Alma Mater Studiorum · Università di Bologna

SCHOOL OF ENGINEERING AND ARCHITECTURE
Second Cycle Degree in Computer Engineering

Learning Features Across Tasks and Domains

Supervisor:
Chiar.mo Prof.
Luigi Di Stefano

Candidate:
Adriano Cardace

Co-supervisors:
Pierluigi Zama Ramirez
Luca De Luigi
Samuele Salti
Luca Paganelli

Session III
Academic Year 2018/2019

"Life can be seen as machine learning problem: Earth is the landscape of your loss function and you move around all the time trying to optimize your goal, happiness."

Abstract

The absence of in-domain labeled data hinders the applicability of powerful deep neural networks. Unsupervised Domain Adaptation (UDA) methods have emerged to exploit such models even when labeled data is not available in the target domain. All these techniques aim to reduce the distribution shift problem that afflicts these models when trained on one dataset and tested in a different one. However, most of the works, do not consider relationships among tasks to further boost performances. In this thesis, we study a recent method called AT/DT (Across Tasks Domain Transfer), that seeks to apply Domain Adaptation together with Task Adaptation, leveraging on the correlation of two popular Vision tasks such as Semantic Segmentation and Monocular Depth Estimation. Inspired by the Domain Adaptation literature, we propose many extensions to the original work and show how these enhance the framework performances. Our contributions are applied at different levels: we first study how different architectures affect the transferability of features across tasks. We further improve performances by deploying Adversarial training. Finally, we explore the possibility of replacing Depth Estimation with popular Self-supervised tasks, demonstrating that two tasks must be semantically connected to be able to transfer features among them.

Sommario

L'assenza di dati annotati limita le applicazioni di potenti modelli come le reti neurali. Tuttavia, grazie alle recenti tecniche di Unsupervised Domain Adaptation (UDA), risulta possibile utilizzare questi strumenti anche quando non sia hanno a disposizione dati labellati. Lo scopo di queste tecniche è quindi quello di ridurre il problema dello shift tra distribuzioni quando un modello viene allenato su un determinato dataset e testato in condizioni differenti. L'obiettivo di questa tesi consiste nello studiare ed estendere un recente metodo chiamato AT/DT (Across Tasks Domain Transfer) che cerca di combinare tecniche di Domain Adaptation e Task Adaptation sfruttando la correlazione che esiste tra due tipici problemi della Computer Vision: Semantic Segmentation e Monocular Depth Estimation. Prendendo ispirazione dalle strategie di Domanin Adaptation presenti in letteratura, si propongono diverse estensioni che possono essere applicate al framework originale per migliorarne le prestazioni. Le contribuzioni di questa tesi agiscono su diversi aspetti. Come prima cosa, viene studiata l'importanza dell'architettura di base dell'intero framework e come questa impatta sulla trasferibilità tra feature di task diversi. Succesivamente, le prestazioni vengono ulteriormente migliorate sfruttando le recenti tecniche di allenamento di tipo Adversarial. Infine, vengono esplorati diversi problemi di tipo Self-supervised in alternativa alla Monocular Depth Estimation, mostrando che per garantire il successo di AT/DT è di fondamentale importanza la presenza di una forte connessione tra i task impiegati.

Contents

Abstract **i**

List of Figures **v**

List of Tables **vii**

1 Introduction **1**

2 Transfer Learning and Domain Adaptation **4**

2.1 Notations and Definitions 4

2.2 Approaches for solving Domain Adaptation 8

2.2.1 Discrepancy-Based Approaches 9

2.2.2 Adversarial-Based Approaches 21

2.2.3 Reconstruction-Based Approaches 26

2.2.4 Self-supervised approaches for Domain Adaptation 34

2.3 Performances analysis 37

3 Learning Features across Tasks and Domains: AT/DT **41**

3.1 Setting 42

3.2 Architecture 44

3.3 Training and Evaluation protocol 47

4 AT/DT Extended **49**

4.1 Ablation study on the number of channels of the transfer network 51

4.2 Batch normalization in the transfer network 52

4.3 Deeplab vs UNET as backbone network 54
4.4 Flat transfer network 56
4.5 Adversarial training 58
4.5.1 Domain alignment through domains 58
4.5.2 Task mapping with adversarial training 59
4.6 Self-supervised learning 59
4.6.1 Autoencoder 60
4.6.2 Rotation prediction 61
4.6.3 Image Colorization 62
4.6.4 Edge detection 63

5 Results **65**
5.1 Results with different architectures 65
5.2 Results with Adversarial training 69
5.3 Results with Self-supervised tasks 71

6 Technologies **73**
6.1 Tensorflow 74
6.2 GCP 75
6.2.1 Compute Engine 75
6.2.2 Google Cloud Storage 76
6.2.3 Big Data Services 76

7 Conclusions and future work **78**

Bibliography **79**

List of Figures

2.1 Distribution shift 5
2.2 Different label space 6
2.3 Transfer Learning classification 7
2.4 Fragile co-adaptation and representation specificity 11
2.5 Using soft labels 12
2.6 Computing soft labels through distillation 14
2.7 Ideal feature space 15
2.8 DDC architecture 17
2.9 Batch Normalization adaptation 21
2.10 PixelDA 22
2.11 CoGAN architecture 23
2.12 Domain Adversarial Neural Network 24
2.13 Adversarial Discriminative Domain Adaptation 26
2.14 Deep Reconstruction Classification Network 27
2.15 Domain Separation Network 28
2.16 Domain adaptation through CycleGANs 29
2.17 Ablation study on the effect of the semantic and cycle consistency loss . 33
2.18 CyCADA architecture 34
2.19 Alignment through weak supervision 35
2.20 Weak supervised learning for Domain Adaptation 36
2.21 Weak supervised learning with adversarial training for Domain Adaptation 36

3.1 Cityscapes training data 43

3.2 Carla dataset 43
3.3 AT/DT framework 44
3.4 Decoder architecture 46
3.5 Transfer architecture 47

4.1 Results with different number of channels in $G_{1\rightarrow 2}$ 51
4.2 Visual effect of different batch size 53
4.3 UNET 55
4.4 DeepLab Architecture 56
4.5 Flat transfer architecture 57
4.6 Image reconstruction with an Autoencoder 61
4.7 Rotation Decoder 62
4.8 Example of Image Colorization 63

5.1 AT/DT qualitative examples 67
5.2 AT/DT with ASPP module qualitative examples 67
5.3 AT/DT with skips connections qualitative examples 68
5.4 AT/DT Flat transfer qualitative examples 68
5.5 Segmentation maps obtained with adversarial training across domains . . 70
5.6 Segmentation maps obtained with adversarial training across tasks 70

6.1 Example of a Beam pipeline executed in Dataflow 77

List of Tables

2.1 Possible instances of Image to Image adaptation 31
2.2 Categorization of several DA methods 38
2.3 Domain Adaptation on the Office dataset 39
2.4 Domain Adaptation on digits datasets 40

4.1 Batch Normalization effect on the transfer network 53

5.1 Experimental results with different architectures on the **Cityscapes** validation set 66
5.2 Experimental results with adversarial training on the **Cityscapes** validation set 69
5.3 Experimental results on the **Cityscapes** validation set when mapping different tasks to Semantic Segmentation. Best results highlighted in bold. 71
5.4 Experimental results on the **Carla** validation set when mapping different tasks to Semantic Segmentation. Best results highlighted in bold. 72

Chapter 1

Introduction

In recent years, Machine Learning, and Deep Learning in particular, has been extensively used for solving Computer Vision related tasks. Since 2012, when AlexNet [22] was introduced, striking results have been achieved. This progress is mostly due to the undeniable effectiveness of Convolutional Neural Networks (CNNs). CNNs achieve amazing performances when trained with high-quality annotated training data. Considering that many pre-trained models are publicly available, one can even reuse these networks and solve complex tasks with almost zero effort. When solving a classification problem for instance, we can use one of the standard network architectures (ResNet, VGG, etc.) and train it using our dataset. This would likely lead to very good results if the data is correctly annotated. Moreover, CNNs have been proved to be effective features extractors. These features can be used to solve many challenging tasks that require a complete understanding of an image, such as semantic segmentation, depth estimation, etc. In 2014, [31] reached state-of-the-art results on several tasks by simply applying a linear SVM classifier on top of a CNN. So far though, most of the approaches tackle each task in an isolated way: collect the dataset, train the model and test it. What if we would like to transfer the knowledge acquired from a previous task to a new one? Or train our network in a specific domain and deploying it to another one? This is what humans do, after all. We progressively learn new things thanks to what we learned in the past. In a sense, our goal is to make these algorithms more human-like. Exploiting previously acquired knowledge can be helpful in many ways. For example, it is well known that

collecting large dataset for challenging tasks such as segmentation is not practically feasible, hence strategies able to reuse old datasets or synthetic ones could save us a lot of time and money. In general, transferring knowledge is a tough task for computers, and although more and more researchers are constantly trying to push the limits of Deep Learning, we are still far from human capabilities. Typically, in the field of Machine learning, the problem of transferring knowledge is referred to as Transfer Learning. It is a broad and active field of research, although it was a well know problem in Computer Vision even before the advent of Deep Learning. Before diving into Transfer Learning and Domain Adaptation, which is a specific type of the former and the real focus of this thesis together with task adaptation, it is important to briefly clarify some simple concepts that will be used throughout this work. Depending on the available data, we may have different flavors of machine learning problems. The most commons are the following:

- Supervised Learning: under the supervised setting, we are given input-label pairs, and the goal is to learn a mapping function between input and labels. A simple example is image classification, in which the input is an image and the label is the class it belongs to.

- Unsupervised Learning: in this scenario, labels are not available. The task here is to learn a feature space that captures the characteristics of the input data while maximizing an objective function without the need of any annotations. Common tasks are clustering and anomaly detection.

