DOMAIN ADAPTATION USING UNSUPERVISED LEARNING

ABSTRACT

Collection of high amounts of real world labeled is now possible but the lack of uniformity in the class labels across different domains of data results in poor prediction by even good neural network classifiers in an unseen target domain. Thus, domain adaptation is a very important and widely discussed problem in the field of Computer Vision with high relevance in research today. Generative adversarial Networks have been recently studied for their use in the domain adaptation problem to generate target features that are similar to source features and hence makes classification more accurate. This project uses GANs to improve existing methods by doing two additional things, feature augmentation and and training a feature extractor which is domain agnostic The work done in this project implements an iterative model that uses Conditional GANs to train an encoder in a multi-phase manner which generates features indistinguishable for both source and target domain. The project also conduct different experiments to try to improve the above mentioned model: by firstly training the classifier using the encoder finally obtained and also implementing a recursive approach to train the final encoder.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

CNN	Convolutional Neural Network
GAN	Generative Adversarial Network
CGAN	Conditional Generative Adversarial Network
MNIST	Modified National Institute of Standards and Technology database
SVHN	Street View House Number
USPS	United States Postal Service Dataset
CoGAN	Coupled Generative Adversarial Network
API	Application programming interface
ConvNet	Convolutional Neural Network
SD1	Special Database 1
SD3	Special Database 3
SYN	Synthetic digit dataset
DIFA	Domain Integration and Feature Augmentation

CHAPTER 1

INTRODUCTION

1.1 Motivation

Domain adaptation is an actively researched topic in many areas of Artificial Intelligence including Machine Learning, Natural Language Processing and Computer Vision. While labeled data is available and getting labeled data has been easier over the years, the lack of uniformity of label distributions across different domains results in suboptimal performance of even the most powerful CNN-based algorithms on realistic unseen test data.

The development of powerful learning algorithms such as Convolutional Neural Networks (CNNs) has provided an effective pipeline for solving many classification problems . The abundance of labeled data has resulted in remarkable improvements for tasks such as the Imagenet challenge: beginning with the CNN framework of AlexNet and more recently ResNets and its variants. The similarity across all these approaches is the dependence on huge amounts of labeled data. While the amount of labeled data is increasing across the years to increasing technology, the lack of uniformity of label distributions across different domains results in non-optimal performance of even the most powerful CNN-based algorithms on realistic test data which has not been seen before.While there is a large availability of labeled synthetic data, models trained on such data do not perform well on real data. This fact is of critical importance in scenarios where labeled set of real data in unattainable. Here, an origin (labeled) dataset and a destination(unlabeled) dataset are viewed which are different from each by so called domain shift, which means they are sampled from two distinct distributions. The most useful path among the various methods to domain adaptation is the employment of unlabeled target samples to reduce the difference between the source and target distributions.

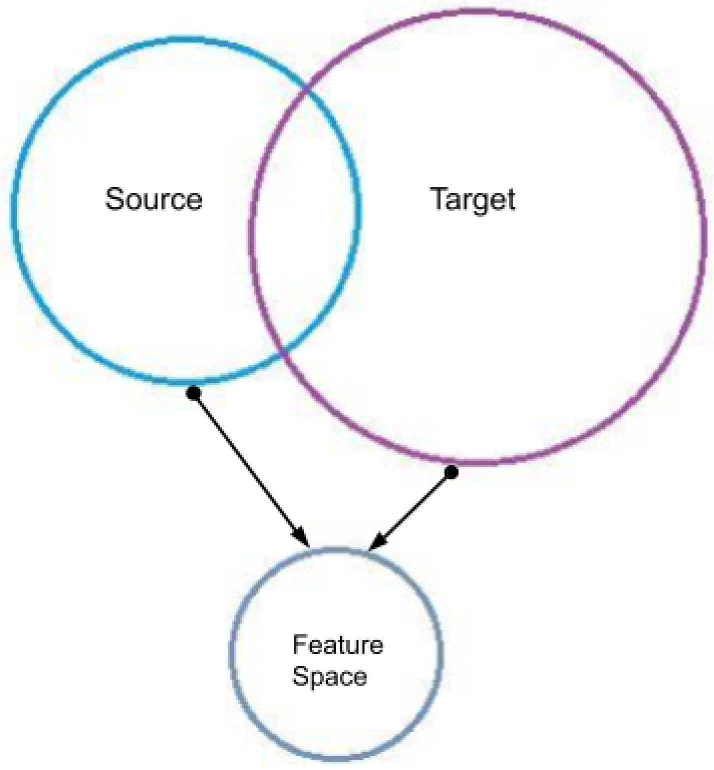


Figure 1.1: Feature extraction from source and target distribution

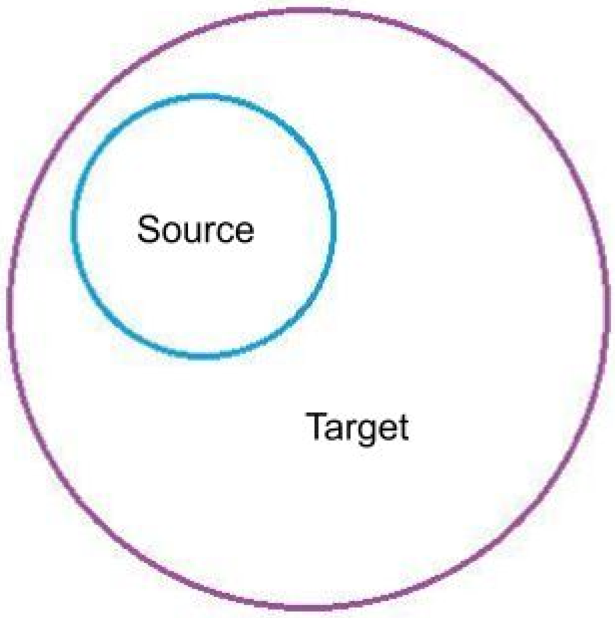


Figure 1.2: Venn diagram for source and target distribution

1.2 Scope

This project focuses on an implementation of unsupervised domain adaptation. Our primary task is to learn an embedding that is resilient to the shift between source and target distribution. We implement an adversarial image generation approach to learn the shared feature embedding from the labeled source data and unlabeled target data. While there have been previous attempts to solve the domain adaptation problem using using an adversarial framework, the uniqueness of the implementation lies in the way the embedding is learnt: a combination of classification loss and an image generation procedure whose basic idea is a Generative Adversarial Network (GANs).

Generative adversarial networks are models capable of mapping noise vectors into realistic samples from a data distribution. Generative Adversarial Networks are models that map noise vectors into realistic samples from a distribution. GANs consist of two networks: the generator and the discriminator. The generator is optimized to produce samples that are as close to the domain of the data distribution using noise. The discriminator is optimized to tell the difference between samples produced by generator and the samples actually taken from the source data distribution. This framework has been shown to produce images with incredible accuracy and has been used for other tasks like generation of a video from a collection of static images and for the mapping of images from one style to another.

The idea is to divide the model in two parts: Feature extractor and classifier. Classifier is only trained and tested on the source dataset while feature extractor is trained and tested on both source and target datasets. While training the model only on source dataset is simpler as the data is labelled and losses are known, training the feature extractor for domain adaptation is a challenging task. The approach requires unsupervised learning as the data is unlabelled. We use generative adversarial networks for learning the target distribution. We use a GAN objective to learn target features that are indistinguishable from the source ones, leading to a pair of feature extractors, one for the source and one for the target samples.

Furthermore, GANs have been exploited in the context of unsupervised domain adaptation. Unsupervised domain adaptation aims at building models that are able to correctly classify target samples, despite the domain shift. In this framework, adversarial training has been used (i) to learn feature extractors that map target samples in a feature space indistinguishable from the one where source samples are mapped, and (ii) to develop image-to-image translation algorithms aimed at converting source images in a style that resembles that of the target image domain.

The datasets we use for training, testing and validation purposes are the MNIST, SVHN and USPS dataset for digit recognition due the vast availability of both labeled and unlabeled data belonging to different domains. This will allow to show the strength of our approach.

1.3 Project Goals

In this project, we try to compare the symbiotic and iterative approaches to train the embeddings for both the source and target datasets. We focus on the iterative approach and try to improve it by training the embedding and the GAN recursively with each other.

CHAPTER 2

LITERATURE SURVEY

2.1 Supervised Domain Adaptation Approaches

In supervised approaches, the model is not trained on the unlabeled data. It is rather trained on datasets with different data distributions.

2.1.1 Ensemble Methods

Ensemble learning is a technique of machine learning in which various models (often referred to as "weak learners") are taught to address the same issue and paired to achieve more accurate outcomes. The primary assumption is that we can get more precise and/or solid models when individually weak classifiers are properly combined.

The idea of ensemble methods is to try reducing bias and/or variance of such weak learners by combining several of them together in order to create a strong learner (or ensemble model) that achieves better performances. A few types of ensemble methods are:

2.1.1.1 Bagging

Bagging stands for Bootstrap Aggregation and is a method where individual learners of the same type are trained separately from each other and then combined via some well-defined averaging method. Bagging is done so that many independent models are averaged so as to obtain an effective model that generates results with lesser variance.

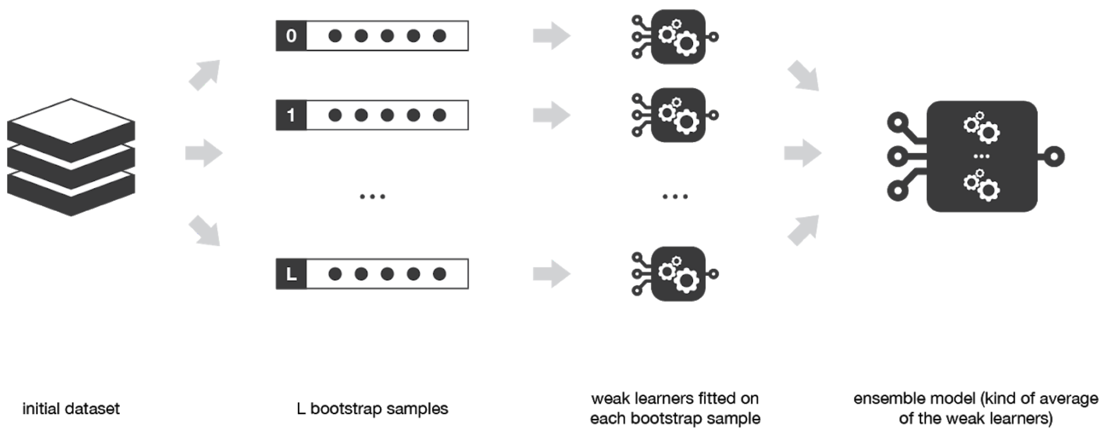

0
1
...
L
...
initial dataset
L bootstrap samples
weak learners fitted on each bootstrap sample
ensemble model (kind of average of the weak learners)

Figure 2.1: Bagging

2.1.1.2 Random Forests

“Forest” is a a collection of many trees wherein each tree comprises a strong learner. The trees that form the forest can be of either two types: deep (no condition on width, but should have many levels) or shallow (not many levels). Shallow trees have lower variance but greater bias whereas deep trees have greater variance but lower bias. This renders shallow trees better for sequential approaches whereas deep trees are more suited to approaches involving bagging whose target is to minimise variance.

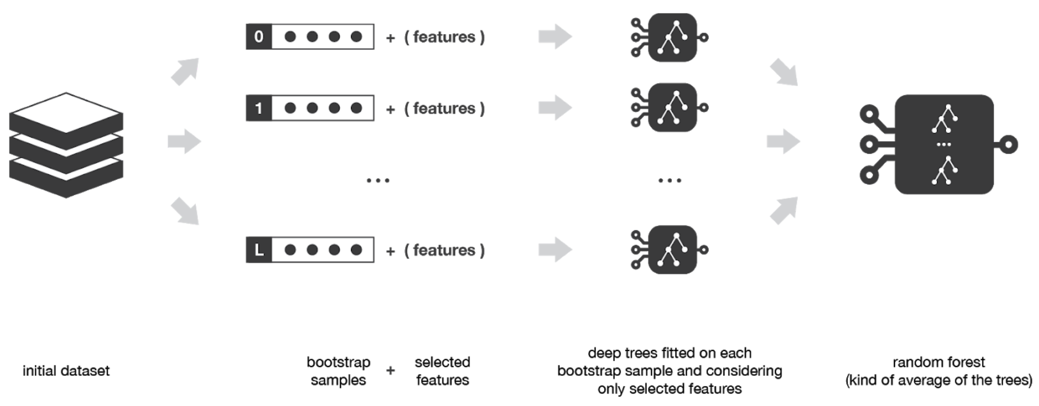

0 + (features)
1 + (features)
...
L + (features)
...
initial dataset
bootstrap samples + selected features
deep trees fitted on each bootstrap sample and considering only selected features
random forest (kind of average of the trees)

Figure 2.2: Random forest

2.1.1.3 Boosting

Boosting is a method where many individual weak learners are combined in a sequential manner. Each of the individual models is trained in a manner that takes into the account the strength and weakness of the classifier trained just before it. It does so by assigning a higher weight to observations that were badly classified by the learner just before it. Boosting then combines all these sequentially learned classifiers in a deterministic manner with the combination being a weighted average based on the power of each individual model.

