An Implementation of "Adversarial Discriminative Domain Adaptation"

Ali Jafari, Behrouz Ghamkhar

April 2, 2023

Contents

1 Introduction

While methods based on deep learning have achieved state-of-the-art results on numerous tasks, most of these methods are based on the assumption that the data used to train the model is drawn from the same distribution that the data used to test the model is drawn from. This assumption can be violated in the real world, where AI models should be able to adapt themselves to new environments with little human supervision. In order to address this problem, methods based on *single-source unsupervised domain adaptation* utilize labeled data from a *source domain* to achieve satisfactory performance on unlabeled data from a different but related *target domain* [1].

In this document, we discuss our implementation of "Adversarial Discriminative Domain Adaptation" [2], which is based on PyTorch. In addition to documenting our implementation of the paper, we provide the results of our replications of the performed experiments. Furthermore, we comment on the result of each replicated experiment and offer possible reasons for the success or failure of every experiment.

1.1 Overview of the Method

In [2], the authors propose a framework for single-source unsupervised domain adaptation on a classification task that is trained in three stages. The assumptions of the method are those of single-source unsupervised domain adaptation presented earlier in the introduction, as well as the assumption that the source domain and the target domain share the same set of classes.

Firstly, a *source encoder* and a *classifier* are jointly trained on labeled data from the source domain. Secondly, a *target encoder* is trained adversarially against a *discriminator*. This is performed in a way that given unlabeled data from the target domain, the target encoder can generate representations similar to the representations generated by the source encoder, given data from the source domain. In this sense, the second stage of training in the framework proposed by [2] can be understood like the training of a Generative Adversarial Network [3]. In this analogy, the target encoder is the generator and the 'real data' that the target encoder is trying to forge consists of representations generated by the source encoder from data belonging to the source domain. At the beginning of the second stage of training, the weights of the target encoder are initialized from the weights of the source encoder trained on the source domain in the first stage. During the second stage, the weights of the source encoder are fixed, and only the target en-

coder and the discriminator are trained. Finally, the weights of the target encoder trained in the second stage and the classifier trained in the first stage are fixed. Given unlabeled data from the target domain, the target encoder and the classifier attempt in conjunction to predict class labels for the unlabeled target data [2]. The intuitive justification behind this framework is that if the source encoder and the classifier are trained well on the source domain in the first stage, then given representations generated by the source encoder from data belonging to the source domain, the classifier can accurately predict class labels for data from the source domain. Consequently, if the target encoder can receive data from the target domain and generate similar representations to those that the source encoder can create from source data, then we can give the representations generated by the target encoder to the classifier and expect a reasonably accurate classification performance on the unlabeled target data. Figure 1 illustrates the three stages of the framework.

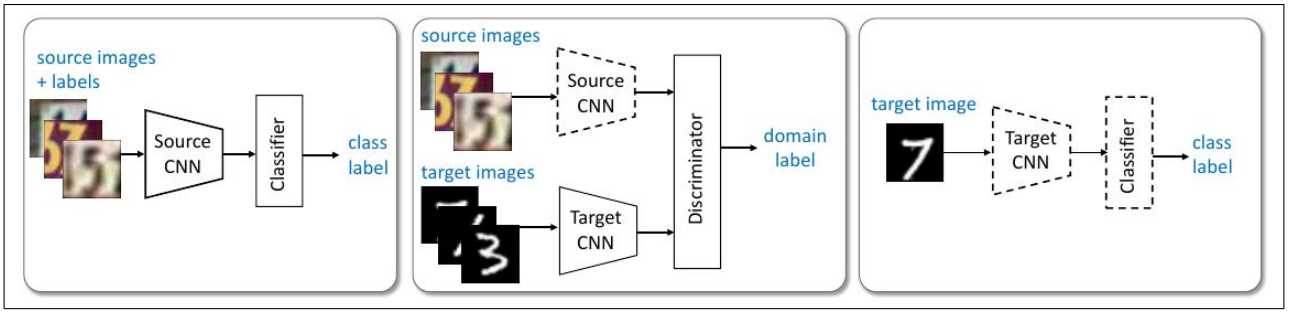

Figure 1: The framework proposed in [2], in which dashed lines around a network indicate that the weights of that network are fixed. This figure belongs to [2].