- Semi-Supervised Learning: These algorithms seek to learn from both unlabeled and labeled samples. The assumption is that both are sampled from the same or similar distribution, therefore they can be used together to improve performances. Indeed, on many real occasions, we have a small amount of data that is labeled, while tons of data is not annotated, so it is fundamental to have techniques able to exploit both kinds of data

- Self-Supervised Learning: this is a relatively recent learning technique where the training data is autonomously labeled. It can be seen as a type of supervised

learning, although in this case the datasets are not manually labeled by humans, but they are annotated automatically in a surrogate task. By designing a complex task from which labels come for free, it is possible to learn good features to be used in the target task. For example [14] proposed to rotate the input image and to predict the degree of rotation. To solve this kind of tasks, a high-level semantic understanding is required. As a result, the model learns representations that can subsequently be used to solve the downstream problem. Self-Supervised Learning can be also referred to as Weak-Supervised Learning.

Transfer Learning can be applied in all the previous settings since these concepts are orthogonal, therefore it is important to have clear in mind all the possibilities to avoid confusion. Let's now clarify with an intuitive example the Transfer Learning problem. In the following chapter instead, we will give a formal definition, analyzing in particular detail the case of Domain Adaptation.

Let us assume, for instance, that we want to build a model able to solve the semantic segmentation task, therefore the objective is to assign labels pixel-wise (buildings, trees, cars, pedestrians, traffic light, etc.). To this purpose, we can use the Cityscapes dataset [7]. After testing the model in the corresponding test set, we can achieve a high score in terms of mIoU (the metric usually measured for Semantic Segmentation). Then we test the same system in another dataset, Kitti [2], and things go very wrong: the mIoU score drops badly. The reason why the model does not perform well is that the domain changed. For example, and even though we test our model on street scenes, the light conditions can vary across domains. This is where domain adaptation comes to rescue. Domain adaptation is a sub-discipline of machine learning which deals with these kind of scenarios. In general, domain adaptation uses labeled data of the source domains to solve the same task in a target domain. Based on the amount of data that we have on the target domain, we may be in one of the settings defined above, i.e. supervised, unsupervised, etc.

Chapter 2

Transfer Learning and Domain Adaptation

2.1 Notations and Definitions

We are now ready to give a formal definition of Transfer Learning and provide a classification of all the possible sub-cases. Then, we will present several approaches that can be found in the literature to alleviate this problem. The notions used in this thesis is the same used in most of the surveys about Transfer Learning [29] [38]. A domain $\mathcal{D}$ consists of a feature space $\mathcal{X}$ and a marginal probability distribution P(X), where $X = \{x_1, ..., x_n\} \in \mathcal{X}$. We may consider X as our training data. If the domain consists of RGB images with size $W \times H \times C$, we have a four dimensional feature space of size $W \times H \times C \times 256$, i.e. all the three channels images that can be generated. A dataset can be thought as a volume inside the whole feature space. For example all the natural images, lie in a specific portion of the four dimensional feature space, and we are only interested to model its marginal probability distribution P(X). For a given domain $\mathcal{D} = \{\mathcal{X}, P(X)\}$, a task $\mathcal{T}$ is defined by two components, $\mathcal{T} = \{\mathcal{Y}, f(\cdot)\}$, where $\mathcal{Y}$ is the label space and $f(\cdot)$ an objective predictive function that under a probabilistic perspective can be seen as the conditional probability distribution $P(Y|X)$. In the classical supervised setting, $P(Y|X)$ can be learned directly from the labeled data $\{x_i, y_i\}$, where $x_i \in \mathcal{X}$ and $y_i \in \mathcal{Y}$. We can generalize to the situation in which we have two domains, the

source domain with sufficient labeled data $\mathcal{D}^s = \{X^s, P(X)^s\}$, and the target one with a small amount of labeled data or no annotated data $\mathcal{D}^t = \{X^t, P(X)^t\}$. $\mathcal{D}^t$ can thereby be decomposed in two sets: the labeled part, $\mathcal{D}^{tl}$, and the unlabeled part, $\mathcal{D}^{tu}$. The entire target domain is $\mathcal{D}^t = \mathcal{D}^{tl} \cup \mathcal{D}^{tu}$. Each domain is coupled with its corresponding task: the former is $\mathcal{T}^s = \{\mathcal{Y}^s, P(Y^s|X^s)\}$, and the latter is $\mathcal{T}^t = \{\mathcal{Y}^t, P(Y^t|X^t)\}$. The previous definition gives as a general framework that can be instantiated in different settings to obtain several cases. For example, if we fix $\mathcal{D}^s = \mathcal{D}^t$ and $\mathcal{T}^s = \mathcal{T}^t$, we are in the traditional Machine Learning case. Since both domain and task are composed by two elements, we have in total four possibilities:

1. The two dataset are different because the feature space is different: $\mathcal{X}^s \neq \mathcal{X}^t$. A typical example can be digits recognition. The source domain only contains one channel images, while in the test domain we have to classify colored digits.

2. The difference between the two dataset is caused by a distribution shift: $P(X^s) \neq P(X^t)$. fig. 2.1 illustrates such scenario. In a simplistic two dimensional feature space, we have images belonging to the same class (the digits 5) that are occupying different portions of the feature space. Hence, training a classifier on domain A, and applying it on domain B, would lead to very poor performances.

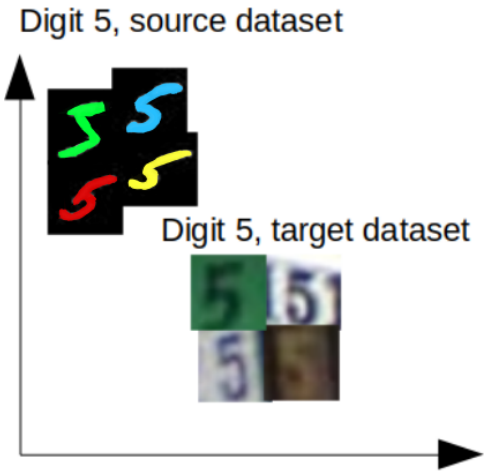


Figure 2.1: Distribution shift

3. Tasks divergence is caused by a label space discrepancy: $\mathcal{Y}^s \neq \mathcal{Y}^t$. A typical

example of this case if face recognition, because in the source domain we may have some faces, while in the target domain we would like to recognize other people.

Figure 2.2: Different label space

4. The conditional probability distribution of source and target tasks are different: $P(Y^s|X^s) \neq P(Y^t|X^t)$. This case appears quite often in practice. This happens for example when we train a model for image classification on a balanced dataset, while the test set is strongly unbalanced.

We are now ready to introduce a formal definition of TL [29]:

Definition 1. *Given a source domain $\mathcal{D}^s$ and a learning task $\mathcal{T}^s$, a target domain $\mathcal{D}^t$ and a learning task $\mathcal{T}^t$,* ***transfer learning*** *aims to help to improve the learning of the target predictive function $f_t(\cdot)$ in $\mathcal{D}^t$ using the knowledge in $\mathcal{D}^s$ and $\mathcal{T}^s$, where $\mathcal{D}^s \neq \mathcal{D}^t$, or $\mathcal{T}^s \neq \mathcal{T}^t$.*

Based on this definition, Pan *et al.* proposed three sub-categories of TL to highlight the possible situations that may occur: inductive, transductive and unsupervised TL. fig. 2.3 outlines this three cases.

1. **inductive TL** In this scenario, the source and target domains can be the same or not, while the source and target tasks are always different from each other. We have at our disposal some labeled data in the target domain and we want to improve the target task by exploiting somehow information on the source domain. Depending upon whether the source domain contains labeled data or not, this can

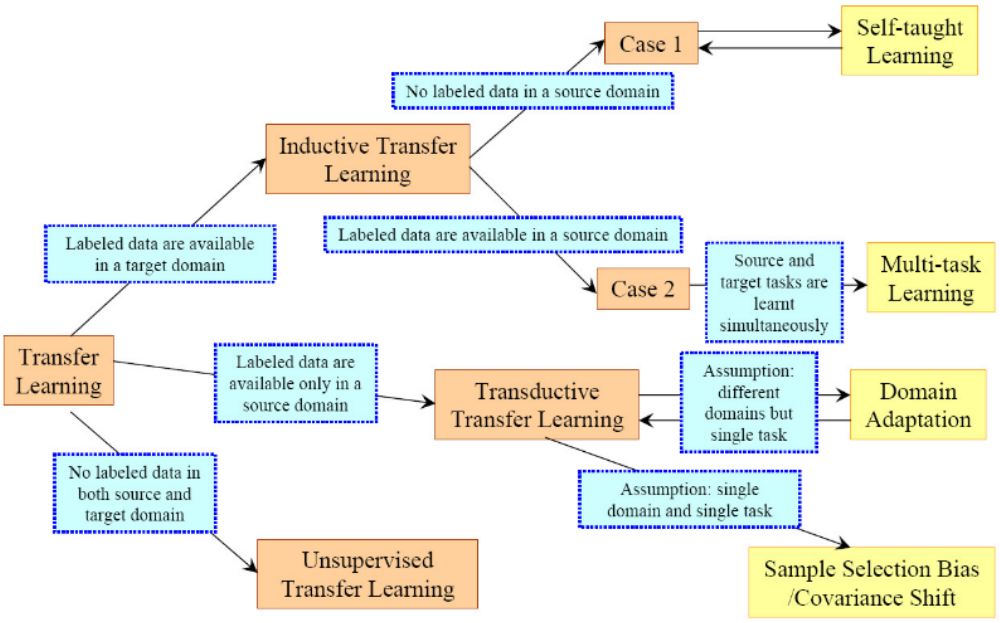


Figure 2.3: Transfer Learning classification [29]

be further divided into two subcategories, similar to multitask learning (but in this case we are aiming to improve performance only on the target, not in both as in multitask learning) and self-taught learning, which instead assumes that the label space between source and target is different, yet related. Thus, it is directly related to inductive learning when no labeled data is available in the source domain.