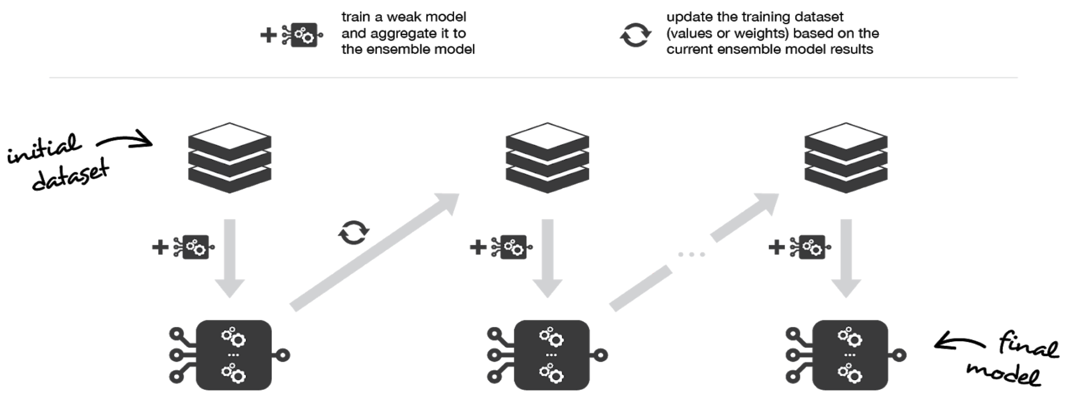

train a weak model and aggregate it to the ensemble model
update the training dataset (values or weights) based on the current ensemble model results
initial dataset
final model

Figure 2.3: Boosting

2.1.1.4 Stacking

The concept of stacking is the training of various weak models and based on the multitude of output results returned by these models, to train a meta-model to predict output by an amalgamation of these weak models.

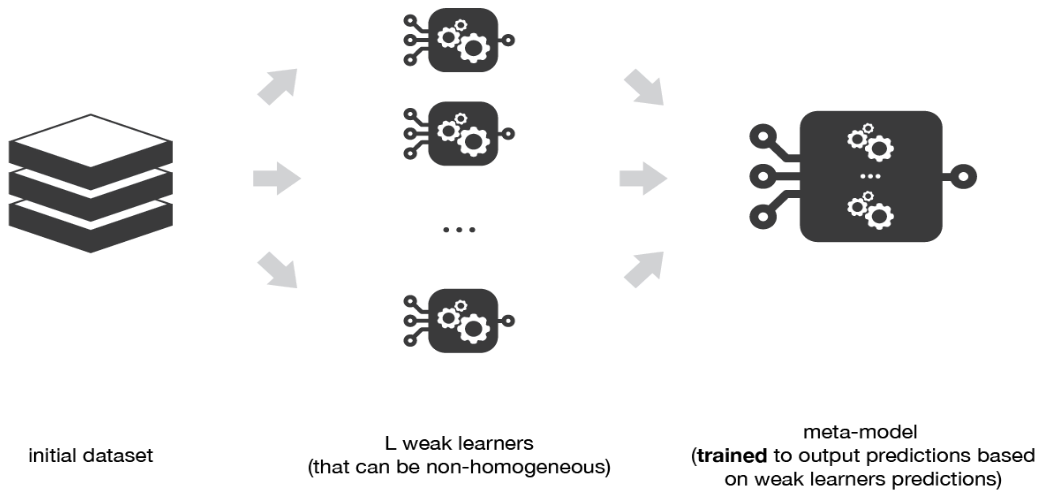

initial dataset
L weak learners
(that can be non-homogeneous)
meta-model
(**trained** to output predictions based on weak learners predictions)

Figure 2.4: Stacking

2.2 Conditional GANs

Generative adversarial networks are models capable of mapping noise vectors into realistic samples from a data distribution. Generative Adversarial Networks are models that map noise vectors into realistic samples from a distribution. GANs consist of two networks: the generator and the discriminator. The generator is optimized to produce samples that are as close to the domain of the data distribution using noise. The discriminator is optimized to tell the difference between samples produced by generator and the samples actually taken from the source data distribution. In specific, classes from which samples are generated can be conditioned to be from particular chosen classes in CGAN.

GANs have lately become popular in the conditional setting. Previous and present works have conditioned GANs on distinct labels, text, and pictures. The image-conditional systems have addressed image estimation from a standard map, prospective frame prediction, item image generation, and sparse annotation picture generation.

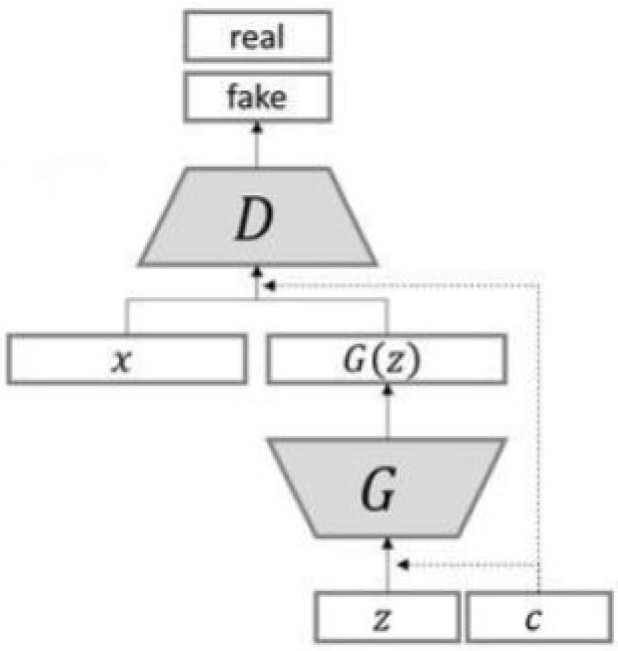


Figure 2.5: Conditional GAN

2.3 Unsupervised Domain Adaptation Approaches

The main limitation of supervised approaches is that it doesn't generalize over a larger domain distribution of unlabeled real data, i.e. target distribution. We need a more generalized approach to make our model robust to this unlabeled data we have. This can only be done using unsupervised learning. Most of the unsupervised approaches focus on making the feature embedding more robust for both source and target distributions. Embedding is involved in both supervised and unsupervised learning steps, while classifier is involved in only supervised learning step.

Ganin and Lempitsky present a neural network (Domain-Adversarial Neural Network, DANN) where a ConvNet-based attribute extractor is optimized to accurately classify source specimens and have domain-invariant characteristics through adversarial learning. Varying works aim to optimize to a minimum value the Maximum Mean Discrepancy between attributes extracted from source and target samples, driving a classifier to classify source samples correctly while minimizing this measure.

Bousmalis et al. suggests learning picture depictions separated into two parts, one distributed across domains and one personal, following the theory that modeling distinctive elements in each domain can aid in obtaining domain-invariant attributes.

Tzeng et al. use GANs to train a target sample encoder by rendering the features obtained with this system indistinguishable from those obtained through a source sample developed encoder. The latter's last part can then be used to infer class tags for both encoders.

Saito et al. present an asymmetric tri-training in which pseudo-labels are inferred and utilized during training for target specimens. Two networks in essence are designed to attach class tags to target specimens and one is trained to acquire target-discriminatory attributes

During training, Haeusser et al. suggest to leverage links between source and target attributes and therein maximizing the domain-invariance of the learned attributes while minimizing the fault on source specimens.

In order to address unsupervised domain adaptation tasks , many image-to-image mapping techniques have been suggested. Taigman et al. suggest the Domain Transfer Network (DTN), which enables pictures to be translated from a source set to a destination set, under a restriction of f-constancy, where f is a generalized function that translates pictures to a feature set The outcome of the translated pictures is depicted in the style of the destination pictures while preserving the information of the pictures supplied in the entry Liu and Tuzel introduce Coupled GAN (CoGAN), a GAN extension that allows the modeling of a joint probability P(X, Y) distribution and the generation of pairs of images coming from P(X) and P(Y) respectively, via some noisy vector. This design can be enforced to image-to-image conversion assignments: setting one image, the noise stream that could most probably have produced that image can be discerned and after feeding to the generating framework, the second image is produced. Bousmalis et al. are suggesting to train an image-to-image conversion network based on both a GAN loss and a task-specific loss (and also a content-specific loss in issues with previous understanding). The resulting system requires both an picture and a sound vector entry, which enables an endless amount of target pictures to be generated. Liu et al.[17] suggest UNIT, a CoGAN adaptation based on both GANs and Variational Auto-Encoders, and draws on the hypothesis of a shared latent space. Methods of image-to-image mapping are implemented to unsupervised domain adjustment by creating destination pictures and training classifiers on them.

Tzeng et al. influenced the domain-invariant attribute extractor we used, with two major distinctions. First, we run the minimax match against characteristics that a pre-trained system generates, thus enhancing the feature space population. Second, we're training the feature extractor.

Domain-invariance also enables both origin and destination sets to use the same feature extractor, while Tzeng et al. need two distinct encoders.

2.3.1 Symbiotic Approach

Swami Sankaranarayanan et al. proposed a symbiotic approach to train the embeddings on both source and target domain to map it in the same feature space. They significantly improved on the unsupervised domain adaptation as most of the previous approaches were based on generating images similar to that in target domain while it trains the network on feature space which is much smaller than the image domain. The latter works better on the datasets which have more complex features such as OFFICE dataset as it is easier to generate features of these images than the images themselves.

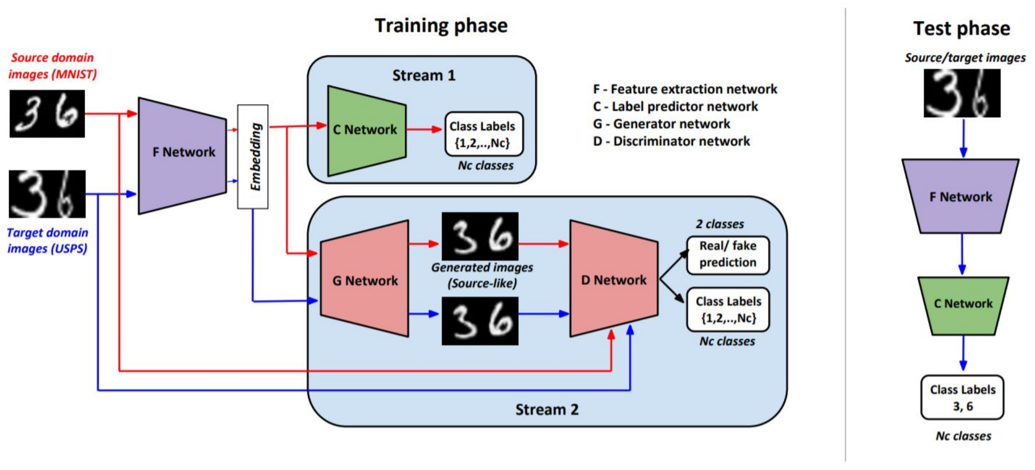


Figure 2.6: Symbiotic approach as proposed by Swami Sankaranarayanan et al.

2.3.2 Iterative Approach

Riccardo Volpi et al. proposed an iterative approach for adversarial feature augmentation for unsupervised domain adaptation. It is a 3 step training procedure to train a shared encoder on both source and target distribution which maps them to same feature space. The approach is faster to train than the symbiotic approach.

2.3.3 Our Work

We implement model proposed by Riccardo Volpi et al. in their paper, “Adversarial Feature Augmentation for Unsupervised Domain Adaptation” in a more modular way to perform our experiments on it.

We also compare the results of the iterative approach and symbiotic approach.

The improvements we propose are as follows:

- Training the classifier after the feature extractor is trained on both source and target distributions, in order to overcome the changes in feature space and features generated.

- Training the feature extractor recursively to make it more robust in both the source and target distributions.

CHAPTER 3

ARCHITECTURE AND ALGORITHM

3.1 Platform

3.1.1 Platform

- We used Google Cloud Virtual Machine instance to train and test our model.
- Training of first experiment took 5 hours to run, second experiment took 5 hours 25 minutes and third experiment with 5 times of repetition took 30 hours to run.

We use **Tensorflow** and **Eager** in Python for implementation of the experiments.

3.1.2 Tensorflow

TensorFlow is a free software library for data flow and differentiable programming across a variety of tasks. It is a symbolic math library and is also used for applications for machine learning such as neural networks.

TensorFlow offers multiple abstraction levels so you can select the right abstraction for your needs. Build and train models using the Keras API, which makes it easy to get started with TensorFlow and learning the machine. If you need more flexibility, immediate iteration and intuitive debugging is possible with eager execution.

3.1.3 Eager

The eager execution of TensorFlow is an imperative programming environment that immediately evaluates operations without creating graphs: operations return concrete values instead of creating a computational graph for later execution. With TensorFlow and debug models, this makes it easy to get started and also reduces boilerplate. To follow this guide, execute the code samples in an interactive python interpreter.

Eager execution is a flexible machine learning platform for research and experimentation, providing:

- *An intuitive interface*—Structure your code naturally and use Python data structures. Quickly iterate on small models and small data.
- *Easier debugging*—Call ops directly to inspect running models and test changes. Use standard Python debugging tools for immediate error reporting.
- *Natural control flow*—Use Python control flow instead of graph control flow, simplifying the specification of dynamic models.

3.2 Model Architecture

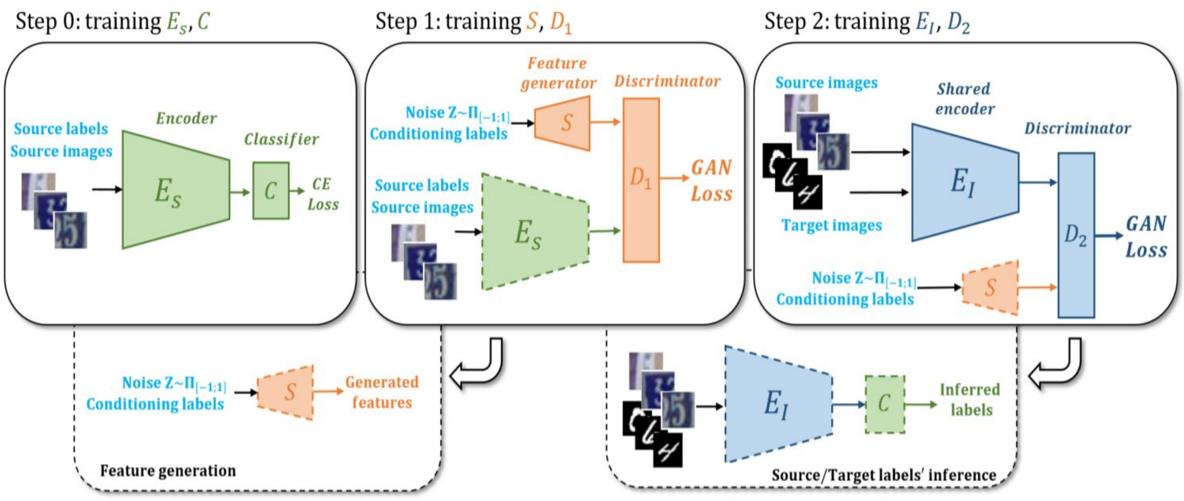


Figure 3.1: Training procedure and Architecture, representing the steps described in Section 3.3

Our objective is to train a domain-invariant feature extractor (E_I), whose learning process is created more powerful by feature space augmentation. The learning method that we intended to fulfill our intention is focused on three distinct stages as shown in Figure 3.1**.** At the beginning , a feature extractor is to be learned from our origin sample data ($C \circ E_S$). This stage is essential because we require a reference attribute space and a well-performing reference classifier on it. Second, we need to train an attribute generator (S) to increase samples in the feature space by augmenting it. We can train it against attributes obtained through E_I by playing a GAN minimax match. Finally, by playing a GAN minimax match against attributes produced by S, we can train a domain independent attribute extractor (E_I). This module can then be coupled with the earlier prepared softmax layer ($C \circ E_I$) to conduct both origin and destination samples inferences. All components are backpropagation trained neural networks.