1.2 Outline of the Document

The rest of this document is organized as follows: Firstly, we explain how our implementation is structured and what the role of each module is. We attempt to follow a 'top-down' approach in explaining the implementation. That is, we begin by discussing the more important parts of the code and then focus on the other modules. Following an explanation of the implementation, we report the results of our replications of the experiments performed in [2]. As mentioned earlier, we aim to offer insights into possible reasons behind the success or failure of each replicated experiment. In addition, we should note that since we have implemented [2] concretely, and since the relevant mathematical content is present in [2] as well, we focus on an intuitive understanding of the learning framework proposed in [2] throughout this document.

2 The Implementation

Before discussing the low-level details of the implementation, we describe the structure of the project and the role of each module.

2.1 Overview of the Implementation

On the highest level, our implementation consists of six modules:

1. `Training`: This module contains three classes, each of which implements a single stage of the framework proposed in [2]. The three classes are called `SourceTrainer`, `Adapter`, and `NetworkTester`. The `SourceTrainer` class implements the first stage of the learning framework, while the `Adapter` and `NetworkTester` classes implement the second and third stages of the learning framework, respectively.

2. `Experiments`: This module contains a single class, called `Experiment`, which uses the three classes provided in the `Training` module to run the experiments described in [2].

3. `Networks`: This module contains implementations for some of the neural networks that are used in the experiments of [2]. It consists of `Encoders`, `Discriminators` and `Classifiers`. We should note that since classifiers of some experiments only consisted of a simple `Linear` layer (as defined in PyTorch), we defined those classifiers directly in the aforementioned experiments.

4. `Extractors`: This module is used to preprocess different datasets and save them in a compressed format, so that entire datasets can be loaded fast and used within the context of PyTorch. The `Extractors` module contains a single concrete class only, called `LabeledDatasetExtractor`. This class works on the assumption that the original dataset is labeled and consists of a number of folders, one for each class, and that each element of the dataset exists as a file in the folder of the class it belongs to. Since all the datasets used in [2] are labeled, the `LabeledDatasetExtractor` class is sufficient for preprocessing all the datasets examined in [2]. However, similar to the experiments performed in [2], the labels belonging to datasets representing target domains are not used in any of the experiments.

5. `Processors`: The `LabeledDatasetExtractor` class relies on a *processor* to transform a single element of a dataset into an intended format. Therefore, it uses a processor on each element of a dataset it extracts. Since all the

datasets used in [2] consist of images, the `Processors` module contains a single concrete class, called `ImageProcessor`. This class normalizes an image into the range $[0, 255]$. The `ImageProcessor` class also allows resizing an image to an intended size, as well as converting an image to grayscale.

6. `Datasets`: This module is used for loading datasets extracted by an extractor. Since all the datasets used in [2] are labeled, the `Datasets` module contains a single class. This class is called `LabeledDataset`, and it follows the conventions of writing datasets in PyTorch.

In addition, the project contains six `.py` files, one for each of the experiments performed in [2]. Each of the aforementioned files (`mnist_usps.py`, `usps_mnist.py`, `svhn_mnist.py`, `modality_adaptation.py`, `amazon_webcam.py`, `dslr_webcam.py`, `webcam_dslr.py`) uses the `Experiment` class to run the experiment indicated in its name. Next we discuss each of the modules described earlier in finer detail.

2.2 The Training Module

As mentioned earlier, the `Training` module consists of three classes. These classes are called `SourceTrainer`, `Adapter`, and `NetworkTester`. Each class implements a single stage of the learning framework proposed in [2]. Here, we describe the details of these classes.

2.2.1 The SourceTrainer Class

This class implements the first stage of the learning framework described in [2]. The constructor of this class receives everything needed to perform the first stage of the aforementioned learning framework. This includes a source encoder, a classifier, a source dataset, a criterion (loss function), an optimizer, the number of iterations used in this stage of the learning framework, the batch size used in this stage of the learning framework, and the device that the source encoder and classifier are trained on (for example, CPU or GPU). Note that since both the source encoder and the classifier should be trained, the optimizer received in the constructor of this class updates the parameters of both the source encoder and the classifier. The `train` method of this class performs the intended supervised learning on the source domain. This method follows the conventional way through which supervised learning is implemented using PyTorch, using a PyTorch `DataLoader` to perform mini-batch gradient descent. In order to follow the procedure described in [2], we always used the categorical cross-entropy loss function as the criterion attribute of this class.