2. **transductive TL** Here, the source and target domain are different, while tasks remain the same. In this situation, we have labeled data for the source domain, and we would like to transfer this knowledge to the target task, for which we only have unlabeled data or a small amount of annotated data. This is the category where Domain Adaptation fell if we assume that the domains are different because of a domain shift, i.e. $P(X^s) \neq P(X^t)$. In the literature, this is sometimes referred to as *homogeneous DA*. In the case of domain difference caused by a different feature space ($\mathcal{X}^s \neq \mathcal{X}^t$), we talk about *co-variate shift* or *heterogeneous DA*.

3. **unsupervised TL** In the case of unsupervised TL, although they vary, there are similarities between the source and target tasks. Similarly to the inductive case,

the domains can be the same or not, but unsupervised TL focuses on solving unsupervised learning tasks in the target domain when no labeled data is available in both domains.

DA can be further categorized into supervised, semi-supervised and unsupervised. These terms have the same semantic of the ones introduced in chapter 1, but applied to the context of DA:

- In the supervised DA, a small amount of annotated data for the target task is available ($\mathcal{D}^{tl}$), yet not enough to tackle the problem in a common supervised way. Therefore, we need to exploit $\mathcal{D}^s$ to obtain better performances in $\mathcal{D}^t$.
- In the semi-supervised DA, in addition to ($\mathcal{D}^{tl}$), we also have available unlabeled data, $\mathcal{D}^{tu}$, that can be used to gain knowledge for the target task.
- Finally, the unsupervised DA, is the most challenging case, since we only have unlabeled data for the target domain. Most of the approaches introduced in the next sections belong to this category. This reflects the interest of the research community, which is putting more effort into this particular case due to its generality and applicability.

2.2 Approaches for solving Domain Adaptation

Now that we understood where Domain Adaptation (abbreviated with just DA from now on) fits in the general Transfer Learning problem, we will move on describing possible solutions for the homogeneous DA, which is the real focus of this work. DA assumes that the target task remains the same as the source, as well as the feature spaces. Therefore the problem we have to face is a shift in the marginal probability distribution. As stated before, this issue was popular in Computer Vision well ahead of the rise of Deep Learning. For this reason, it is possible to find in literature many algorithms that try to reduce the domain shift between source and target domain with techniques that don't rely on deep neural networks. Our goal instead, is to follow the recent trend in the research community that heavily makes use of these powerful architectures. Intuitively, we want

to embed the process of DA in the training process of a neural network, in order to have an end-to-end architecture that can be optimized via back-propagation. The important question is then how do we extract good features that are meaningful and reusable across domains using neural networks? Although many methods have been proposed in recent years, we can identify two main categories. The first one is called domain-invariant feature learning, while the second one is referred to as domain mapping. In the former case, the domain adaptation methods try to align source and target distribution by creating a domain invariant feature space, hence no matter the initial distribution, two samples that belong to two different domains but share the same class, will look the same in feature space. The idea is that if we have such representation that also preserves some discriminative properties, then a classifier that works well for the source domain should work reasonably well in the target domain too. An alternative to creating a domain invariant feature space is mapping from one domain to the other. This mapping can be learned at the pixel-level or at the feature-level, although the former is more popular. In addition to this categorization, we can also give a more fine-grained classification proposed in [38], which identifies three main categories: Discrepancy-Based Approaches, Adversarial-Based Approaches, and Reconstruction-Based Approaches. In the following sections we, analyze several methods belonging to these classes. Each of these algorithms falls either in the domain-invariant feature learning or in the domain mapping category. Then, a comparison in terms of performances is given. Finally, we present the general framework called AT/DT [43] on which this work is built upon. As we will see, AT/DT also makes use of task adaptation in order to boost performances for DA. We believe that having a complete understanding and knowledge of all the previous techniques proposed in the literature may be useful for developing new ideas. For this reason, even though AT/DT is a general framework, we will treat it keeping in mind that our final goal is DA.

2.2.1 Discrepancy-Based Approaches

Discrepancy-based methods are probably the most simple, yet effective. The requirement is that we have available some annotated data for the target domain, thus they only work in the supervised or Semi-supervised setting. All the methods in this category

are based on fine-tuning, a technique that seeks to reduce domain shift by pretraining a neural network with the source data and using these weights to initialize the model for the target task. The underlying idea is that by pretraining on a similar but different domain, we are still able to get close to a good solution, so we can start from this point rather than from scratch. In a sense, we are guiding the network in the right direction. The reason why this works is that the first layers are able to capture simple features such as colors, lines, and shapes. Such first-layers features appear not to be specific to a particular dataset or task, but they eventually transition from general to going towards the final layers of the network. Yosinski *et al* [41], studied extensively the transferability of features learned by CNNs, and discovered that features learned by deep networks and used without fine-tuning have limitations due to fragile-coadaptation and representation specificity. Their conclusion was that fine-tuning is in general a good idea since it allows the network to adapt the weights on the target domain. The way we perform fine-tuning though depends on some aspects. Typically, there are two possible scenarios assuming that source and target domains are similar:

1. A small dataset is available in the target domain. In this case, it might not be convenient to fine-tune the whole network. If the two domains are similar, we can expect that even high-level features are well transferable. Moreover, since only little annotated data is available, there is a high risk of overfitting. This suggests to attach and train some new randomly initialized fully-connected layers to the pre-trained network keeping fixed the first layers.

2. The target domain is large. In this case, it is suggested to fine-tune the whole architecture. Doing this way, we are able to exploit the source domain by pretraining and to generalize better in the target domain.

Discrepancy-based approaches can be divided into three major sub-categories: Class Criterion, Architecture Criterion and Statistic Criterion. We will analyze each of these categories and provide some examples.

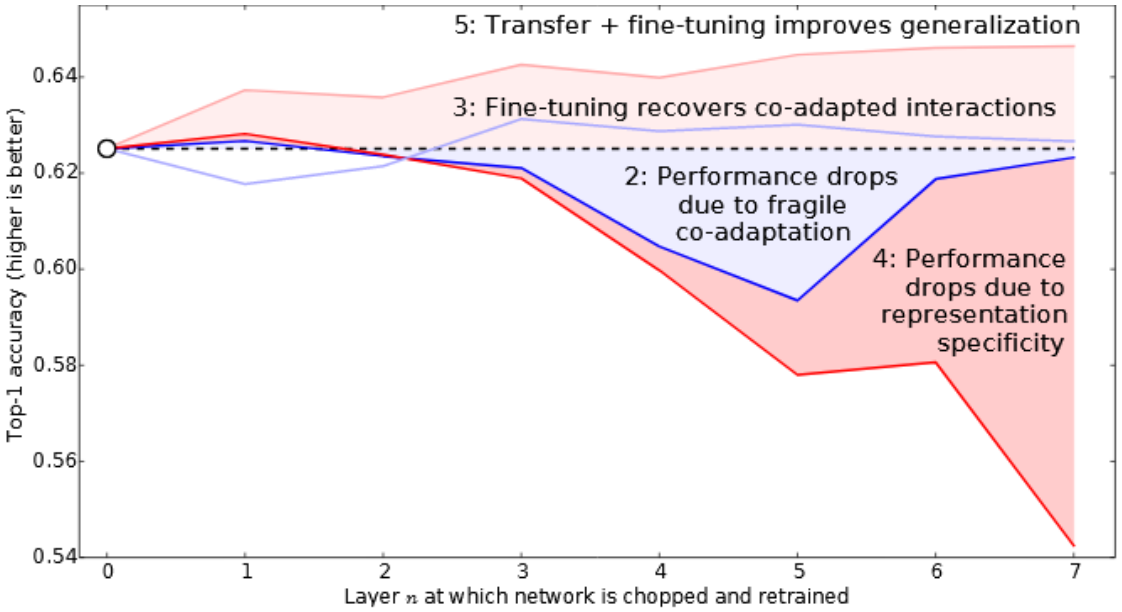


Figure 2.4: Results from [41] when training a neural network on dataset A and B. 1) The dotted line represents the baseline: accuracy of the network on dataset B. 2) A network trained on dataset B, suffers of fragile co-adaptation when freezing some of the layers and training in B again. 3) Fine-tuning the whole architecture allows to recover from fragile co-adaptation. 4)A network trained on A is directly used in B. The higher is the layer at which we chop, the lower is the accuracy because of representation specificity. 5) Transfer features from domain A and fine-tuning all layers performs better than the baseline.

2.2.1.1 Class Criterion

Class Criterion techniques directly use the labels of the target domain to minimize the objective function for the target task, as in a classical fine-tuning strategy. Several losses can be used to do this. The most popular is cross entropy, but other variations have been proposed. For example, [35] used a softer version of cross entropy introduced originally by Hinton *et al.* [16]:

$$q_i = \frac{\exp\left(z_i/T\right)}{\sum_j \left(\exp\left(z_j/T\right)\right)}$$