3.2.1 Encoder and Classifier

As proposed by Riccardo Volpi et al., chosen encoder and classifier for the model is represented in Fig. 3.2. Encoder will output a **128x1** feature set, describing the feature space of the model. While classifier will be simpler structure, which maps this feature set to a **10x1** probability/score vector for each of the labels (0 - 9).

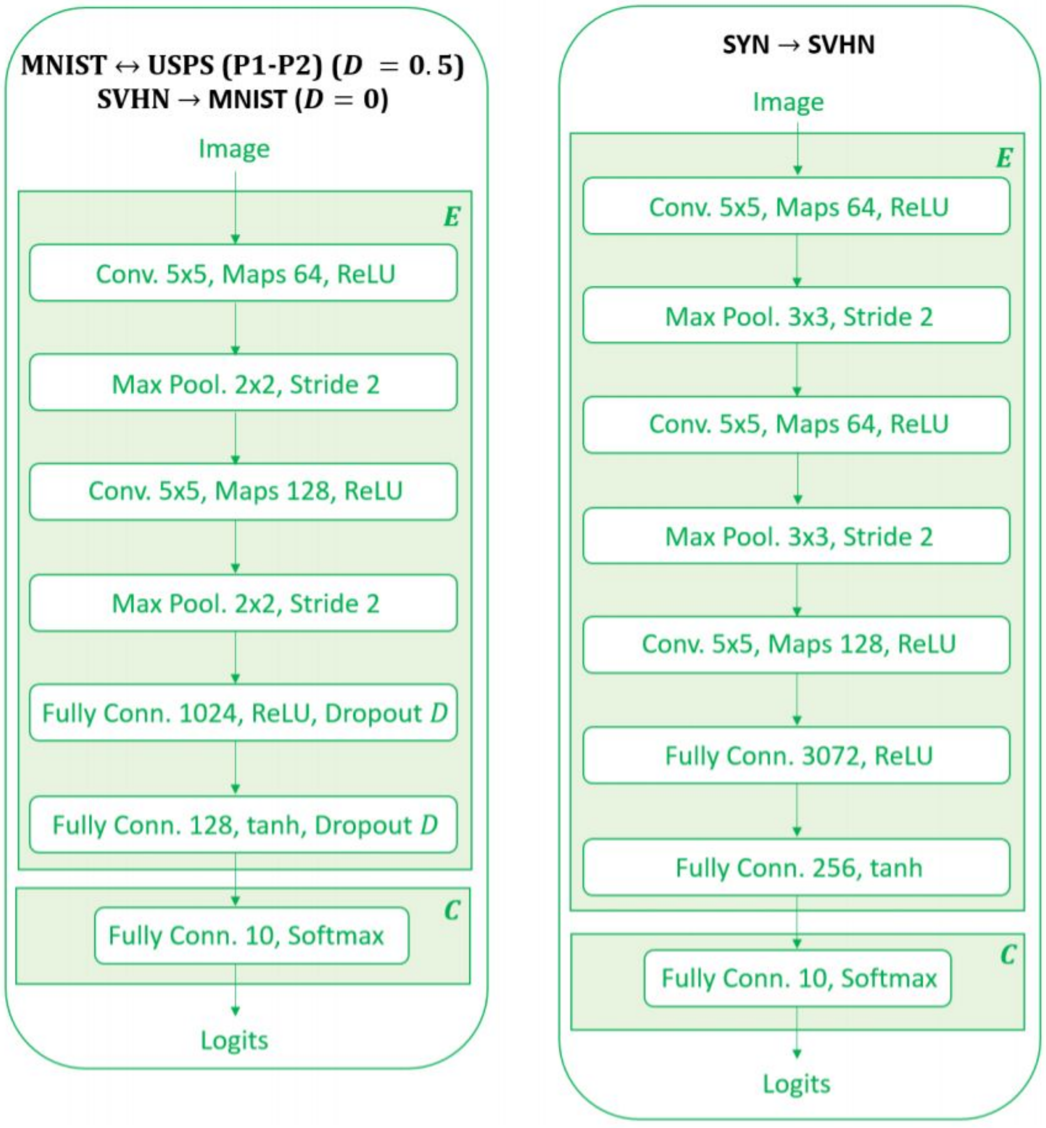


Figure 3.2: Architecture of Encoder and classifier

3.2.2 Generator and Discriminator (D_1)

Generator will take a concatenated string of noise and one hot label to output a feature set for that label. Discriminator will learn classification of real and fake feature sets, outputs of the generator being fake and outputs from the encoder being real. Architecture for S and D_1 is shown in the Fig. 3.3.

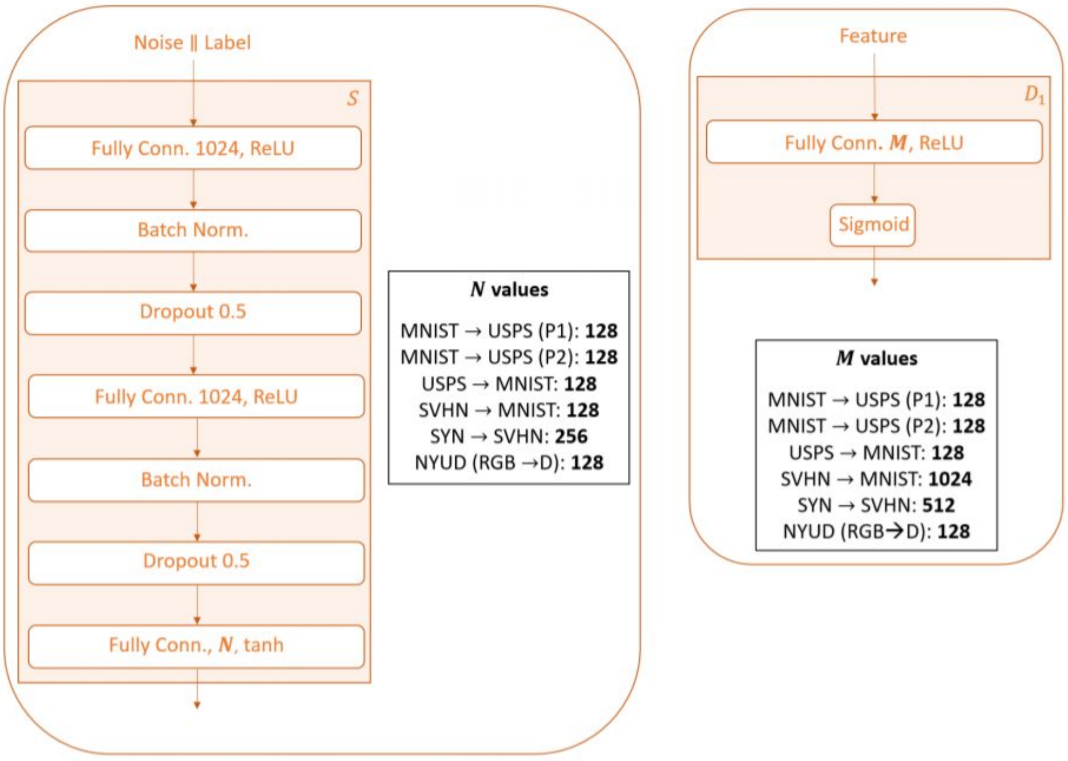


Figure 3.3: Architecture of Generator and Discriminator (D1)

3.2.3 Discriminator DIFA

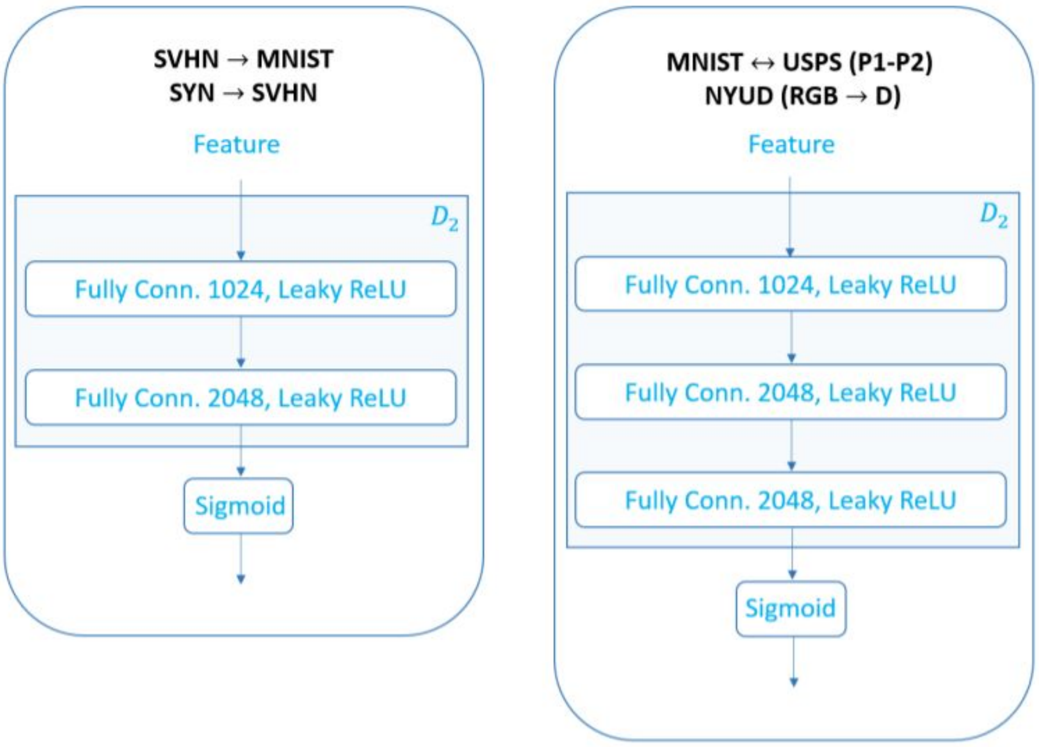

SVHN → MNIST
SYN → SVHN
Feature
D_2
Fully Conn. 1024, Leaky ReLU
Fully Conn. 2048, Leaky ReLU
Sigmoid
MNIST ↔ USPS (P1-P2)
NYUD (RGB → D)
Feature
D_2
Fully Conn. 1024, Leaky ReLU
Fully Conn. 2048, Leaky ReLU
Fully Conn. 2048, Leaky ReLU
Sigmoid

Figure 3.4: Architecture of Discriminator DIFA

3.3 Training Algorithm

3.3.1 Step 0

The model $C \circ E_s$ is trained to classify source samples. Es represents a ConvNet feature extractor and C represents a fully connected softmax layer, with a size that depends on the problem. The optimization problem consists in the minimization of the following cross-entropy loss (CE Loss):

$$\min_{\theta_{E_s}, \theta_C} \ell_0 = \mathbb{E}_{(x_i, y_i) \sim (X_s, Y_s)} H(C \circ E_s(x_i), y_i),$$

(3.1)

where θ_{Es} and θ_C indicate the parameters of E_s and C, respectively, X_s, Y_s are the distributions of source samples (x_i) and source labels (y_i), respectively, and H represents the softmax cross-entropy function.

3.3.2 Step 1

The model S is trained to generate feature samples that resemble the source features. Exploiting the CGAN framework, the following minimax game is defined:

$$\min_{\theta_S} \max_{\theta_{D_1}} \ell_1 = \mathbb{E}_{(z, y_i) \sim (p_z(z), Y_s)} \|D_1(S(z\|y_i)\|y_i) - 1\|^2 + \mathbb{E}_{(x_i, y_i) \sim (X_s, Y_s)} \|\|D_1(E_s(x_i)\|y_i)\|^2, \quad (2)$$

(3.2)

where θ_S and θ_{D1} indicate the parameters of S and D_1, respectively, $p_z(z)$ is the distribution from which noise samples are drawn, and k denotes a concatenation operation. In this and the following steps, we relied on Least Squares GANs since we observed more stability during training.

3.3.2.1 Feature generation

In order to generate an arbitrary number of new feature samples, we only need S, which takes as input the concatenation of a noise vector and a one-hot label code, and outputs a feature vector from the desired class:

$$F(z|y) = S(z||y)$$

(3.3)

where z ~ $p_z(z)$ and F is a feature vector belonging to the class label associated with y (dashed box in Figure 1, left).