2.2.2 The Adapter Class

This class implements the second stage of the learning framework described in [2]. The constructor of this class receives a source encoder, a target encoder, and a discriminator. In addition, the constructor receives a source dataset, a target dataset, optimizers for the discriminator and the target encoder, the number of iterations used in this stage of the learning framework, the batch size used in this stage of the learning framework, the device used in this stage of the learning framework, and a classifier. We note that based on [2], a classifier is not necessary for this stage. However, we used the classifier trained by the `SourceTrainer` class to monitor the performance of the target encoder on the unlabeled target data during this stage of the learning framework. Another detail to note is that unlike `SourceTrainer`, the constructor of `Adapter` does not receive a loss function. This is because the adversarial training described in [2] always uses a binary cross-entropy loss function.

The `Adapter` class contains three other methods: `adapt`, `train_discriminator`, and `train_target_encoder`. The `adapt` method runs a loop for the number of iterations received in the class constructor. In each iteration of this loop, `train_discriminator` and then `train_target_encoder` are called. This is a high-level implementation of Algorithm 1 described in [3].

The `train_discriminator` method implements one iteration of training the discriminator, as described in Algorithm 1 of [3] and adapted by [2]. In order to do so, first we sample a batch of data from the source dataset and give it to the source encoder. Then, we detach the representations generated by the source encoder to prevent an unwanted parameter update of the source encoder. Afterwards, we give the aforementioned representations to the discriminator as 'real data', and ask the discriminator to predict whether its input data is 'real' (coming from the source encoder) or 'fake' (coming from the target encoder). Next, we sample a batch of data from the target dataset and give it to the target encoder. Then, we detach the representations generated by the target encoder to prevent an unwanted paramter update of the target encoder. Afterwards, we give the aforementioned representations to the discriminator as 'fake data', and ask the discriminator to predict whether its input data is 'real' or 'fake'. Finally, we update the parameters of the discriminator using a binary cross-entropy loss function, based on its prediction error.

The `train_target_encoder` method implements one iteration of training the target encoder in the second stage of the learning framework proposed by [2]. As

mentioned earlier, this is similar to how the generator is trained in Algorithm 1 of [3]. In order to do so, first we sample a batch of data from the target dataset and give it to the target encoder. Since the target encoder has to be updated in this method, we do not detach the representations generated by the target encoder. Next, we give the aforementioned representations to the discriminator as 'real data'. Finally, we update the parameters of the target encoder using a binary cross-entropy loss function, which is based on the prediction error of the discriminator. That is, the target encoder will have a higher loss, if and only if the discriminator has more success in correctly detecting that the representations are generated by the target encoder and not by the source encoder.

2.2.3 The NetworkTester Class

This class implements the third stage of the learning framework proposed in [2]. The constructor of this class receives an encoder, a classifier, a dataset, a batch size, and a device. The `test` method of this class computes the classification accuracy of the received encoder on the received dataset. It counts the total number of elements present in the dataset, as well as those which are classified correctly by the aforementioned encoder. In addition, it counts the number of elements belonging to each class in the dataset, as well as the number of elements in each class that the encoder has correctly classified. Finally, the `test` method computes the classification accuracy of the encoder on the entire dataset, as well as the classification accuracy of the encoder on each class in the dataset. We should note that even though the `NetworkTester` class is primarily used for implementing the third stage of the learning framework illustrated in [2], we also used this class for monitoring other classification accuracies throughout the training process. For example, earlier we mentioned that in the `Adapter` class, we monitored the accuracy of the target encoder on the unlabeled target data. In order to do so, we used the `NetworkTester` class.

2.3 The Experiments Module

As stated earlier, this module consists of a single class, called `Experiment`. We used this class to run the experiments mentioned in [2]. In order to do so, the Experiment class uses the three classes in the Training module.

2.3.1 The Experiment Class

The constructor of this class receives everything needed to initialize objects of `SourceTrainer`, `Adapter` and `NetworkTester` classes, as explained earlier. The

`run` method of this class implements the complete learning framework illustrated in [2]. As mentioned earlier, the learning framework relies on the assumption that the source encoder and the classifier are trained well on the source domain. Therefore, we used `NetworkTester` to measure their performance on the source domain. In order to compare against the *source only* evaluation metric reported in [2], we also used `NetworkTester` to measure the performance of the source encoder and the classifier on the target domain. Note that similar to [2], we initialized the parameters of the target encoder with those of the source encoder after the first stage of training.

2.4 The Networks Module

Earlier, we stated that this module consists of `Encoders`, `Discriminators` and `Classifiers`. Here, we explain our method behind providing an implementation of each.