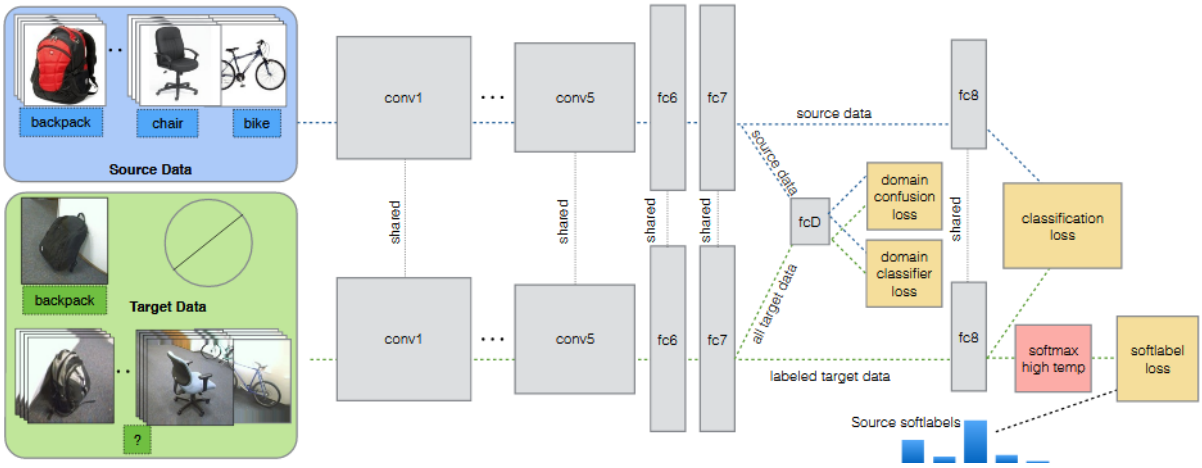


Figure 2.5: architecture from [35] that uses soft labels

where z_i is the logit computed by the last layer. T is a parameter called *Temperature*, that when it is set to 1, gives the standard cross entropy, while when increased, it has the effect of softening the final probability distribution. This allows us to reduce spikes and to preserve information between classes. By using these distributions as labels rather than the classical one hot encoding representation, we are teaching the network not only which is the correct class, but also how the classes are related between each other. This trick can be extremely helpful when little data is available and it is called *distillation*. The full architecture proposed in [35] that makes use of this idea is depicted in fig. 2.5. Source data (x_s, y_s) is initially fed to the CNN (from *conv1* to *fc7*) to find a good latent representation. The parameters of this CNN are denoted with θ_{repr}. On top of these layers, a final layer θ_{C} (*fc8*), considered as the classifier, is used to solve the classification task on source data. The loss function, in this case, is the standard cross entropy:

$$\mathcal{L}_C\left(x, y; \theta_{\text{repr}}, \theta_C\right) = -\sum_k \mathbb{1}[y = k] \log p_k$$

Then, the model is augmented with a domain classifier, namely fcD, that tries to recognize to which domain the input vector belongs. These vectors are computed again with the same base CNN, but feeding images from both domains. On the other hand, the CNN is trained to fool the discriminator. By doing this, if a classifier works well on the source domain, it should work reasonably well on the target domain too. This goal can

be obtained by training θ_{repr} to minimize the cross entropy error between the output predicted domain labels and a uniform distribution over domain labels. This is equivalent to maximize domain confusion since we are forcing the output of the discriminator to be uniform (i.e. each class with the same probability):

$$\mathcal{L}_{\text{conf}}\left(x_s, x_t, \theta_D; \theta_{\text{repr}}\right) = -\sum_d \frac{1}{D} \log q_d$$

On the other hand, θ_D is updated so that domain confusion is minimized:

$$\mathcal{L}_D\left(x_s, x_t, \theta_{\text{rep}}; \theta_D\right) = -\sum_d \mathbb{1}\left[y_D = d\right] \log q_d$$

with q corresponding to the softmax activations of the domain classifier. Through these loss functions, the model confuses the discriminator to align the marginal distributions of the two domains while solving the source task. This step can be thought of as the initialization step of the whole architecture. Afterward, the whole model is fine-tuned with soft-labels computed using the pre-trained architecture. To explain why fine-tuning is done with soft-labels rather than hard-labels, it is useful to remind the reader that although some annotated data is at our disposal in the target domain, this data is still limited. Moreover, sometimes this data is available only for a subset of the categories. For these reasons, training on hard labels is not satisfactory, while using soft labels, maximizes the impact of a single training target example because we are also exploiting the relationship between classes. The soft labels are computed by averaging the output distributions of source examples for each class. To make it crystal clear let us analyze an example. Let assume that in the source domain one of the classes is *bottle*. Then we collect all the outputs distributions when instances of such classes are provided to the CNN using a Softmax layer with temperature $T > 1$. Finally, by averaging these activations, we obtain a K-dimensional vector that represents the soft label for the class *bottle*. fig. 2.6 illustrates the whole process. Hence the loss function for a single sample drawn from the target domain becomes:

$$\mathcal{L}_{\text{soft}}\left(x_t, y_t; \theta_{\text{repr}}, \theta_C\right) = -\sum_i l_i^{(y_t)} \log p_i$$

where $l_i^{(y_t)}$ is the soft label for class y (computed after the initialization step and before fine-tuning) and p_i denotes the activation of the target image computed again with a

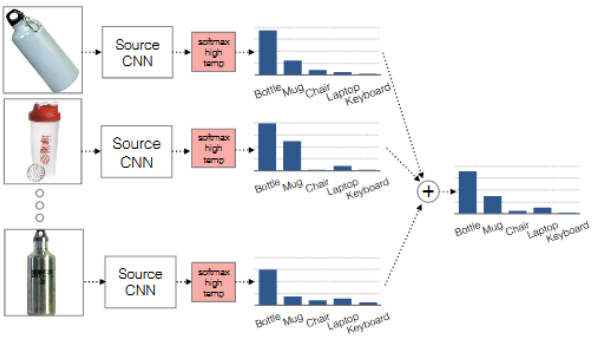


Figure 2.6: Computing soft labels through distillation [35]

softer version of softmax. The full loss function is a weighted sum of the previous:

$$\begin{aligned}\mathcal{L}\left(x_s, y_s, x_t, y_t, \theta_D; \theta_{\text{repr}}, \theta_C\right) = \\ \mathcal{L}_C\left(x_s, y_s, x_t, y_t; \theta_{\text{repr}}, \theta_C\right) \\ + \lambda \mathcal{L}_{\text{conf}}\left(x_s, x_t, \theta_D; \theta_{\text{repr}}\right) \\ + \mathcal{L}_D\left(x_s, x_t, \theta_{\text{rep}}; \theta_D\right) \\ + \nu \mathcal{L}_{\text{soft}}\left(x_t, y_t; \theta_{\text{repr}}, \theta_C\right)\end{aligned}$$

2.2.1.2 Statistic Criterion

Rather than using a softmax layer in combination with cross entropy, other techniques rely on statistical approaches to formulate a loss function to minimize. Moreover, differently from class criterion approaches, statistic criterion do not usually require labels and work under a unsupervised setting. An example of such methods is DTML [19] (Deep Transfer Metric Learning), which uses the MMD (Maximum Mean Discrepancy) criterion to align the distributions of the two domains. MMD is a measure of the difference between two probability distributions and it can be approximated by sampling from the two distributions without explicitly knowing their density function. Thanks to the kernel trick in fact, we are able to compare all the orders of statistic moments of the two distributions. In particular, if ϕ is a function in the unit ball in a Reproducing Kernel Hilbert Space (RKHS), it was shown that the MMD between two distributions s and t is 0 if and only if the two are identical. The formal definition of MMD is the

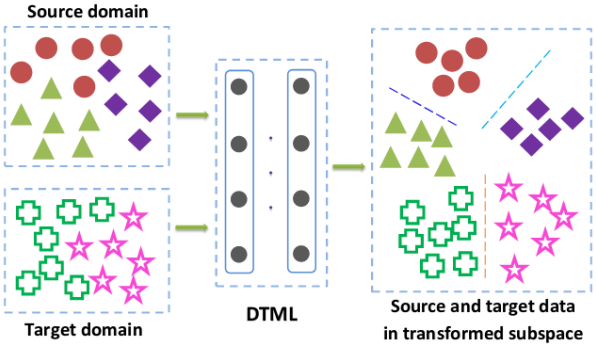


Figure 2.7: Ideal feature space [19]

following:

$$MMD(s,t) = \sup_{\|\phi\|_{\mathcal{H}} \leq 1} \left\| E_{\mathrm{x}^s \sim s} \left[\phi \left(\mathrm{x}^s \right) \right] - E_{\mathrm{x}^t \sim t} \left[\phi \left(\mathrm{x}^t \right) \right] \right\|_{\mathcal{H}}$$

DTML also uses the marginal Fisher analysis to enforce minimization between intra-class variations and maximization of the inter-class variations relying on annotated data only in the source domain. This is done in order to help a neural network to find a common space for the two domains, but also to make sure that similar classes of the two domains lie close in feature space. fig. 2.7 shows an example of such ideal latent space. The optimization process seeks to minimize two metrics. First, images from both domains are passed through a M-layers network, in order to obtain an N-dimensional vector representation. Then, the MMD criterion is approximated using the representation given by the neural network and minimizing the following constraint:

$$D_{ts}^{(m)} \left(\mathcal{X}_t, \mathcal{X}_s \right) = \left\| \frac{1}{N_t} \sum_{i=1}^{N_t} f^{(m)} \left(\mathbf{x}_{ti} \right) - \frac{1}{N_s} \sum_{i=1}^{N_s} f^{(m)} \left(\mathbf{x}_{si} \right) \right\|_2^2$$

with $f^{(m)} \left(\mathbf{x}_{ti} \right)$ and $f^{(m)} \left(\mathbf{x}_{si} \right)$ denoting the activations of the mth layer of the neural network with target and source domain data respectively. Finally, Fisher analysis criterion is applied by minimizing the following equation:

$$\min_{f(M)} J = S_c^{(M)} - \alpha S_b^{(M)} + \gamma \sum_{m=1}^{M} \left(\left\| \mathbf{W}^{(m)} \right\|_F^2 + \left\| \mathbf{b}^{(m)} \right\|_F^2 \right)$$

with $S_c^{(m)}$ and $S_b^{(m)}$ defined as:

$$S_c^{(m)} = \frac{1}{Nk_1} \sum_{i=1}^{N} \sum_{j=1}^{N} P_{ij} d^2_{f(m)} (\mathbf{x}_i, \mathbf{x}_j)$$

$$S_b^{(m)} = \frac{1}{Nk_2} \sum_{i=1}^{N} \sum_{j=1}^{N} Q_{ij} d^2_{f(m)} (\mathbf{x}_i, \mathbf{x}_j)$$

where P_{ij} is set to one if x_j is one of the *k1-intra-class* nearest neighbors of x_i, and zero otherwise; and Q_{ij} is set to one if x_j is one of the *k2-inter-class* nearest neighbors of x_i, and zero otherwise. To build the *k1-intra-class* and *k2-inter-class* graphs, labels for the source domain are required. *k1* and *k2* are usually empirically selected. $d^2_{f(m)}$ is simply the Euclidean distance between the feature vectors obtained by passing the two inputs up to the mth layer of the neural network. The third component is instead a regularization term. By minimizing this function, we are basically asking the network to put similar examples close in feature space, while examples from different classes should be drifted apart. By combining the two previous losses, we obtain:

$$\min_{f^{(M)}} J = S_c^{(M)} - \alpha S_b^{(M)} + \beta D_{ts}^{(M)} (\mathcal{X}_t, \mathcal{X}_s) + \gamma \sum_{m=1}^{M} \left(\left\| \mathbf{W}^{(m)} \right\|_F^2 + \left\| \mathbf{b}^{(m)} \right\|_2^2 \right)$$