3.3.3 Step 2

The domain-invariant encoder E_I is trained via the following minimax game, after being initialized with weights optimized on Step 0 (note that E_S and E_I have the same architecture), a requirement to reach optimal convergence:

$$\min_{\theta_{E_I}} \max_{\theta_{D_2}} \ell_2 = \mathbb{E}_{x_i \sim X_s \cup X_t} \|D_2(E_I(x_i)) - 1\|^2 + \mathbb{E}_{(z,y_i)\sim(p_z(z),Y_s)} \|D_2(S(z||y_i))\|^2,$$

(3.4)

where θ_{EI} and θ_{D2} indicate the parameters of E_I and D_2, respectively. Since the model EI is trained using both source and target domains, the feature extractor results domain-invariant. In particular, it maps both source and target samples in a common feature space, where features are indistinguishable from the ones generated through S. Being the latter trained to produce features indistinguishable from the source ones, the feature extractor E_I can be combined with the classification layer of Step 0 (C) and used for inference (as in Tzeng et al.):

$$\tilde{y}_i = C \circ E_I(x_i),$$

(3.5)

where x_i is a generic image from the source or the target data distribution and $\tilde{Y}_i$ is the inferred label .

3.3.4 Step 3 (Experiment)

The domain-invariant encoder E_I is used to train the classifier according to the newly learnt encoder in the feature space. Classifier is initialized with the weights of classifier trained in step 0. The optimization problem consists in the minimization of the following cross-entropy loss (CE Loss):

$$\min_{\theta_{E_s}, \theta_C} \ell_0 = \mathbb{E}_{(x_i, y_i) \sim (X_s, Y_s)} H(C \circ E_s(x_i), y_i),$$

(3.6)

Here the minimization depends only on the component C. E_I is treated as a constant, i. e. it is not trained. Training is done only on source dataset.

3.3.5 Step 4 (Experiment)

Repeat from the step 1. This time the encoder involved is the encoder trained in the step 2. The generator is supposed to learn a better mapping not biased to only source distribution. Then repeat the step 2. Now that the generator has learned a better mapping, the encoder will also learn the better mapping making it more robust. This step can be done recursively as it is supposed to make encoder more robust. However, if the target distribution does not contain a large number of samples, this may lead to overfitting of the encoder.

CHAPTER 4

IMPLEMENTATION

4.1 Dataset

The following datasets were used for the source and target distributions in our project:

4.1.1 MNIST

MNIST dataset is a very common dataset used by both amateurs and experts to fine-tune their abilities in computer vision and digital image processing.

- The MNIST dataset is a subgroup of the Special Database 3 (SD3) and Special Database 1 (SD1) datasets of the NIST comprising handwritten digits' binary pictures.
- The MNIST dataset contains a total of 60,000 data samples, each of which is 30 thousand from the SD1 and SD3 datasets. The initial NIST dataset specimens were size-normalized into a cluster of twenty-by-twenty images while preserving the aspect ratio.
- These pictures are focused on a block of 28x28 pixels around their centers of mass.

- Since half of these images are entries from college pupils while the other half are from the authorities of the U.S. Censor Bureau, there is a wide range of types in this data set, which makes this data set appropriate for learning a system of identification of characters.

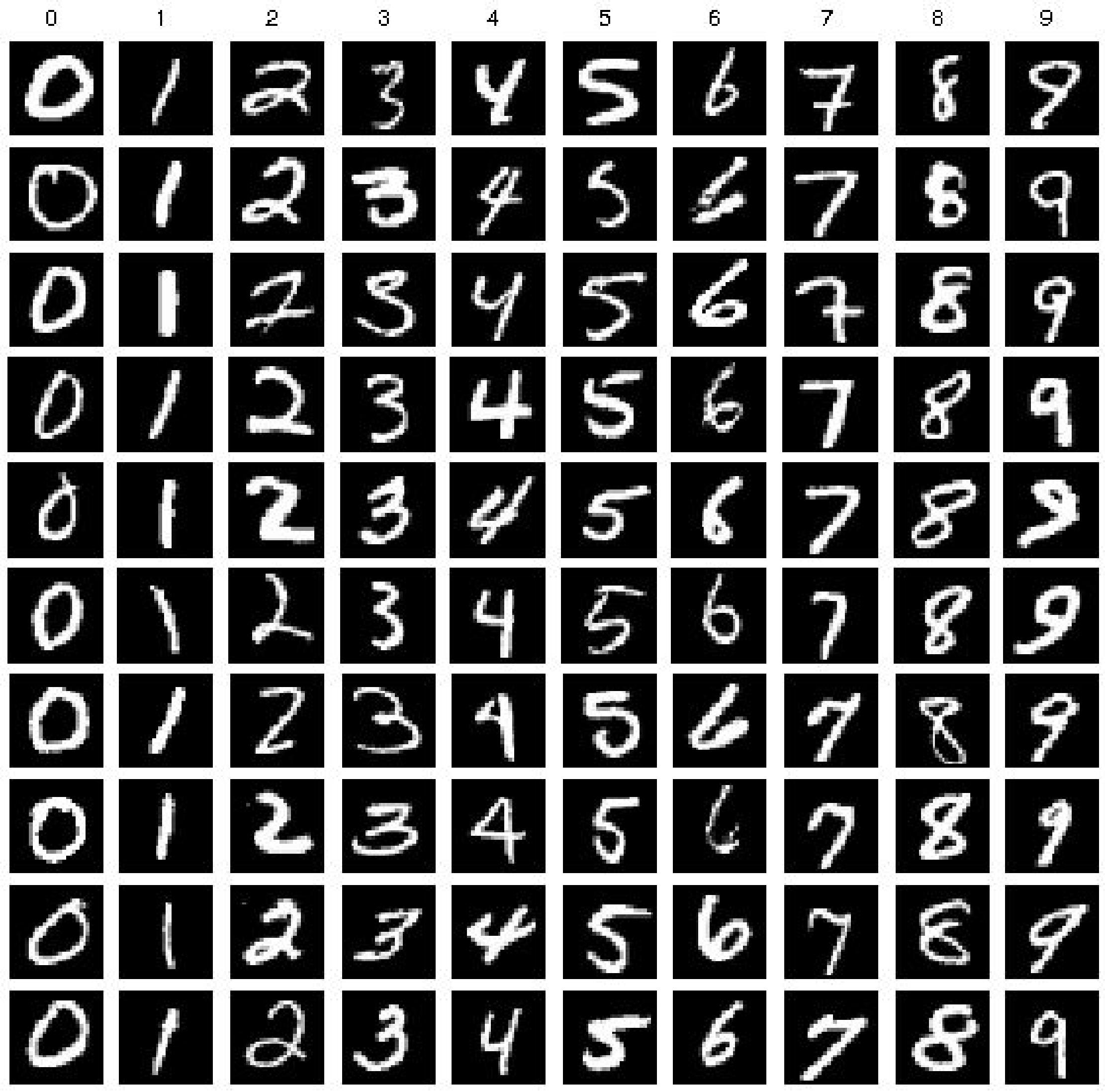


Figure 4.1: Samples from MNIST Dataset

4.1.2 USPS

One of the standard datasets for handwritten digital recognition is the USPS digit database. It includes digital pictures of about 5,000 city names, 5,000 state names, 10,000 ZIP codes, and 50,000 alphanumeric letters. Each picture was screened on a high-quality flatbed digitizer from postal in a operating post office at 300 pixels / in in 8-bit black scale. The data for the author, style, and training process were unconstrained. These features assist to resolve the constraints of previous databases that contained only isolated characters or were ready under specified conditions in a laboratory environment. In addition, the database is split into specific instruction and testing sets to promote investigators ' exchange of outcomes and comparisons of performance.

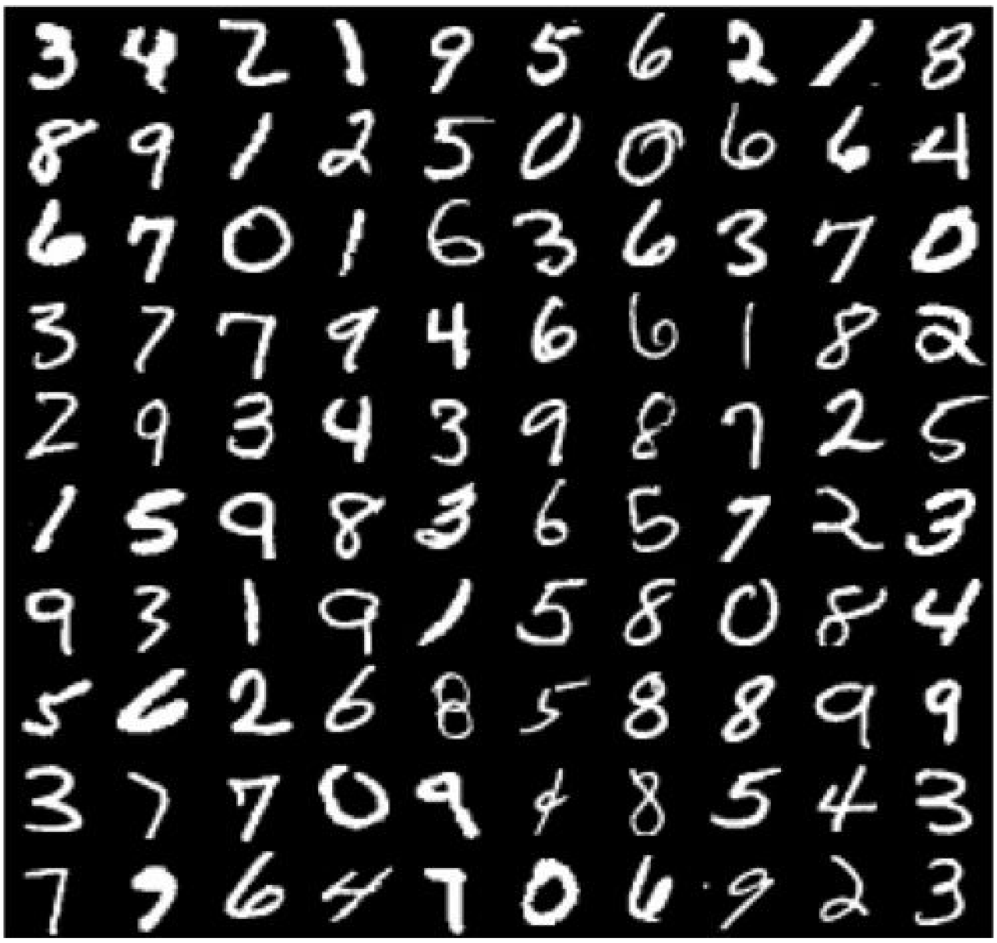


Figure 4.2: Samples from USPS Dataset

4.1.3. SVHN

- Through its vast and creative potential applications in the real world, object recognition and image processing has become one of the hottest topics in machine learning.
- Machine training algorithms can be very helpful in processing visual data, such as evaluating the reliability of NYC Bike Lanes through road imagery.
- Within this sector, one of the most famous is the Street View House Numbers (SVHN) dataset.
- SVHN is a real-world picture dataset designed to develop machine learning and object identification algorithms with minimal input preprocessing and formatting requirements.
- It can be seen as comparable in taste to MNIST (e.g., the pictures are of tiny cropped digits), but includes more marked data (over 600,000 digit pictures) in order of size and arises from a much difficult, unsolved, real-world issue (acknowledging digits and figures in natural scene pictures).
- Via Google Street View pictures, SVHN is generated from the house numbers.
- It was used in Google-created neural networks to display and link home figures to their geolocations.
- This is a excellent benchmark dataset to perform with, discover and train designs that correctly define road figures and integrate them into all kinds of initiatives. This dataset includes three.zip documents containing over 600k real-world pictures of Google Street View home figures. The number sequence in the pictures is limited in duration.
- This collection of information includes three.zip documents containing more than 600k pictures of real-world home numbers captured from Google Street View. The number sequence in the pictures is limited in length.
    - test.zip: 26,032 digits for testing
    - train.zip: 73,257 digits for training
    - extra.zip: 531,131 additional specimens, somewhat less challenging to be used as supplementary teaching information

- Each of the digits have corresponding labels, i.e., digit 1 has label 1 and so on. Thus, there are 10 classes for digits 0-9.
- The pictures are the real color house-number pictures with bounding boxes of character type in.png format.
- For each respective.zip file, digitStruct.mat includes bounding box data that can be loaded using Matlab.The digitStruct.mat documents comprise a struct of the same size as the amount of real pictures called digitStruct.
- The following fields are available for each object in digitStruct:
  - name: filename of the respective image, a string
  - bbox: struct array that contains the position, size and label of each digit bounding box in the image. Ex. digitStruct(300).bbox(2).height gives height of the 2nd digit bounding box in the 300th image.

