8
TCA (Pan et al. 2011)	59.0±0.0	90.2±0.0	88.2±0.0	57.8±0.0	51.6±0.0	47.9±0.0	65.8
GFK (Gong et al. 2012)	58.4±0.0	93.6±0.0	91.0±0.0	58.6±0.0	52.4±0.0	46.1±0.0	66.7
DDC (Tzeng et al. 2014)	61.0±0.5	95.0±0.3	98.5±0.3	64.9±0.4	47.2±0.5	49.4±0.4	69.3
DAN (Long et al. 2015)	68.5±0.3	96.0±0.1	99.0±0.1	66.8±0.2	50.0±0.4	49.8±0.3	71.7
RTN (Long et al. 2016)	73.3±0.2	96.8±0.2	99.6±0.1	71.0±0.2	50.5±0.3	51.0±0.1	73.7
RevGrad (Ganin and Lempitsky 2015)	73.0±0.5	96.4±0.3	99.2±0.3	72.3±0.3	52.4±0.4	50.4±0.5	74.1
MADA	**78.5**±0.2	**99.8**±0.1	**100.0**±.0	**74.1**±0.1	**56.0**±0.2	**54.5**±0.3	**77.1**
ResNet (He et al. 2016)	68.4±0.2	96.7±0.1	99.3±0.1	68.9±0.2	62.5±0.3	60.7±0.3	76.1
TCA (Pan et al. 2011)	74.7±0.0	96.7±0.0	99.6±0.0	76.1±0.0	63.7±0.0	62.9±0.0	79.3
GFK (Gong et al. 2012)	74.8±0.0	95.0±0.0	98.2±0.0	76.5±0.0	65.4±0.0	63.0±0.0	78.8
DDC (Tzeng et al. 2014)	75.8±0.2	95.0±0.2	98.2±0.1	77.5±0.3	67.4±0.4	64.0±0.5	79.7
DAN (Long et al. 2015)	83.8±0.4	96.8±0.2	99.5±0.1	78.4±0.2	66.7±0.3	62.7±0.2	81.3
RTN (Long et al. 2016)	84.5±0.2	96.8±0.1	99.4±0.1	77.5±0.3	66.2±0.2	64.8±0.3	81.6
RevGrad (Ganin and Lempitsky 2015)	82.0±0.4	96.9±0.2	99.1±0.1	79.7±0.4	68.2±0.4	**67.4**±0.5	82.2
MADA	**90.0**±0.1	**97.4**±0.1	**99.6**±0.1	**87.8**±0.2	**70.3**±0.3	66.4±0.3	**85.2**

In digit datasets

	In paper
U->M	96.23
M->U	93.47
S->M	90.83

In Office-31 datasets

(There are some running out.)

	In paper	**In my test**
A->W	90.0 ± 0.2	100
A->D	87.8 ± 0.2	
W->A	66.4 ± 0.3	
W->D	$100.0 \pm .0$	100
D->A	70.3 ± 0.3	
D->W	99.8 ± 0.1	100

Post-preface

Every experiment has been reproduced. Since I first learned about migration and learning, the code may be a little sloppy. If you have some problems, please contact to me.