After the training process, we have at our disposal a neural network capable of extracting deep hierarchical features that should be domain invariant. These feature vectors can then be used directly by a common classifier to actually solve the classification problem. MMD can also be deployed as a regularizer term together with cross entropy to formulate a loss function as done in DaNN[12] (Domain adaptive Neural Networks). This technique is different from DTML since it does not only learn a good latent space, but it also perform classification. More precisely, the optimization process is done in two steps. First, a mini-batch of data from the source domain is used to update all the parameters of the network such that the cross entropy error for classification in the source domain is minimized. Then, a batch containing data belonging to both domains is provided to minimize MMD. Thanks to the regularization term, the network is trained to minimize the classification error and at the same time, the hidden layer representations are encouraged to be invariant across different domains. By assuming D_s and D_t vectors drawn from distributions s and t respectively, and deploying the kernel trick, MMD is

estimated as:

$$MMD_e\left(D_s, D_t\right) = \left\| \frac{1}{M}\sum_{i=1}^{M}\phi\left(\mathrm{x}_i^s\right) - \frac{1}{N}\sum_{j=1}^{N}\phi\left(\mathrm{x}_j^t\right) \right\|_H$$
$$= \left(\frac{1}{n_s^2}\sum_{i=1}^{n_s}\sum_{j=1}^{n_s} k\left(\mathbf{x}_s^{(i)}, \mathbf{x}_s^{(j)}\right) + \frac{1}{n_t^2}\sum_{i=1}^{n_t}\sum_{j=1}^{n_t} k\left(\mathbf{x}_t^{(i)}, \mathbf{x}_t^{(j)}\right) \right.$$
$$\left. -\frac{2}{n_s n_t}\sum_{i=1}^{n_s}\sum_{j=1}^{n_s} k\left(\mathbf{x}_s^{(i)}, \mathbf{x}_t^{(j)}\right) \right)^{\frac{1}{2}}$$

Subsequently, this idea has been extended and applied together with CNNs in DDC [37] (Deep Domain Confusion) and achieved great success at the time. The idea is basically the same, i.e. minimize the classification error and at the same time reduce representation discrepancy via MMD, but they used a more powerful architecture, a CNN rather than a 2 layers feed-forward neural network (see fig. 2.8).

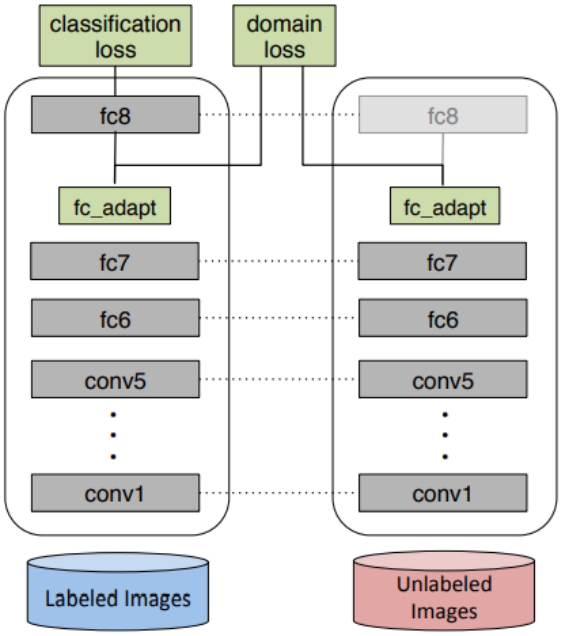


Figure 2.8: DDC Architecture [37]

Although MMD is a great metric to minimize domain discrepancy, it neglects class information, and this may lead to poor generalization. For this reason, [21] recently proposed a *Contrastive Domain Discrepancy* (CDD) metric built upon MMD that tries

to address this issue. The key idea of this new network called CAN (Contrastive Adaptation Network) is to explicitly model intra-class domain discrepancy and inter-class domain discrepancy by estimating proxy labels for the target domain through clustering techniques. CAN shows state-of-the-art performances and is able to obtain a latent space more similar to the one depicted in fig. 2.7. These gains come at a cost though, since it is considerably more complex than other methods. Starting from the MMD estimation formula, and taking in account class information (i.e. $y^s_{1:n_s}$ and $\hat{y}^t_{1:n_t}$), the CDD for class c_1 and c_2 given the current parameters of the networks and the current estimation for target samples is defined as

$$\hat{\mathcal{D}}^{c_1c_2}\left(\hat{y}^t_1, \hat{y}^t_2, \cdots, \hat{y}^t_{n_t}, \phi\right) = e_1 + e_2 - 2e_3$$

$$e_1 = \sum_{i=1}^{n_s}\sum_{j=1}^{n_s} \frac{\mu_{c_1c_1}\left(y^s_i, y^s_j\right) k\left(\phi\left(\boldsymbol{x}^s_i\right), \phi\left(\boldsymbol{x}^s_j\right)\right)}{\sum_{i=1}^{n_s}\sum_{j=1}^{n_s}\mu_{c_1c_1}\left(y^s_i, y^s_j\right)}$$

$$e_2 = \sum_{i=1}^{n_t}\sum_{j=1}^{n_t} \frac{\mu_{c_2c_2}\left(\hat{y}^t_i, \hat{y}^t_j\right) k\left(\phi\left(\boldsymbol{x}^t_i\right), \phi\left(\boldsymbol{x}^t_j\right)\right)}{\sum_{i=1}^{n_t}\sum_{j=1}^{n_t}\mu_{c_2c_2}\left(\hat{y}^t_i, \hat{y}^t_j\right)}$$

$$e_3 = \sum_{i=1}^{n_s}\sum_{j=1}^{n_t} \frac{\mu_{c_1c_2}\left(y^s_i, \hat{y}^t_j\right) k\left(\phi\left(\boldsymbol{x}^s_i\right), \phi\left(\boldsymbol{x}^t_j\right)\right)}{\sum_{i=1}^{n_s}\sum_{j=1}^{n_t}\mu_{c_1c_2}\left(y^s_i, \hat{y}^t_j\right)}$$

The previous equation measures intra-class domain discrepancy when $c_1 = c_2$ and inter-class discrepancy when $c_1 \neq c_2$. $\mu_{cc'}$ is a mask indicating whether $y_i = c$, $y_j = c'$. To compute the masks $\mu_{c_1c_2}$ and $\mu_{c_2c_2}$ the target labels are required. The authors provided ablation studies that estimating these labels through clustering techniques is more effective than simply using the prediction of the network as noisy labeler. The details on how this is done will be provided later. ϕ and k simply denote the mapping defined by the neural network and the selected kernel respectively. Finally, by computing $\hat{\mathcal{D}}^{c_1c_2}$ for all possible pairs of classes, we obtain:

$$\hat{\mathcal{D}}^{\text{cdd}} = \underbrace{\frac{1}{M}\sum_{c=1}^{M}\hat{\mathcal{D}}^{cc}\left(\hat{y}^t_{1:n_t}, \phi\right)}_{\text{intra}} \underbrace{- \frac{1}{M(M-1)}\sum_{c=1}^{M}\sum_{\substack{c'=1 \\ c'\neq c}}^{M}\hat{\mathcal{D}}^{cc'}\left(\hat{y}^t_{1:n_t}, \phi\right)}_{\text{inter}}$$

The backbone network is a ResNet50, that is optimized minimizing the classical cross-entropy on labeled source data. The last FC layers are used to extract compact feature representations of samples. At each training loop, the current configuration of the network is used to estimate the underlying label hypothesis of target samples through clustering. In particular, these proxy labels, together with the available source labels, are used to compute CCD and finally update the parameters of the network. The overall objective function is formalized as follows:

$$\min_{\theta} \ell = \ell^{ce} + \beta \hat{\mathcal{D}}_{\mathcal{L}}^{cdd}$$

Where l^{ce} indicates the standard cross-entropy error and $\hat{\mathcal{D}}_{\mathcal{L}}^{cdd}$ is the CDD metric computed for all the L FC layers. Regarding the target label estimation, spherical K-means with K equal to the number of classes is adopted: first, each target cluster center is initialized with the corresponding source class center, then it proceeds iteratively by attaching each sample to the class the minimizes the cosine similarity between the sample itself and the cluster centers, that are subsequently updated with the new attached samples. Two more important details that are essential in order to reduce domain shift between distributions. First, to partially reduce noise during the label estimation process, only classes with a certain amount of target samples assigned are considered during iteration T_e, i.e. points from the ruled out classed will not be sampled. Finally, to be able to compute CCD at each mini-beach, some precautions need to be made, because for any class C there should be points from both distributions, otherwise the inter-class term could not be estimated. To this purpose, class-aware sampling is deployed, which means that a random subset of classes is selected among the preserved C'_{T_e}, and then samples from these set of classes are taken to compose a mini-batch. The mini-batch for minimizing the cross-entropy error is instead drawn randomly as usual.