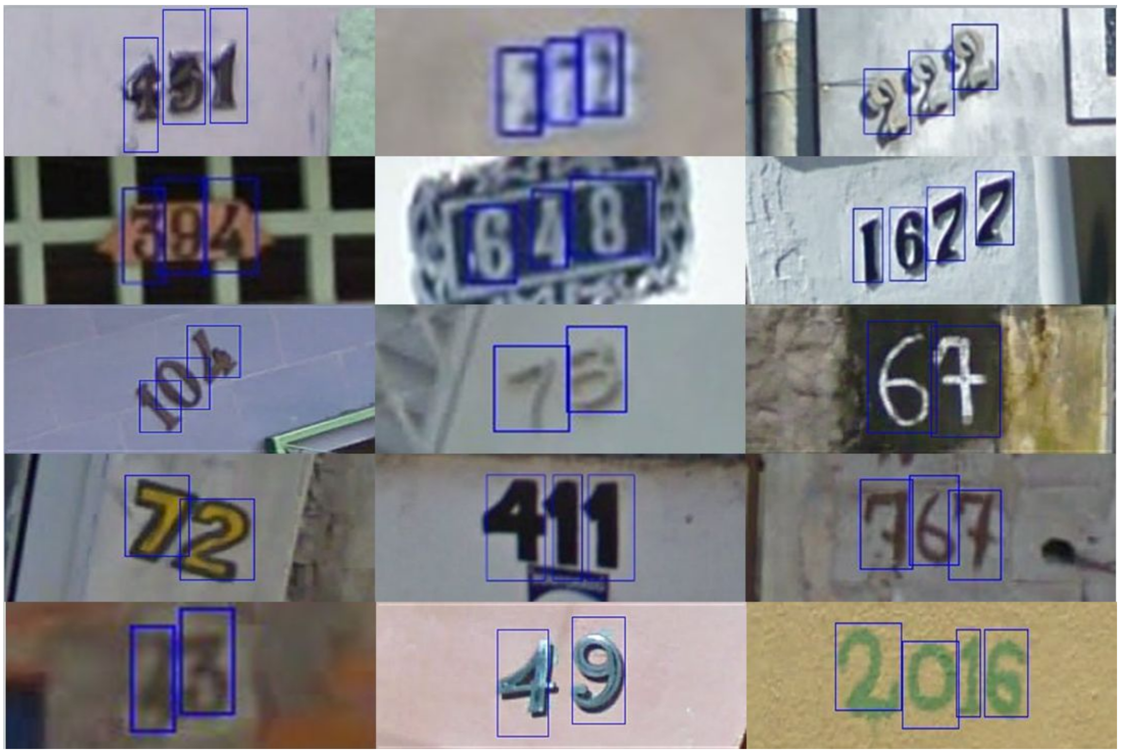


Figure 4.3: Samples from SVHN Dataset

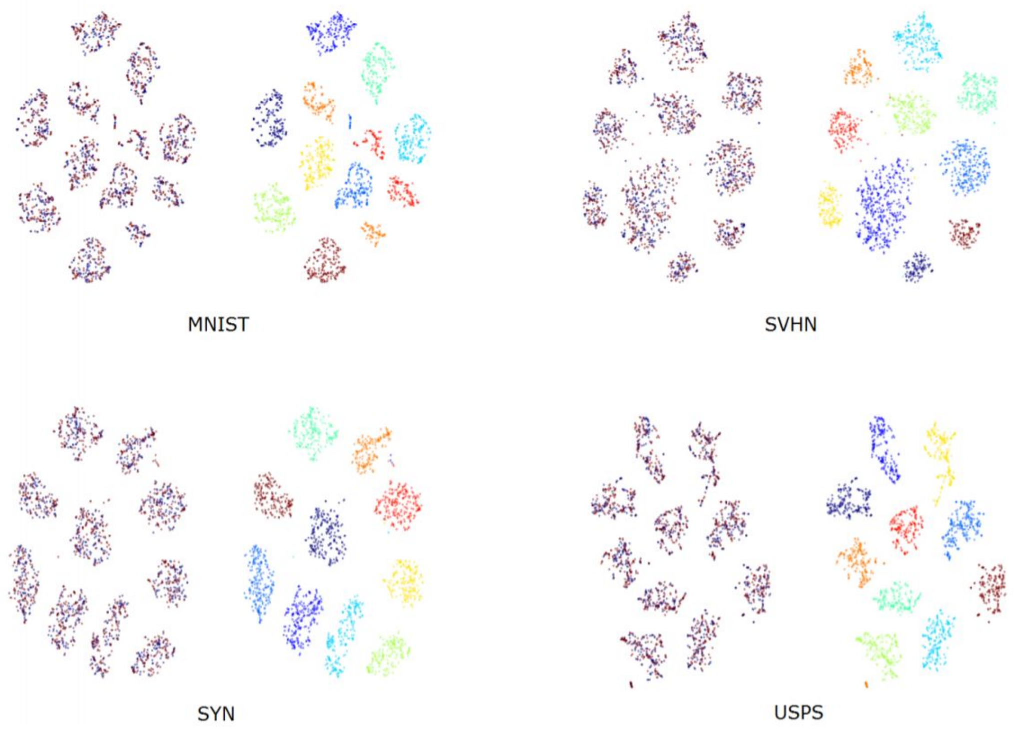


Figure 4.4: t-SNE plots of features associated with different adopted datasets (MNIST, SVHN, SYN, USPS)

We used a number of benchmark divisions of open source / target data sets taken in domain alignment to assess our method.

4.1.4 MNIST ↔ USPS

Both datasets comprise of a strong black background with white numbers. Two distinct procedures have been investigated: the first (P1) is to sample 2,000 MNIST pictures and 1,800 USPS pictures. The second (P2) comprises of using the entire MNIST training collection, 50,000 pictures, and splitting USPS into 6,562 training pictures, 2,007 testing pictures, and 729 verification pictures. We checked the split's two routes (MNIST to USPS and MNIST to USPS) for P1. For P2, we only evaluated MNIST to USPS, and in this situation we also prevented using the verification set. We resized USPS numbers to 28 by 28 pixels in both experimental procedures, which is the size of the MNIST pictures.

4.1.5 SVHN → MNIST

SVHN is constructed with true Street View House numbers pictures. Following the standard protocol for unsupervised domain adaptation, we used the entire training sets of both datasets and tested them on the MNIST test set. (SVHN training set contains 73,257 images) For this task, SVHN was converted to grayscale and MNIST images altered to 32 by 32 pixels. The extra set of SVHN was unused.

4.1.6 SYN DIGITS → SVHN

This divide reflects an instance of synthetic-to-real domain adjustment, of major significance to computer vision exploration as creating labeled synthetic information quite often involves less effort than acquiring massive labeled information set of real instances. SYN DIGITS comprises 500,000 pictures from the same categories of SVHN. SVHN test set was used for evaluating the accuracy.

4.2 Case Study

4.2.1 Experiments

We perform a few experiments in order to verify, compare and improve the existing unsupervised domain adaptation approaches.

- Implement, train, validate and test the iterative model for unsupervised domain adaptation as proposed by Riccardo Volpi et al.
- Train the classifier after the domain invariant encoder is trained in step 2. This is supposed to ensure that classifier adapts to the changes in mapping of the encoder in feature space.
- Perform the steps 1 and 2 recursively, i. e. 1, 2, 1, 2, 1, 2… to see if there is improvement in learning of the encoder in both source and target domain. It is followed by training of classifier with the new domain invariant encoder as in previous experiment.

4.2.2 Evaluation

Each experiment was evaluated on the basis of its accuracy to correctly classify samples in both source and target distributions. Accuracy is calculated as follows:

Accuracy = (Correct / Total) X 100 %

We also evaluate by comparison with other models. Comparison can be done by using accuracy as metric, the higher the better.

4.2.3 Implementation

We implemented the aforementioned model using **Tensorflow** and **Eager** in Python.

The code for the all the experiments is available at https://colab.research.google.com/drive/1SmaOjBKC-hjoezPB7mf53HpmuSvjlI4r. Some code snippets are also shown in Appendix 1.

CHAPTER 5

RESULTS & ANALYSIS

5.1 Model Evaluation

While training the model with SVHN as the source dataset and MNIST as the target dataset, the base encoder-classifier pair gave an accuracy of **99.2%** on the SVHN **68.2%** on the MNIST dataset. After training the domain-invariant encoder (Experiment 1), the accuracy significantly improved to **89.7%** on target dataset. The experiment 2 further slightly improved accuracy by **0.4%** and Experiment 3 improved accuracy by **1.3%**. Detailed accuracy and comparison is shown in the table.

However the Experiment 2 showed no improvements in the other two source and target pairs, the Experiment 3 still shows improvement of **0.766±0.009%**.

Feature Generator in step 1 was also trained and tested on SVHN, MNIST and USPS to verify how well the generator generates the features when trained on each of the datasets. Detailed evaluation of Generator S is shown in the table.

Dataset	**Accuracy**
SVHN	0.974
USPS	0.998
MNIST	0.995

Table 5.1: Accuracies for Feature Generator with different datasets

5.2 Comparison

We compare the different models with our experiments as suggested in the paper "Adversarial Feature Augmentation using Unsupervised Domain Adaptation". We compare these models with our own experiments performed. The comparison is displayed in Table 5.2.

	SVHN→MNIST	MNIST→USPS (P1)	MNIST→USPS (P2)	USPS→MNIST
Source	0.682	0.723	0.797	0.627
DANN	0.739	0.771 ± 0.018	-	0.730 ± 0.020
DDC	0.681 ± 0.003	0.791 ± 0.005	-	0.665 ± 0.033
DSN	0.827	-	-	-
ADDA	0.760 ± 0.018	0.894 ± 0.002	-	0.901 ± 0.008
Tri	0.862	-	-	-
DTN	0.844	-	-	-
PixelDA	-	-	0.959	-
UNIT	0.905	-	0.960	-
CoGANs	no conv.	0.912 ± 0.008	0.957	0.891 ± 0.008
Ours (Expt. 1)	0.897 ± 0.028	0.914	0.959	0.910
Ours (Expt. 2)	0.901 ± 0.026	0.912	0.960	0.911
Ours (Expt. 3)	**0.910 ± 0.020**	**0.920**	**0.961**	**0.912**
Target	0.992	0.999	0.999	0.975

Table 5.2: Comparison between different models on different pairs of source, target distributions

5.3 Visual Analysis

We graphically present the analysis of our experiments using accuracy as the metric for evaluation (the higher, the better).

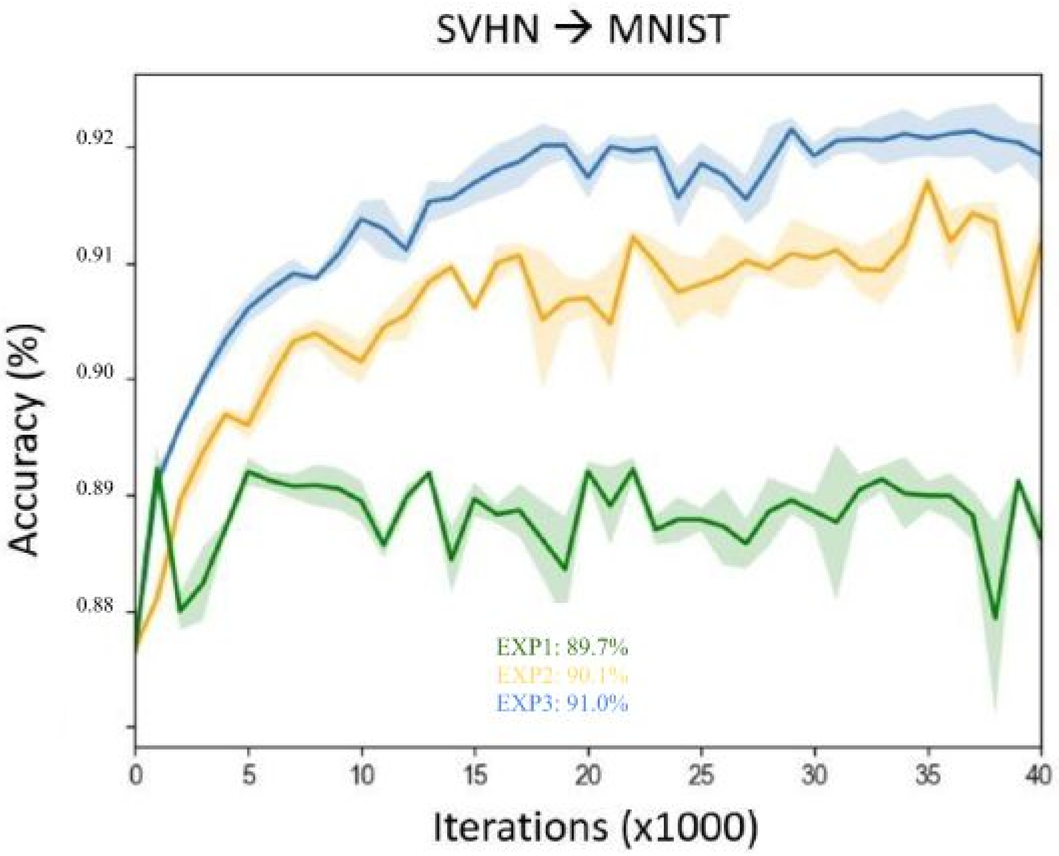


Figure 5.1: Iterations v/s Accuracy graph for Experiments 1, 2 and 3 on SVHN→MNIST

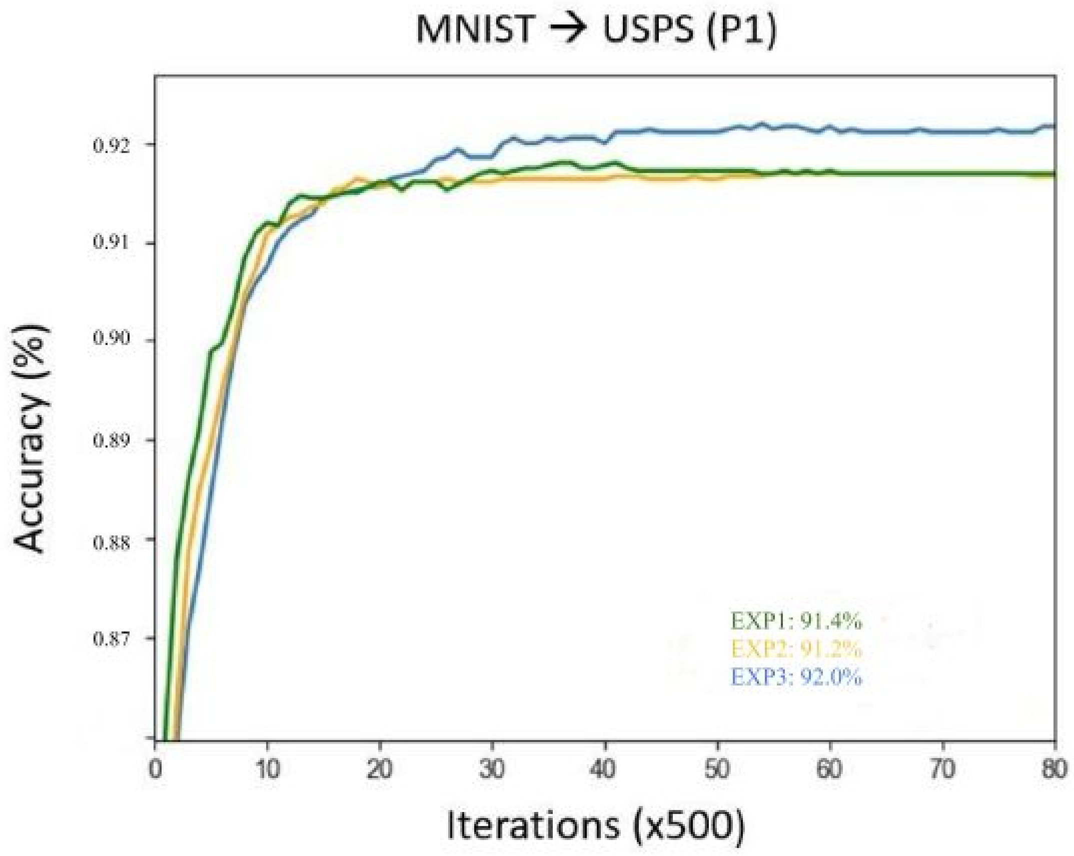


Figure 5.2: Iterations v/s Accuracy graph for Experiments 1, 2 and 3 on MNIST→USPS (P1)