2.4.1 Encoders

The encoders consist of `LeNetEncoder`, `VGG16Encoder`, `ResNet50Encoder` and `ResNet18Encoder`. There is also a class called `Identity`, which implements an identity function as a PyTorch network. `Identity` is used to separate the encoder of a pretrained PyTorch model from its classifier, since we needed separate encoder and classifier networks to replicate the learning framework of [2]. Similar to [2], we adapted the variant of LeNet implemented in the source code of Caffe [4]. However, we excluded the last linear layer of the network in our adapted implementation and created it directly as the classifier network in the relevant experiments instead.

For `VGG16Encoder`, `ResNet50Encoder` and `ResNet18Encoder`, we used the pretrained networks provided in PyTorch. For the reason mentioned in the previous paragraph, we replaced the classifiers of the pretrained networks with identity functions. We note that even though none of the experiments in [2] use ResNet18, we had to use it because limitations in computational resources prevented us from using VGG16 or ResNet50 in the relevant experiments. Although we still chose to publish their classes. Also, we fixed the parameters in the last convolutional layer of `ResNet18Encoder`, for the reason mentioned in the *office* experiment of [2].

2.4.2 Discriminators

There are three different discriminators described in the experiments of [2]. Consequently, we have `DigitsDiscriminator`, `ModalityAdaptationDiscriminator`

and `OfficeDiscriminator` as our discriminator networks. Each discriminator is named after the experiment in [2] it belongs to, and is implemented as described in [2].

2.4.3 Classifiers

As stated earlier, we separated encoders from classifiers. Classifiers belonging to architectures other than VGG16 were single linear layers that were used directly in experiments as needed. For VGG16, we implemented the `VGG16Classifier` class. This class loads the pretrained VGG16 network available in PyTorch and only keeps the classifier attribute.

2.5 The Extractors Module

As explained earlier, we used this module to preprocess different datasets and save them in a compressed format. This module contains a single concrete class, called `LabeledDatasetExtractor`. However, `LabeledDatasetExtractor` inherits from an abstract class called `Extractor`. We did so to give the code more flexibility and provide the possibility of adding further extractors, in case one needed to extend this project.

2.5.1 The LabeledDatasetExtractor Class

The `LabeledDatasetExtractor` class assumes that the input dataset is labeled and consists of a number of folders, one for each class in the dataset, and that each element of the dataset exists as a file in the folder of the class in the dataset it belongs to. In addition, the `LabeledDatasetExtractor` class assumes that each element of the dataset should ultimately have the same *feature shape*. This class has two methods: `extract` and `save`. The `extract` method of this class traverses the folders belonging to the classes of the dataset in order, and retrieves each element of the dataset. Furthermore, this method uses a *processor* to transform each element of the input dataset into an intended format. Additionally, the `extract` method assigns a numerical class label to the members of each class in the dataset, starting from zero and incrementing by one for each new class. Finally, this method uses the `class_dict` attribute to keep track of the corresponding class name of each numerical class label. The `save` method of `LabeledDatasetExtractor` stores the preprocessed dataset in a location. It creates a `.npz` file to store the elements of the dataset and their numerical labels, and creates a `.pkl` file to store the `class_dict` attribute. We should note that we stored all of our preprocessed datasets in a folder called `StoredDatasets` (as is evident in our `.gitignore` file).

However, due to the larger size of this folder compared to the rest of the project, we decided not to upload it on the GitHub repository of the project. Nevertheless, all the datasets used in [2] can be extracted with the `LabeledDatasetExtractor` class, in case other people desire to test the code. Some of them already existed in a format to be used by this class [5], [6]. We also converted the rest of the datasets to the aforementioned format [7], [8], [9].

2.6 The Processors Module

As stated earlier, the `LabeledDatasetExtractor` class relies on a processor to transform a single element of a dataset into an intended format. This module consists of a single concrete class called `ImageProcessor`. However, `ImageProcessor` inherits from an abstract class called `Processor`. This decision was also with the aim of making the code suitable for possible extensions in the future.

2.6.1 The ImageProcessor Class

References

[1] G. Wilson and D. J. Cook, "A survey of unsupervised deep domain adaptation," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 11, no. 5, pp. 1–46, 2020.

[2] E. Tzeng, J. Hoffman, K. Saenko, and T. Darrell, "Adversarial discriminative domain adaptation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 7167–7176, 2017.

[3] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," 2014.

[4] "https://github.com/bvlc/caffe/blob/master/examples/01-learning-lenet.ipynb."

[5] "https://github.com/teavanist/mnist-jpg."

[6] "https://paperswithcode.com/dataset/office-31."

[7] "https://github.com/alijfri99/usps-jpg."

[8] "https://github.com/alijfri99/svhn-jpg."

[9] "https://github.com/behrouzghamkhar/nyud2-jpg."