2.2.1.3 Architecture Criterion

Another possibility for reducing domain shift is to optimize the architecture of the network. These adaptation techniques are complementary with the others, hence they can be used in most deep DA models, in both supervised and unsupervised settings. A simple architectural adjustment was suggested by [33]. The intuition is that weights

associated with the source model and the task model are related, but shouldn't be forced to be equal (i.e. shared). For example, we can use two identical models, one for each domain, and add the following regularization term to the loss function:

$$\Omega = \sum_{i=1}^{L} \left(\left\| W_S^{(l)} - W_T^{(l)} \right\|_F^2 + \left\| b_S^{(l)} - b_T^{(l)} \right\|_F^2 \right)$$

where $W_S^{(l)}, b_S^{(l)}$ and $W_T^{(l)}, b_T^{(l)}$ are the parameters of the l^{th} layer in the source and target domain. F denotes the *Frobenius* norm. Another simple idea is revisiting the batch normalization layer in the target model such that each layer receives data from a similar distribution independently of the domain. Formally, batch normalization applies the following transformation:

$$\hat{x}_j = \frac{x_j - \mathbb{E}\left[\mathbf{X}_{.j}\right]}{\sqrt{\mathrm{Var}\left[\mathbf{X}_{.j}\right]}}$$

$$y_j = \gamma_j \hat{x}_j + \beta_j$$

where x and y are the input and output of the layer, X corresponds to the current mini-batch and γ and β are learnable parameters. The scaling of each input data is done for each dimension j, and it guarantees that the input distribution of each layer remains unchanged across different mini-batches. A stable input distribution facilitates the model convergence and accelerates the training process. The authors of [24] suggest that class related knowledge is stored in the weight matrix of each layer, whereas domain related knowledge is represented by the statistics of the batch normalization. To demonstrate this, they used two different (but with same classes) datasets. Then, for each mini-batch sampled from one dataset, they concatenated the mean and variance of all neurons from one layer to form a feature vector. Using a linear SVM, they were able to almost perfectly classify to which domain the feature vector it belongs to. This is an evidence that the batch normalization parameters for each layer are domain related. They proposed thereby AdaBN, which aims to perform domain adaptation by modulating all the batch norm layers' statistics from the source to target domain. fig. 2.9 shows calibration algorithm. Basically, the target data is not used to learn the network weights but only for adjusting the statistics of each batch normalization. Hence, it is still an unsupervised method. Given a pre-trained neural network on the source domain, the adaptive BN algorithm estimates the mean and the variance of each neuron activation using target

Algorithm 1 Adaptive Batch Normalization (AdaBN)

for neuron j in DNN **do**
  Concatenate neuron responses on all images of target domain t: $\mathbf{x}_j = [\ldots, x_j(m), \ldots]$
  Compute the mean and variance of the target domain: $\mu_j^t = \mathbb{E}(\mathbf{x}_j^t)$, $\sigma_j^t = \sqrt{\text{Var}(\mathbf{x}_j^t)}$.
end for
for neuron j in DNN, testing image m in target domain **do**
  Compute BN output $y_j(m) := \gamma_j \frac{\left(x_j(m) - \mu_j^t\right)}{\sigma_j^t} + \beta_j$
end for

Figure 2.9: Batch Normalization adaptation [24]

data only. Then, at test time, the scaling is performed by using the same parameters λ and β learned during training, but instead of using the pre-computed running mean and running variance the source domain, it normalizes with the new adapted parameters. This domain-aware normalization ensures that each layer receives data from a similar distribution, no matter it comes from the source domain or the target domain.

2.2.2 Adversarial-Based Approaches

Thanks to the great success of GANs [15], a lot of researchers took inspiration from their adversarial nature to approach DA. In particular, many models exploit the adversarial losses to maximize domain confusion or to generate real-looking images that can be used to boost the target task. Based on the fact that we may have a generator or not, two sub-categories can be defined: generative models and non generative models.

2.2.2.1 Generative Models

By using GANs, we are able to generate high-quality images. In particular, one can use the source data to generate synthetic yet real looking data similar to the target ones. Moreover, if the newly generated images appear as if they were sampled from the target distribution and preserve the label information of the source images they have been generated on, we can train a classifier in a classical supervised fashion. The big

advantage is that this transformation can be done with no labels. PixelDA [3] does exactly this. An advantage of this strategy is that the DA process is decoupled from the target task, hence it is easily extendable. PixelDA generates target images conditioned on both noise and source images such that the new synthetic data has the style of the target domain and preserves the label of the conditioning image. In this sense the process of DA is decoupled from the objective task. This simple architecture is depicted in fig. 2.10. It is important to note that this model works under the assumption that the gap between the two domains is not too wide: only simple variations such as noise and illumination changes are allowed, while geometric differences are not. Conditioned on both noise and source data, the generator G aims to generate data belonging to the target domain. The Discriminator D is in charge to check whether this is true or not. Next, the model is augmented with a classifier, responsible for classifying both source images and the fake ones that are synthesized. The idea is that even though the fake data should belong to the other domain, it must have the same semantic of the image on which it has been conditioned, hence the classifier T should be able to determine the correct class. This trick facilitates the generation of coupled images. An important detail is that if we only use the fake images as input for T, the generator would still be able to generate images with a correct label associated, but then it may exhibit the shift class problem (i.e. class 0 assigned to class 1, class 2 to class 0, etc.) A slightly different idea is CoGAN [25]. While the previous generative approach fells in the category of the methods that try to learn a mapping from one domain to the other (in pixel space), CoGAN learns a domain-invariant feature space. As the name suggests, a CoGAN consists of a pair of GANs, each one responsible for synthesizing images in the corresponding domain. During training, some layers (see fig. 2.11) are shared. This results in a model capable of mapping the same noise vector into images belonging to two different domains but still preserving some information (i.e. the high level content). Note how also this approach does not rely on coupled images.

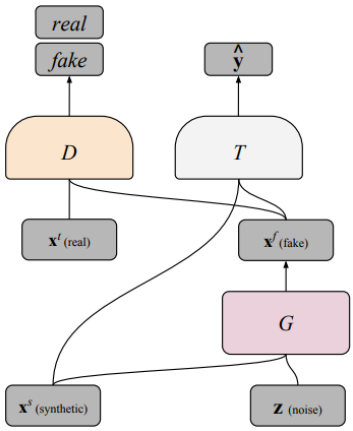


Figure 2.10: PixelDA

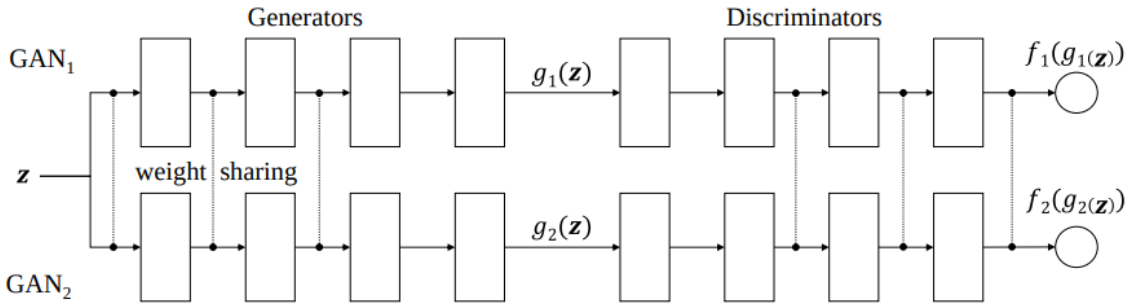


Figure 2.11: CoGAN architecture from [25]

Since the generator maps noise to image, its first layers decode high-level features, while the last layers capture low-level and domain specific details. This is different from what happens in a general CNN or in the discriminator itself, in which the flow is quite the opposite. This behavior suggests that by sharing the first layers, the semantic of the image is preserved, while domain-specific characteristics may change so that the corresponding discriminator is fooled. In addition, this aspect helps in aligning the high level features of the two domains. The weight sharing in the discriminators instead, is not essential for the generation task (even though it is useful for reducing the number of parameters), but it becomes very important when deploying this architecture for DA. In order to do so, a Softamx layer (the classifier) must be attached to the last layer shared discriminator. By embedding the model with such layer, we can jointly train the architecture to solve the classification problem, which uses images and labels form the source domain, and the generation learning problem, which instead utilizes the images from both domains. After training, thanks to weight sharing, the discriminator for the target domain can be used together with the classification layer to predict the class of target data.

2.2.2.2 Non-Generative Models

Differently from the previous approaches, non-generative models exploit the adversarial loss but without the need for a generator. The idea is similar to the one adopted in the first model of subsubsection 2.2.1.1: maximize domain confusion without the need for annotated data in the target domain. This can be done by training a discriminator to decide whether the feature vector comes from the source or target domain.

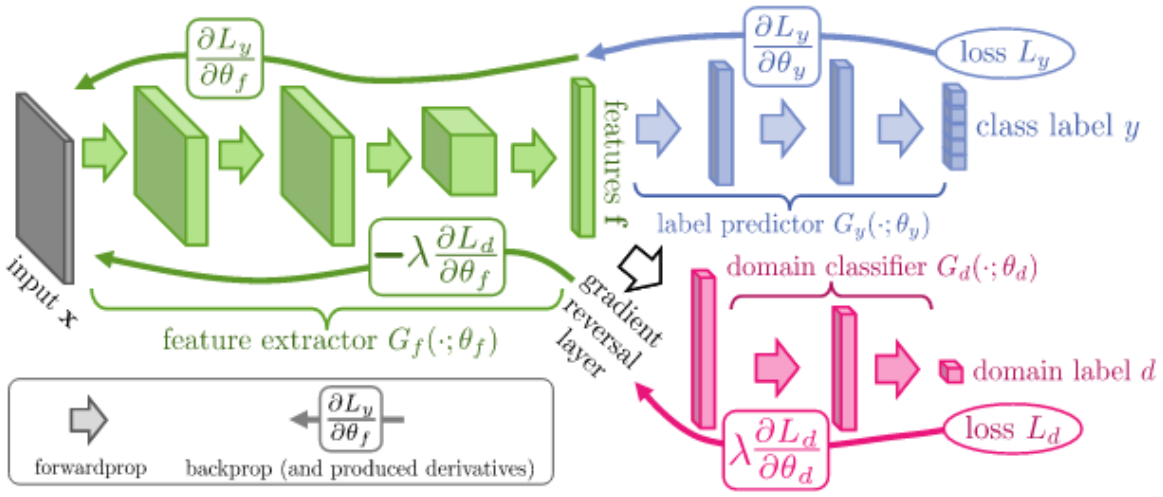


Figure 2.12: Domain Adversarial Neural Network [11]

Being able to fool the discriminator, implies that the model has learned a common and indistinguishable feature space. One of the most popular proposed approaches is the Domain-Adversarial Neural Network [11] illustrated in fig. 2.12.