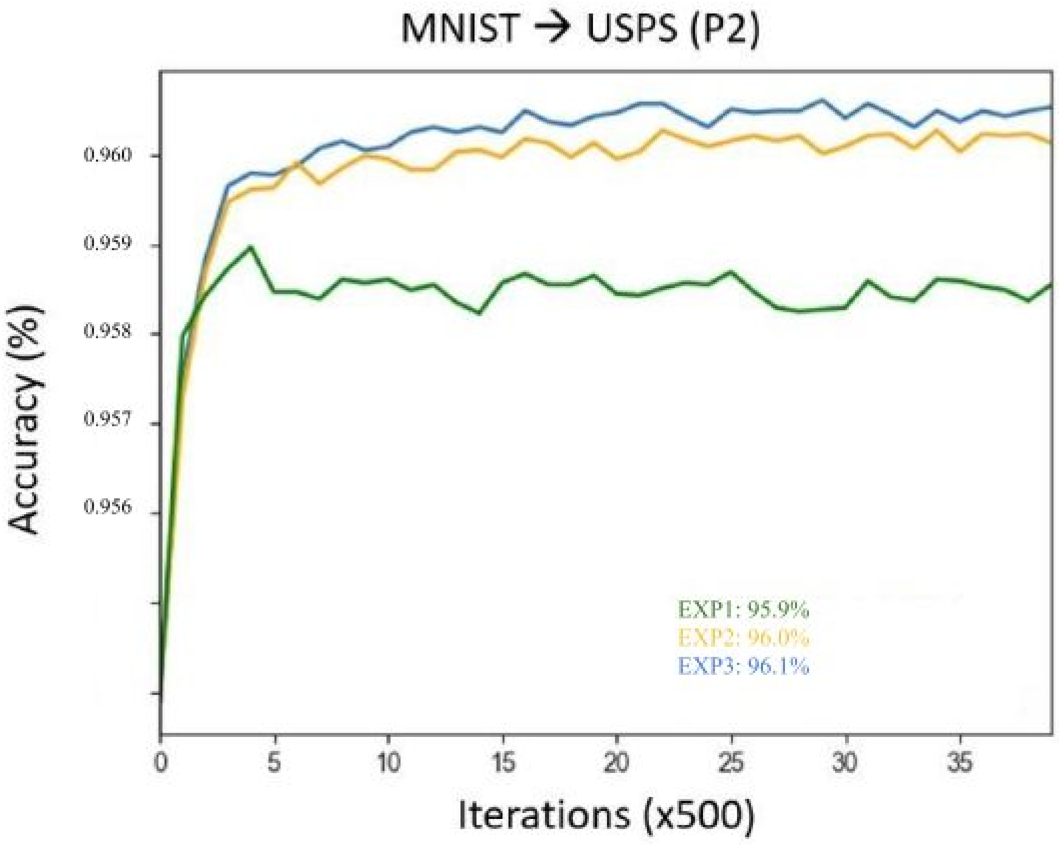


Figure 5.3: Iterations v/s Accuracy graph for Experiments 1, 2 and 3 on MNIST→USPS (P2)

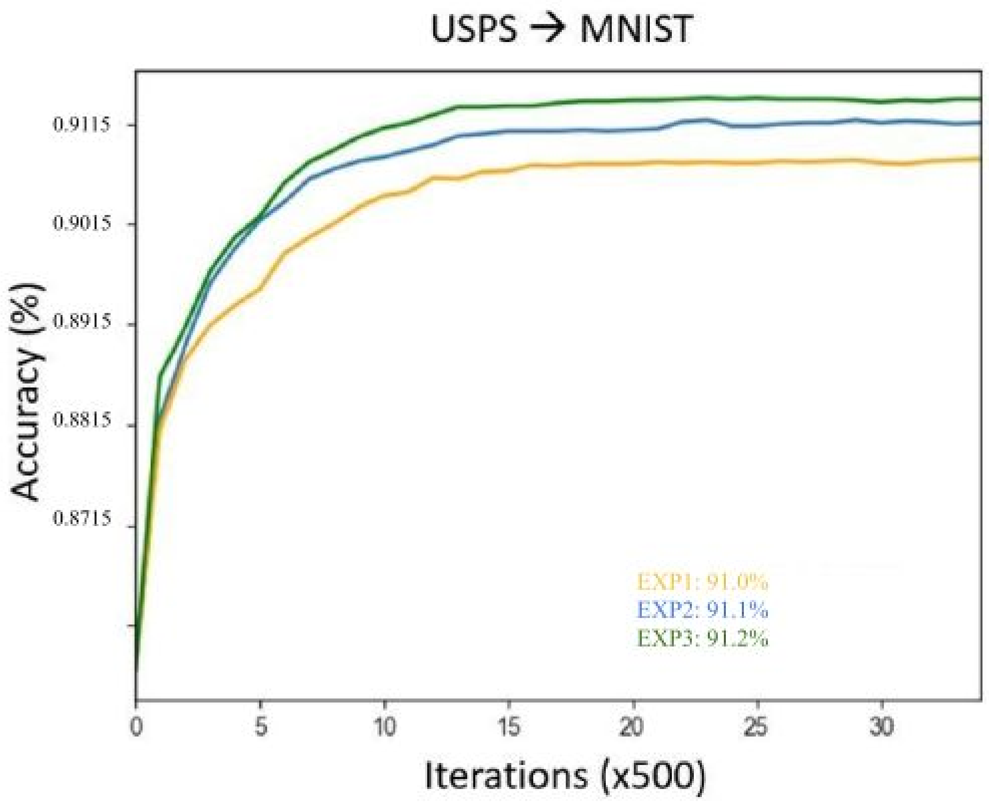


Figure 5.4: Iterations v/s Accuracy graph for Experiments 1, 2 and 3 on USPS→MNIST

5.4 Inferences

- The Experiment 2 does not have a major effect on the accuracy of the model. This is justified as we keep the classifier model much simpler than the encoder model (classifier only contains one fully connected layer). So training it won't improve the accuracy by a significant amount.
- The Experiment 3 have a significant improvement in the accuracy of the model. It was a successful addition in the existing model. However, the number of recursions depends on a number of factors. One factor is amount of computing power and time available for training the classifier. Another factor could be overfitting of the model in case the target dataset is not as large.

CHAPTER 6

CONCLUSION

6.1 Conclusion

In this work, we proposed techniques to improve the current usage of GAN objectives in the unsupervised domain adaptation framework. First, we induced domain invariance recursively through an extension of the data augmentation in the feature space using GANs. Second, we proposed to train the classifier with the domain invariant feature encoder. A comprehensive evaluation of standard domain adaptation benchmarks was carried out and the results confirmed that both approaches lead to higher accuracy of target data. We have also shown that the feature extractors obtained can also be used on source data. Results indicated that our approach is comparable or superior to current state-of - the-art methods, except for a single benchmark. In particular, we have done better than recent, more complex methods that rely on target images to address unsupervised domain adaptation tasks. This achievement reopened the debate about the need to generate images that belong to the target distribution: recent results seemed to suggest this.

6.2 Future Scope

The problem of domain adaptation is still in research stage and there is no single solution for each of the machine learning applications. For example, the symbiotic and iterative approaches doesn’t improve much from the baseline on problems like digit classification. However the complex problems such as object recognition on OFFICE dataset or other unstructured datasets, the unsupervised domain adaptation approaches that train feature mappings work significantly better than the baseline classifier. We plan to test our approach to more complex unsupervised domain adaptation issues for future work, as well as investigate whether feature augmentation can be applied to different frameworks, e.g. contexts where traditional data augmentation has proven successful.

REFERENCES

[1] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[2] Riccardo Volpi, Pietro Morerio et al.; 2018; Adversarial Feature Augmentation for Unsupervised Domain Adaptation

[3] Sankaranarayanan, S , Balaji, Y, Castillo, C, Chellappa, R, 2018, Generate to Adapt: Aligning Domains Using Generative Adversarial Networks. ; The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2018, pp. 8503-8512.

[4] Saito, K, Watanabe,K, Ushiku, Y, Harada T; 2018; Maximum Classifier Discrepancy for Unsupervised Domain Adaptation; The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2018, pp. 3723-3732.

[5] Bottou et al. ; 1994; Comparison of classifier methods: a case study in handwritten digit recognition; IEEE; Print ISBN: 0-8186-6270-0.

[6] K. Bousmalis, N. Silberman, D. Dohan, D. Erhan, and D. Krishnan. Unsupervised pixel-level domain adaptation with generative adversarial networks. In The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), July 2017.

[7] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In CVPR09, 2009.

[8] P. Haeusser, T. Frerix, A. Mordvintsev, and D. Cremers. Associative

domain adaptation. In International Conference on Computer Vision (ICCV), 2017.

[9] M.-Y. Liu and O. Tuzel. Coupled generative adversarial networks. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, Advances in Neural Information Processing Systems 29, pages 469–477. Curran Associates, Inc., 2016.

[10] M. Long, Y. Cao, J. Wang, and M. I. Jordan. Learning transferable

features with deep adaptation networks. In Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37, ICML’15, pages 97–105, 2015.

[11] M. Mirza and S. Osindero. Conditional generative adversarial nets. CoRR, abs/1411.1784, 2014.

[12] A. Radford, L. Metz, and S. Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. In Proceedings of the International Conference on Learning Representations (ICLR), 2016.

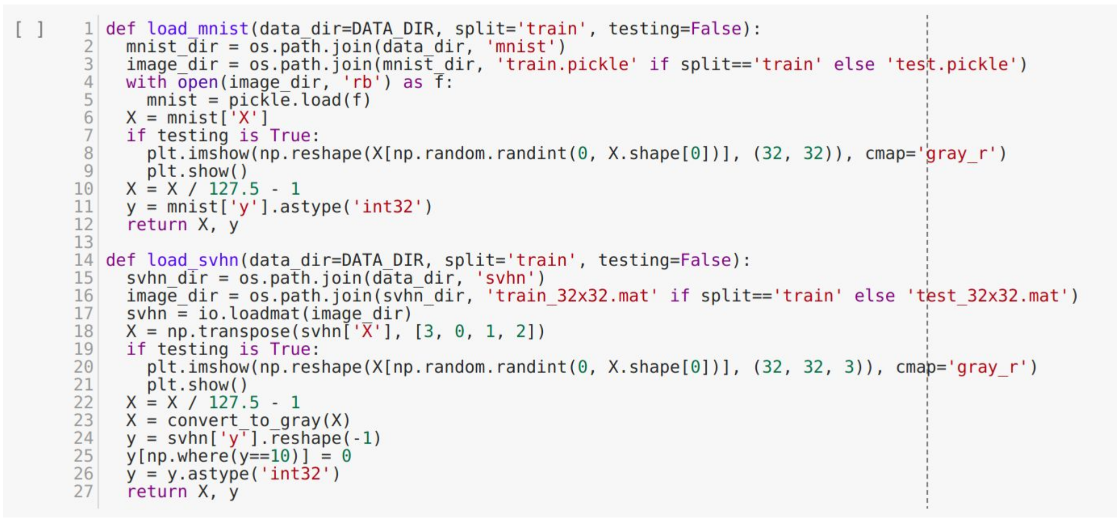
[13] E. Tzeng, J. Hoffman, K. Saenko, and T. Darrell. Adversarial discriminative domain adaptation. In The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), July 2017.

[14] E. Tzeng, J. Hoffman, N. Zhang, K. Saenko, and T. Darrell. Deep domain confusion: Maximizing for domain invariance. CoRR, abs/1412.3474, 2014.