The architecture consists of a CNN acting as a feature extractor and two attached heads: the label classifier (blue part) and a domain classifier (purple layers). The flow is the following: source data is used to train the main CNN to extract relevant features for the label classifier. The domain classifier is trained jointly to decide whether the features are obtained from the source domain or the target domain. Hence, the purple part aims to minimize domain confusion while the green part tries to maximize it. To accomplish this, a gradient reversal layer is placed between the last layer of the CNN and the first of the domain classifier. The consequence is that during the forward pass nothing changes, i.e. the features are extracted from the input image and passed directly to the domain classifier. Things are different instead in the backward pass. The purple layers are updated as usual to minimize the confusion loss, while the green layers are optimized so that the same loss is maximized, because of the gradient reversal layer that inverts the sign of the back flowing gradient (and amplifies it by a factor of λ). The idea of maximizing domain confusion is used in another popular architecture, called ADDA [36] (Adversarial Discriminative Domain Adaptation). The novelty of this model

relies on the fact that the weights of the backbone CNN are not shared anymore. This gives more flexibility since it allows us to learn more domain-specific aspects, improving thereby performances on the target task. A source CNN is firstly trained with the source domain data with the corresponding labels; in this way we are able to learn discriminative features by simply minimizing cross entropy error:

$$\min_{M_s,C} \mathcal{L}_{\text{cls}}\left(\mathbf{X}_s, Y_s\right) = \quad -\mathbb{E}_{(\mathbf{x}_s,y_s)\sim(\mathbf{X}_s,Y_s)} \sum_{k=1}^{K} \mathbb{1}_{[k=y_0]} \log C\left(M_s\left(\mathbf{x}_s\right)\right)$$

Where C denotes the classifier and M_s is the source CNN. This step can be seen as a pre-training phase. Afterward, an identical CNN, namely the target CNN (M_t), is initialized with the same weights and fine-tuned in an adversarial fashion with the domain discriminator (D).

$$\begin{aligned}\min_{D} \mathcal{L}_{\text{adv}_D}\left(\mathbf{X}_s, \mathbf{X}_t, M_s, M_t\right) = \\ &- \mathbb{E}_{\mathbf{x}_s\sim\mathbf{X}_s}\left[\log D\left(M_s\left(\mathbf{x}_s\right)\right)\right] \\ &- \mathbb{E}_{\mathbf{x}_t\sim\mathbf{X}_t}\left[\log\left(1 - D\left(M_t\left(\mathbf{x}_t\right)\right)\right)\right]\end{aligned}$$

$$\begin{aligned}\min_{M_t} \mathcal{L}_{\text{adv}_M}\left(\mathbf{X}_t, \mathbf{X}_t, D\right) = \\ &- \mathbb{E}_{\mathbf{x}_t\sim\mathbf{X}_t}\left[\log D\left(M_t\left(\mathbf{x}_t\right)\right)\right]\end{aligned}$$

The parameters of the discriminator D are updated so that $\mathcal{L}_{\text{adv}_D}$ is minimized, while M_t is updated to minimize $\mathcal{L}_{\text{adv}_M}$. Note that one could have also defined $\mathcal{L}_{\text{adv}_M} = -\mathcal{L}_{\text{adv}_D}$, that corresponds to minimizing the probability of the discriminator being correct. But this formulation causes the gradient to vanish at the beginning of the training process since the discriminator converges more quickly. This is the step that tries to alleviate the domain shift. It is important to note that the parameters of the source CNN are kept fixed, hence we are pushing the target model to find a similar mapping learned by the source CNN while using data from the target domain. This is sometimes referred to as asymmetric mapping, because we learn a transformation so that it matches the other distribution, rather than trying to learn jointly two similar distributions (symmetric mapping). This behavior reflects the GAN training, in which only one distribution is allowed to change while the other (real images) remains unchanged. The initialization of the target CNN is required because we don't have labels for the target domain (we are

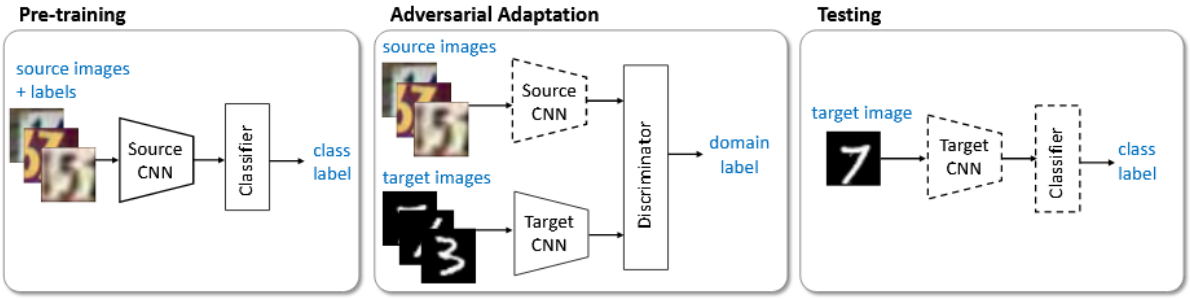


Figure 2.13: Adversarial Discriminative Domain Adaptation [36]

in the unsupervised scenario). If we don't do that, the feature extractor may learn to fool the discriminator, but without being able to capture relevant features for the target task. The hope is that by initializing with the source weights that are good for class discrimination, and by reducing the difference between the two distributions, the target CNN can perform well on the target task. The message suggested by this model is that we don't actually need to learn to generate images belonging to the target domain to be able to capture meaningful features without labels (as done with CoGAN), but we should rather focus on learning discriminative features as they are more useful for the final task.

2.2.3 Reconstruction-Based Approaches

The third and last category that we analyze is reconstruction-based approaches. In this case, the basic idea is to use the reconstruction through an encoder-decoder architecture or a GAN in order to learn useful features that can be used to boost performances for the target task.

2.2.3.1 Encoder-Decoder reconstruction

One of the most simple architectures that uses an encoder-decoder structure is DRCN, short for Deep Reconstruction Classification Network [13]. The whole architecture can be described with few words: a shared encoder receives in input data from both domains; in the case of source data, the labels are used directly to minimize the cross entropy error for classification, while when target images are fed, the decoder tries to

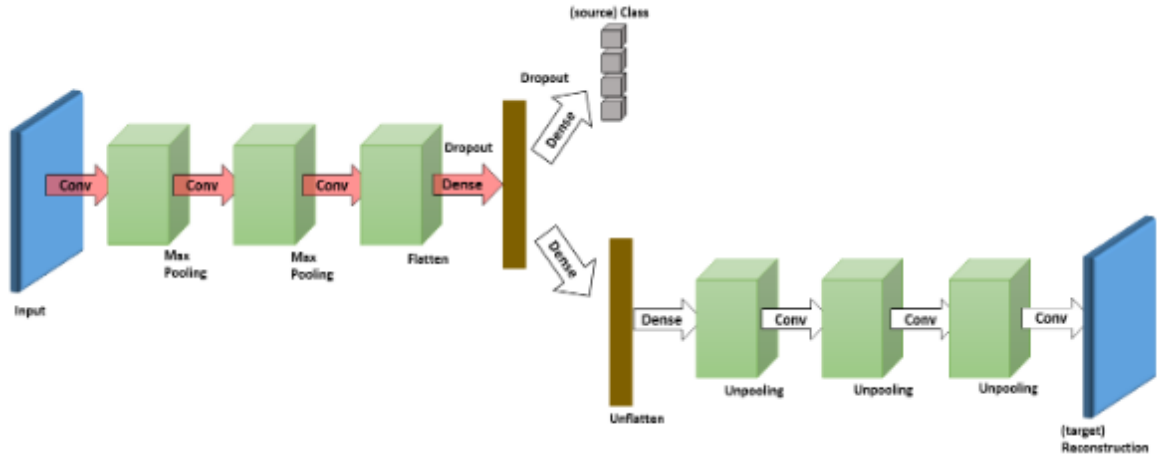


Figure 2.14: Deep Reconstruction Classification Network [13]

reconstruct them. This simple mechanism allows us to capture features relevant to the target domain (thanks to the reconstruction component) but at the same time useful for the classification task. The same author of PixelDA proposed another interesting architecture called DSN [4]. Although the performances are slightly worse compared to PixelDA, it is still worthy to show this encoder-decoder architecture due to the novelty of the approach. The intuition, in this case, consists of learning a shared latent space by forcing the network to use exclusively common characteristics. This is done by using a shared-weight encoder $E_c(x)$ that learns to capture the common representation components among the two domains. Two private encoder $E_p(x)$ are instead responsible of learning to capture domain-specific components. A shared decoder learns then to reconstruct the input samples by using both the private and source features. In order to encourage such orthogonality (the feature learned by the private encoder should not be used by the shared encoder and vice versa), the private and shared representation components are pushed away with a contrasting loss, whereas the shared representation components are kept similar with a similarity loss. A classifier G is finally trained on the shared representation using the available labels for the source domain.