APPENDIX 1: CODE SNIPPETS

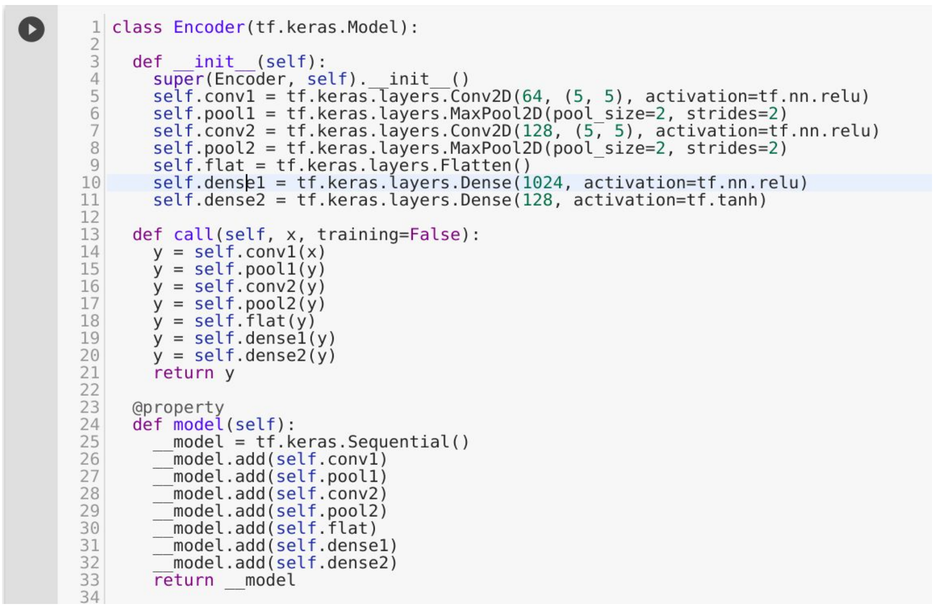
Loading datasets

Loading MNIST and SVHN with same shape (None x 32 x 32 x 1)



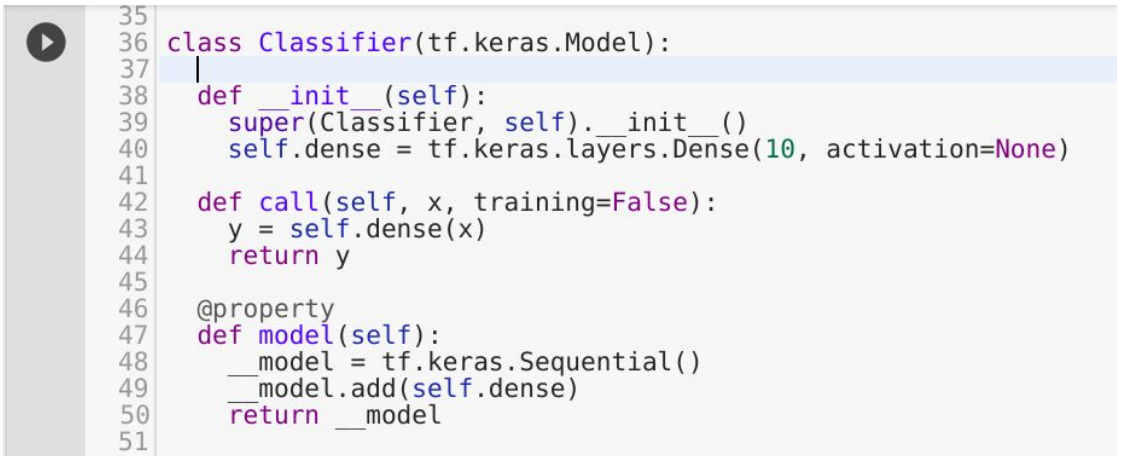
```
[ ]
def load_mnist(data_dir=DATA_DIR, split='train', testing=False):
  mnist_dir = os.path.join(data_dir, 'mnist')
  image_dir = os.path.join(mnist_dir, 'train.pickle' if split=='train' else 'test.pickle')
  with open(image_dir, 'rb') as f:
    mnist = pickle.load(f)
  X = mnist['X']
  if testing is True:
    plt.imshow(np.reshape(X[np.random.randint(0, X.shape[0])], (32, 32)), cmap='gray_r')
    plt.show()
  X = X / 127.5 - 1
  y = mnist['y'].astype('int32')
  return X, y

def load_svhn(data_dir=DATA_DIR, split='train', testing=False):
  svhn_dir = os.path.join(data_dir, 'svhn')
  image_dir = os.path.join(svhn_dir, 'train_32x32.mat' if split=='train' else 'test_32x32.mat')
  svhn = io.loadmat(image_dir)
  X = np.transpose(svhn['X'], [3, 0, 1, 2])
  if testing is True:
    plt.imshow(np.reshape(X[np.random.randint(0, X.shape[0])], (32, 32, 3)), cmap='gray_r')
    plt.show()
  X = X / 127.5 - 1
  X = convert_to_gray(X)
  y = svhn['y'].reshape(-1)
  y[np.where(y==10)] = 0
  y = y.astype('int32')
  return X, y
```

Model for Encoder



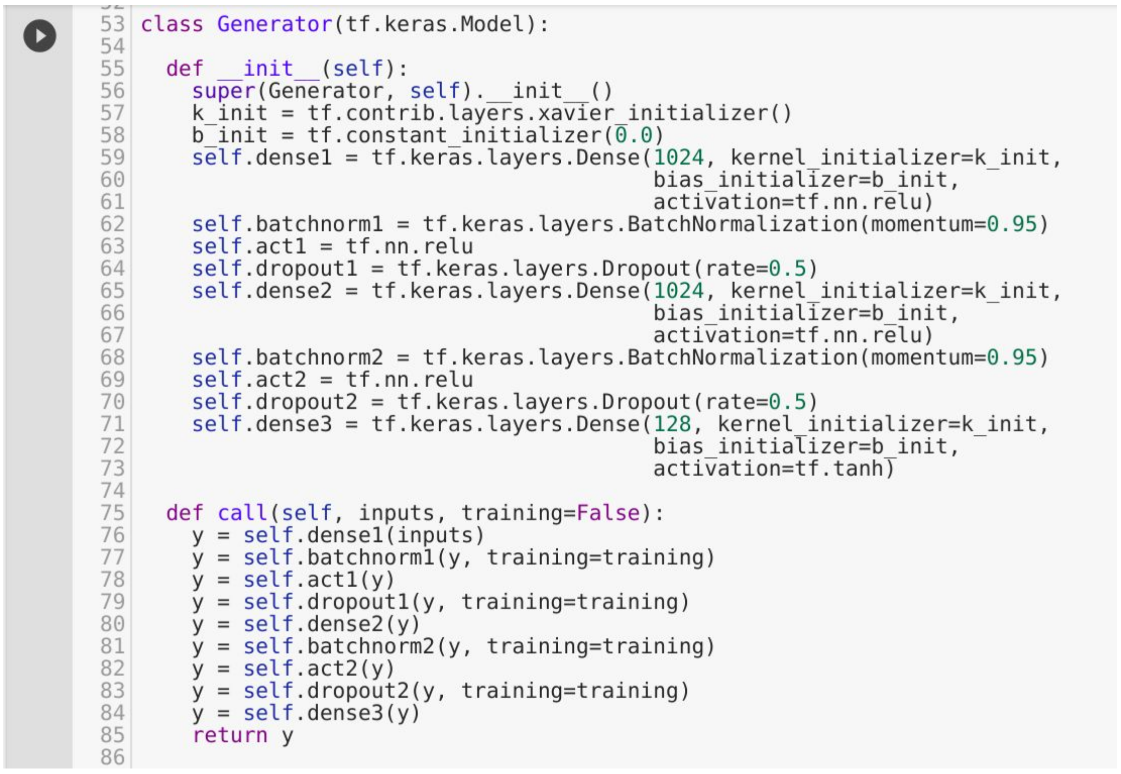
```
class Encoder(tf.keras.Model):

  def __init__(self):
    super(Encoder, self).__init__()
    self.conv1 = tf.keras.layers.Conv2D(64, (5, 5), activation=tf.nn.relu)
    self.pool1 = tf.keras.layers.MaxPool2D(pool_size=2, strides=2)
    self.conv2 = tf.keras.layers.Conv2D(128, (5, 5), activation=tf.nn.relu)
    self.pool2 = tf.keras.layers.MaxPool2D(pool_size=2, strides=2)
    self.flat = tf.keras.layers.Flatten()
    self.dense1 = tf.keras.layers.Dense(1024, activation=tf.nn.relu)
    self.dense2 = tf.keras.layers.Dense(128, activation=tf.tanh)

  def call(self, x, training=False):
    y = self.conv1(x)
    y = self.pool1(y)
    y = self.conv2(y)
    y = self.pool2(y)
    y = self.flat(y)
    y = self.dense1(y)
    y = self.dense2(y)
    return y

  @property
  def model(self):
    __model = tf.keras.Sequential()
    __model.add(self.conv1)
    __model.add(self.pool1)
    __model.add(self.conv2)
    __model.add(self.pool2)
    __model.add(self.flat)
    __model.add(self.dense1)
    __model.add(self.dense2)
    return __model

```

Model for classifier



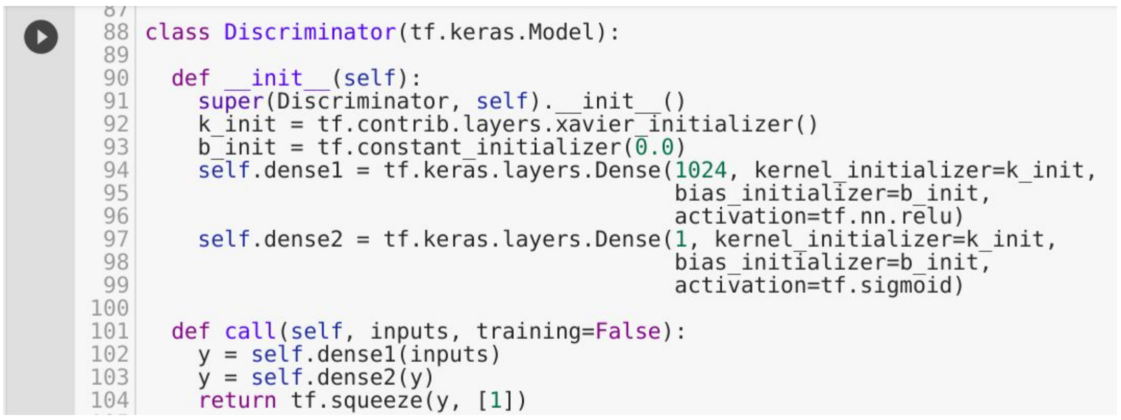
```python
class Classifier(tf.keras.Model):

  def __init__(self):
    super(Classifier, self).__init__()
    self.dense = tf.keras.layers.Dense(10, activation=None)

  def call(self, x, training=False):
    y = self.dense(x)
    return y

  @property
  def model(self):
    __model = tf.keras.Sequential()
    __model.add(self.dense)
    return __model

```

Model for Generator



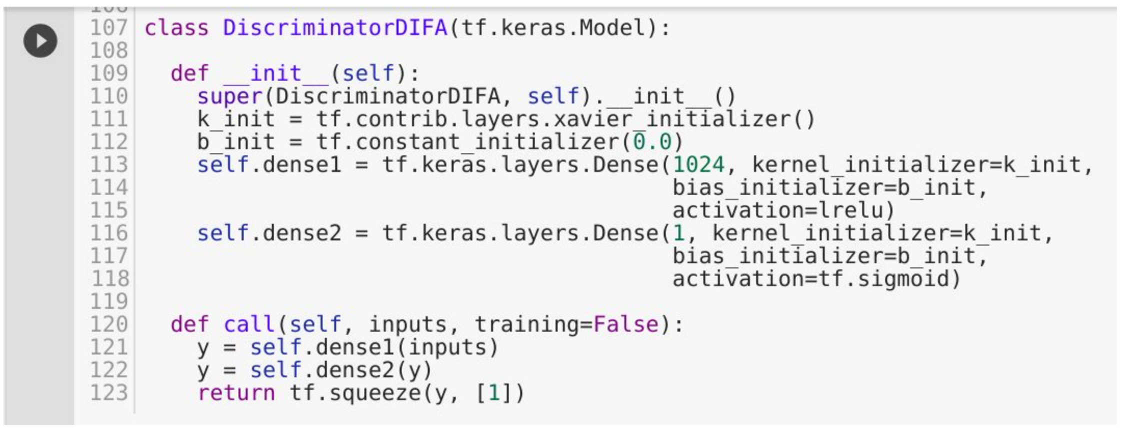
```python
class Generator(tf.keras.Model):

  def __init__(self):
    super(Generator, self).__init__()
    k_init = tf.contrib.layers.xavier_initializer()
    b_init = tf.constant_initializer(0.0)
    self.dense1 = tf.keras.layers.Dense(1024, kernel_initializer=k_init,
                                        bias_initializer=b_init,
                                        activation=tf.nn.relu)
    self.batchnorm1 = tf.keras.layers.BatchNormalization(momentum=0.95)
    self.act1 = tf.nn.relu
    self.dropout1 = tf.keras.layers.Dropout(rate=0.5)
    self.dense2 = tf.keras.layers.Dense(1024, kernel_initializer=k_init,
                                        bias_initializer=b_init,
                                        activation=tf.nn.relu)
    self.batchnorm2 = tf.keras.layers.BatchNormalization(momentum=0.95)
    self.act2 = tf.nn.relu
    self.dropout2 = tf.keras.layers.Dropout(rate=0.5)
    self.dense3 = tf.keras.layers.Dense(128, kernel_initializer=k_init,
                                        bias_initializer=b_init,
                                        activation=tf.tanh)

  def call(self, inputs, training=False):
    y = self.dense1(inputs)
    y = self.batchnorm1(y, training=training)
    y = self.act1(y)
    y = self.dropout1(y, training=training)
    y = self.dense2(y)
    y = self.batchnorm2(y, training=training)
    y = self.act2(y)
    y = self.dropout2(y, training=training)
    y = self.dense3(y)
    return y

```

Model for Discriminator



```python
class Discriminator(tf.keras.Model):

  def __init__(self):
    super(Discriminator, self).__init__()
    k_init = tf.contrib.layers.xavier_initializer()
    b_init = tf.constant_initializer(0.0)
    self.dense1 = tf.keras.layers.Dense(1024, kernel_initializer=k_init,
                                        bias_initializer=b_init,
                                        activation=tf.nn.relu)
    self.dense2 = tf.keras.layers.Dense(1, kernel_initializer=k_init,
                                        bias_initializer=b_init,
                                        activation=tf.sigmoid)

  def call(self, inputs, training=False):
    y = self.dense1(inputs)
    y = self.dense2(y)
    return tf.squeeze(y, [1])
```

Model for DIFA Discriminator



```python
class DiscriminatorDIFA(tf.keras.Model):

  def __init__(self):
    super(DiscriminatorDIFA, self).__init__()
    k_init = tf.contrib.layers.xavier_initializer()
    b_init = tf.constant_initializer(0.0)
    self.dense1 = tf.keras.layers.Dense(1024, kernel_initializer=k_init,
                                        bias_initializer=b_init,
                                        activation=lrelu)
    self.dense2 = tf.keras.layers.Dense(1, kernel_initializer=k_init,
                                        bias_initializer=b_init,
                                        activation=tf.sigmoid)

  def call(self, inputs, training=False):
    y = self.dense1(inputs)
    y = self.dense2(y)
    return tf.squeeze(y, [1])
```

Training operations for step 0

```python
class S0TrainOps:

  def __init__(self,
               learning_rate=0.0003,
               batch_size=64,
               checkpoint_basedir=CHECKPOINT_DIR,
               training_stats_dir=TRAINING_STATS_DIR,
               data_dir=DATA_DIR):
```

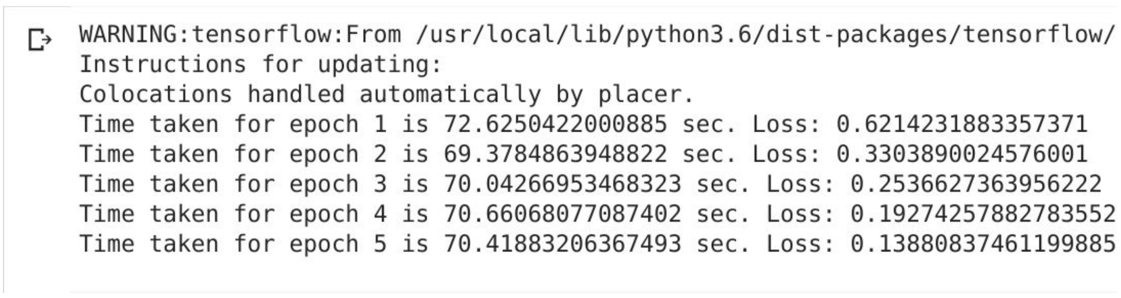
```
    if not os.path.exists(checkpoint_basedir + '/S0'):
      os.makedirs(checkpoint_basedir + '/S0')
    self.name = self.__class__.__name__ + '_'
    self.encoder = Encoder()
    self.classifier = Classifier()
    self.data_dir = data_dir
    self.checkpoint_dir = checkpoint_basedir + '/S0'
                             self.training_stats_file        =
os.path.join(training_stats_dir,          self.name          +
'training_stats.pickle')
                                     self.optimizer          =
tf.optimizers.Adam(learning_rate=learning_rate)
    self.batch_size = batch_size
                               self.checkpoint_prefix        =
os.path.join(self.checkpoint_dir, self.name + "ckpt")
                                    self.checkpoint          =
tf.train.Checkpoint(optimizer=self.optimizer,
                                        encoder=self.encoder,
classifier=self.classifier)

  @property
  def encoder(self):
    return self.__encoder

  @encoder.setter
  def encoder(self, encoder):
    self.__encoder = encoder

  @property
  def classifier(self):
    return self.__classifier

  @classifier.setter
  def classifier(self, classifier):
    self.__classifier = classifier

  def restore_from_checkpoint(self):
                                    checkpoint_file          =
tf.train.latest_checkpoint(self.checkpoint_dir)
    self.checkpoint.restore(checkpoint_file)
  def loss(self, logits, labels):
                                                            return
tf.nn.sparse_softmax_cross_entropy_with_logits(labels=labels,
logits=logits)

  def train(self, epochs=1, train_on='mnist'):
    training_stats = []
     x_train, y_train = load_mnist(data_dir=self.data_dir) if
train_on=='mnist' else load_svhn(data_dir=self.data_dir)

    for epoch in range(epochs):

      avg_loss = 0
      start_time = time.time()
      it = 0
             for  start,  end  in  zip(range(0,  len(x_train),
self.batch_size),     range(self.batch_size,     len(x_train),
self.batch_size)):
        it += self.batch_size
        with tf.GradientTape() as tape:
                                                  logits     =
self.classifier(self.encoder(x_train[start: end]))
          loss = self.loss(logits, y_train[start: end])
          avg_loss += np.sum(loss.numpy())
                           gradients   =   tape.gradient(loss,
self.encoder.variables + self.classifier.variables)
          self.optimizer.apply_gradients(zip(gradients,

self.encoder.variables + self.classifier.variables))
      time_taken = time.time() - start_time
      avg_loss /= it
             training_stats.append((epoch   +   1,   time_taken,
avg_loss))
                         self.checkpoint.save(file_prefix     =
self.checkpoint_prefix)
      with open(self.training_stats_file, 'wb') as f:
        pickle.dump(training_stats, f)
        print('Time  taken  for  epoch  {}  is  {}  sec.  Loss:  {}
'.format(epoch + 1, time_taken, avg_loss))

  def predict(self, X):
    logits = self.classifier(self.encoder(X))
    return np.argmax(logits, axis=-1)

  def accuracy(self, labels, predictions):
    correct = tf.equal(predictions, labels)
    return tf.reduce_mean(tf.cast(correct, tf.float32))
```

Saving the progress to GCloud bucket

```
gsutil -m cp -r {BASE_DIR} gs://{BUCKET_NAME}
```

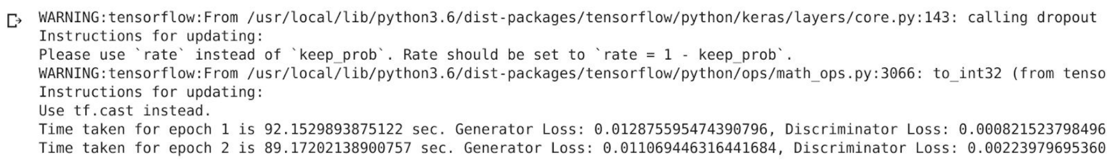
APPENDIX 2: SYSTEM SNAPSHOTS

Training the encoder and classifier (step 0) on Google Colab



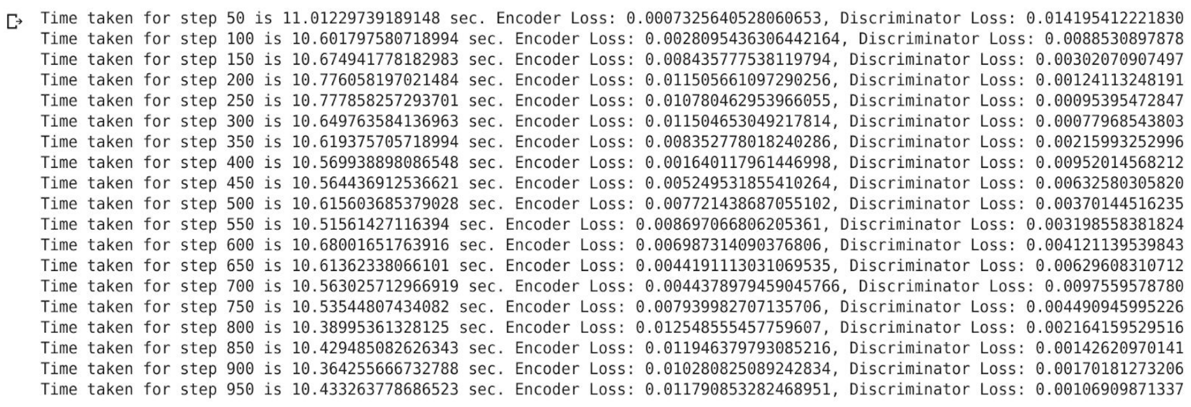
```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/tensorflow/
Instructions for updating:
Colocations handled automatically by placer.
Time taken for epoch 1 is 72.6250422000885 sec. Loss: 0.6214231883357371
Time taken for epoch 2 is 69.3784863948822 sec. Loss: 0.3303890024576001
Time taken for epoch 3 is 70.04266953468323 sec. Loss: 0.2536627363956222
Time taken for epoch 4 is 70.66068077087402 sec. Loss: 0.19274257882783552
Time taken for epoch 5 is 70.41883206367493 sec. Loss: 0.13880837461199885
```

Training the Generator S (Step 1)



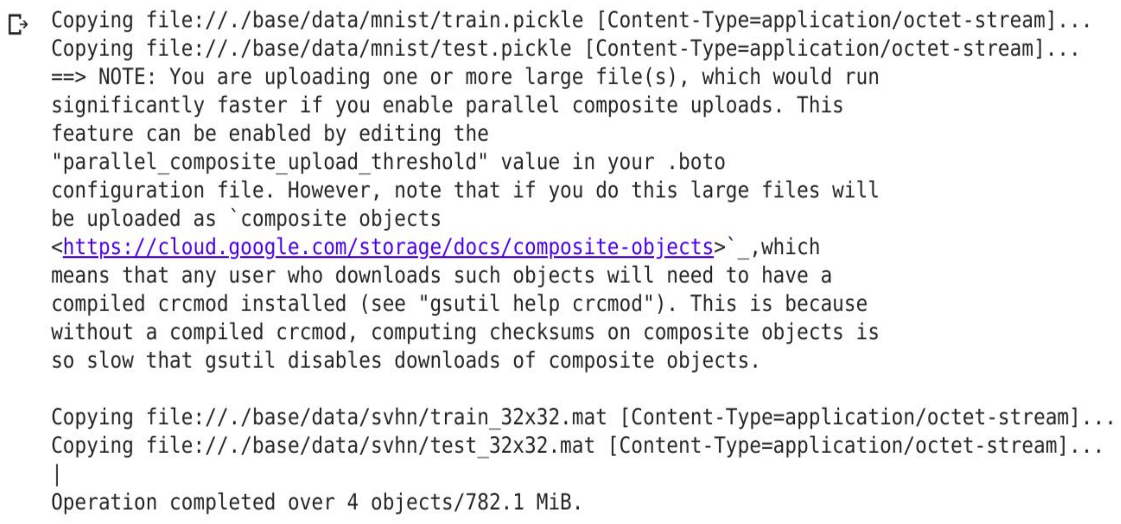
```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/tensorflow/python/keras/layers/core.py:143: calling dropout
Instructions for updating:
Please use `rate` instead of `keep_prob`. Rate should be set to `rate = 1 - keep_prob`.
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/tensorflow/python/ops/math_ops.py:3066: to_int32 (from tenso
Instructions for updating:
Use tf.cast instead.
Time taken for epoch 1 is 92.1529893875122 sec. Generator Loss: 0.012875595474390796, Discriminator Loss: 0.000821523798496
Time taken for epoch 2 is 89.17202138900757 sec. Generator Loss: 0.011069446316441684, Discriminator Loss: 0.00223979695360
```

Training the domain-invariant encoder (step 2) on Google Colab



```
Time taken for step 50 is 11.01229739189148 sec. Encoder Loss: 0.0007325640528060653, Discriminator Loss: 0.014195412221830
Time taken for step 100 is 10.601797580718994 sec. Encoder Loss: 0.0028095436306442164, Discriminator Loss: 0.0088530897878
Time taken for step 150 is 10.674941778182983 sec. Encoder Loss: 0.008435777538119794, Discriminator Loss: 0.00302070907497
Time taken for step 200 is 10.776058197021484 sec. Encoder Loss: 0.011505661097290256, Discriminator Loss: 0.00124113248191
Time taken for step 250 is 10.777858257293701 sec. Encoder Loss: 0.010780462953966055, Discriminator Loss: 0.00095395472847
Time taken for step 300 is 10.649763584136963 sec. Encoder Loss: 0.011504653049217814, Discriminator Loss: 0.00077968543803
Time taken for step 350 is 10.619375705718994 sec. Encoder Loss: 0.008352778018240286, Discriminator Loss: 0.00215993252996
Time taken for step 400 is 10.569938898086548 sec. Encoder Loss: 0.001640117961446998, Discriminator Loss: 0.00952014568212
Time taken for step 450 is 10.564436912536621 sec. Encoder Loss: 0.005249531855410264, Discriminator Loss: 0.00632580305820
Time taken for step 500 is 10.615603685379028 sec. Encoder Loss: 0.007721438687055102, Discriminator Loss: 0.00370144516235
Time taken for step 550 is 10.51561427116394 sec. Encoder Loss: 0.008697066806205361, Discriminator Loss: 0.003198558381824
Time taken for step 600 is 10.68001651763916 sec. Encoder Loss: 0.006987314090376806, Discriminator Loss: 0.004121139539843
Time taken for step 650 is 10.61362338066101 sec. Encoder Loss: 0.0044191113031069535, Discriminator Loss: 0.00629608310712
Time taken for step 700 is 10.563025712966919 sec. Encoder Loss: 0.0044378979459045766, Discriminator Loss: 0.0097559578780
Time taken for step 750 is 10.53544807434082 sec. Encoder Loss: 0.007939982707135706, Discriminator Loss: 0.004490945995226
Time taken for step 800 is 10.38995361328125 sec. Encoder Loss: 0.012548555457759607, Discriminator Loss: 0.002164159529516
Time taken for step 850 is 10.429485082626343 sec. Encoder Loss: 0.011946379793085216, Discriminator Loss: 0.00142620970141
Time taken for step 900 is 10.364255666732788 sec. Encoder Loss: 0.010280825089242834, Discriminator Loss: 0.00170181273206
Time taken for step 950 is 10.433263778686523 sec. Encoder Loss: 0.011790853282468951, Discriminator Loss: 0.00106909871337
```

Saving the progress to GCloud Bucket



```
Copying file://./base/data/mnist/train.pickle [Content-Type=application/octet-stream]...
Copying file://./base/data/mnist/test.pickle [Content-Type=application/octet-stream]...
==> NOTE: You are uploading one or more large file(s), which would run
significantly faster if you enable parallel composite uploads. This
feature can be enabled by editing the
"parallel_composite_upload_threshold" value in your .boto
configuration file. However, note that if you do this large files will
be uploaded as `composite objects
<https://cloud.google.com/storage/docs/composite-objects>`_,which
means that any user who downloads such objects will need to have a
compiled crcmod installed (see "gsutil help crcmod"). This is because
without a compiled crcmod, computing checksums on composite objects is
so slow that gsutil disables downloads of composite objects.

Copying file://./base/data/svhn/train_32x32.mat [Content-Type=application/octet-stream]...
Copying file://./base/data/svhn/test_32x32.mat [Content-Type=application/octet-stream]...
|
Operation completed over 4 objects/782.1 MiB.
```