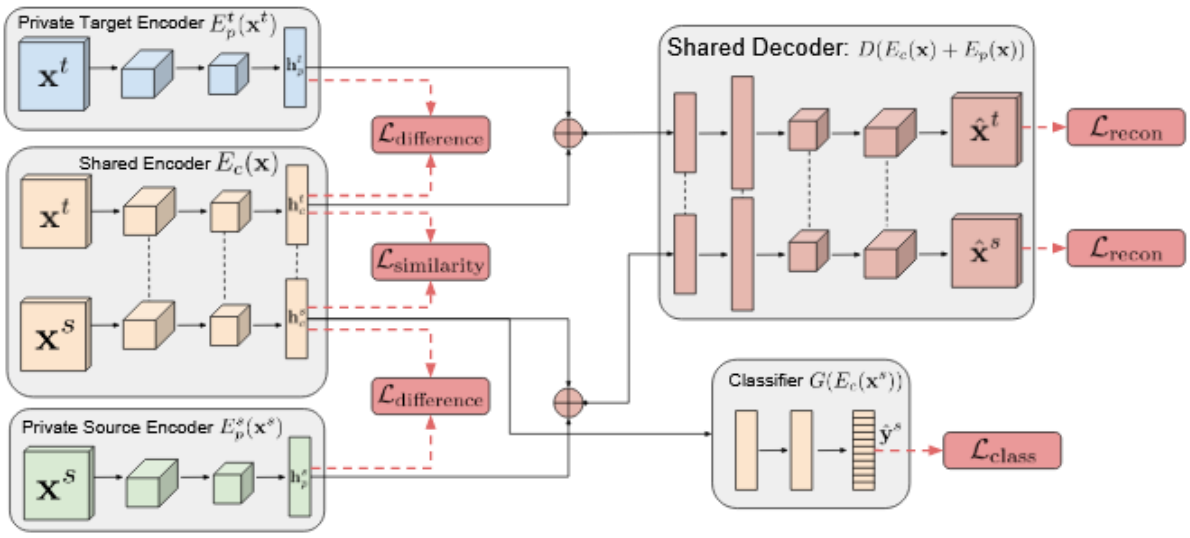


Figure 2.15: Domain Separation Network [4]

2.2.3.2 Adversarial reconstruction

Instead of using the feature learned by an encoder-decoder architecture, one can also use GANs as a proxy task to capture meaningful features for the target domain. An example of such approach is based on the CycleGANs, a variation of the original GANs that can lean to translate images across domains without paired training examples. A framework that directly uses a CycleGAN was introduced in [27]. This model summarizes many of the ideas explained so far and uses many losses trained jointly; for this reason, it may seem a bit complex. It is based on three intuitions:

1. Good features that can work on different domains should be domain agnostic (as suggested by DSN). This can be motivated by the following example. Let us say that we are working to build a system able to solve the semantic segmentation task for autonomous vehicles. At our disposal, we have a dataset containing street scenes in sunny days and a dataset in which are collected street scenes again but only in rainy days. Of course, the system should be able to correctly classify pixels belonging to the class *street*, independently from the weather conditions. We can thereby think of the weather as domain-specific information, whereas the street as domain agnostic content. By using this kinds of features, we are able to learn a

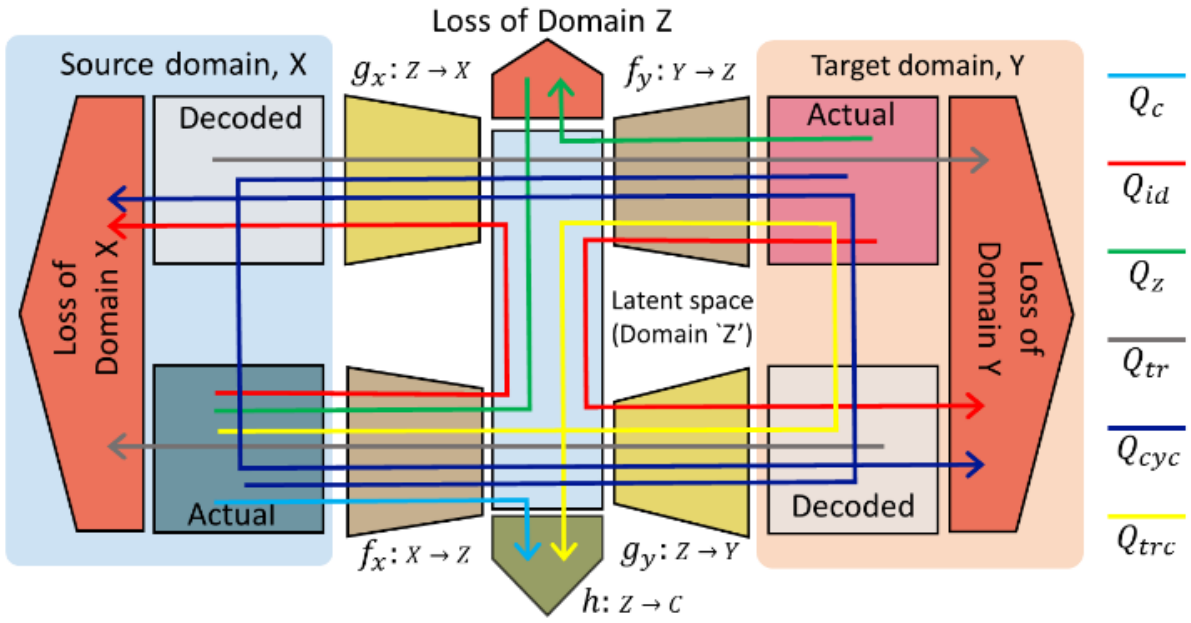


Figure 2.16: Domain adaptation through CycleGANs [27]

common feature space that can be adapted to different domains.

2. If the feature space is general enough to be shared among domains, we should be able to map images belonging to one domain into feature space, and then convert them into an image belonging to the other domain.

3. Finding a common feature space is not enough, because we also have to make sure that similar classes among domains lie close, while different classes should be far in latent space. This can be done through a cycle consistency loss, or through a semantic loss as done with PixelDA, where both the translated images and the original source image should share the same label. Another option is to do this by means of statistical methods as seen in section 2.2.1.2, although they are more difficult to apply in the semantic segmentation case.

Again, these ideas have been already used on several strategies, but here they are proposed altogether. fig. 2.16 summarize the model. The whole model is based on several components. Two encoders, f_x and f_y, learn a mapping from source and target domain to a common feature space denoted with Z. The objective is to learn a classifier

$h : Z \rightarrow C$, with C representing the label space. First, we need to make sure that some discriminative features are learned. As usual, this is done using the annotated source data:

$$Q_c = \sum_i l_c \left(h \left(f_x \left(x_i \right) \right), c_i \right)$$

l_c denotes a suitable classification loss as cross entropy. To learn a common feature space Z for both domains, several losses and architectures are applied. Following the intuitions listed before, Z should retain only domain agnostic aspects. In order to accomplish this, two decoders g_x and g_y learn to generate images for source and target domain respectively starting from Z. The idea is that f_x and f_y should learn to remove private domain information, while g_x and g_y should add back this content. Basically, by applying first f_x and then g_x, we are obtaining the identity function:

$$\begin{aligned} Q_{id} = & \sum_i l_{id} \left(g_x \left(f_x \left(x_i \right) \right), x_i \right) + \\ & \sum_j l_{id} \left(g_y \left(f_y \left(y_j \right) \right), y_j \right) \end{aligned}$$

l_{id} is any pixel-wise loss such as L_2. Afterward, a discriminator d_z tries the recognize the starting domain given a feature vector. This is again the idea of using a domain confusion loss:

$$\begin{aligned} Q_z = & \sum_i l_\alpha \left(d_z \left(f_x \left(x_i \right) \right), c_x \right) + \\ & \sum_j l_a \left(d_z \left(f_y \left(y_j \right) \right), c_y \right) \end{aligned}$$

l_a is an appropriate loss for classification such as binary cross entropy, while c_x and c_y are binary labels (i.e. 0 for source domain and 1 for target domain). The discriminator is trained to minimize domain confusion and consequently Q_z. On the contrary, f_x and f_y are updated so that it is maximized. To further ensure this common representation and avoid the typical problem of model collapse, they also define a translation adversarial loss that mimics the image-to-image translation task. The idea is that it should be possible to first map an image belonging to one domain to the latent space Z and then decode it back to the other domain to generate a 'fake' (translated) image. Two discriminators $d_x : X \rightarrow c_x, c_y$ and $d_y : Y \rightarrow c_x, c_y$ are trained to determine whether the fake translated

images are real or not. This ensures that different images can be generated from the latent space and consequently the model does not collapse:

$$Q_{tr} = \sum_i l_a \left(d_y \left(g_y \left(f_x \left(x_i \right) \right) \right), c_x \right) + \sum_j l_a \left(d_x \left(g_x \left(f_y \left(y_j \right) \right), c_y \right) \right)$$

As suggested by the last intuition, we need to ensure that the semantically similar images in both domains are projected into close vicinity in the latent space. To force this behavior, a cycle consistency loss is added as in the classical architecture of a CycleGAN:

$$Q_{cyc} = \sum_i l_{id} \left(g_x \left(f_y \left(g_y \left(f_x \left(x_i \right) \right) \right) \right), x_i \right) + \sum_j l_{id} \left(g_y \left(f_x \left(g_x \left(f_y \left(y_j \right) \right) \right) \right), y_j \right)$$

A final trick that allows the target encoder to be trained to capture discriminative features useful for the target task (this is after all our initial goal) is to map into Z a source image, translate into the target domain space, map it back to z and then classify it with the original source label:

$$Q_{trc} = \sum_i l_c \left(h \left(f_y \left(g_y \left(f_x \left(x_i \right) \right) \right) \right), c_i \right)$$

Finally, the loss is a weighted average of all the previous:

$$Q = \lambda_c Q_c + \lambda_z Q_z + \lambda_{tr} Q_{tr} + \lambda_{id} Q_{id} + \lambda_{cyc} Q_{cyc} + \lambda_{trc} Q_{trc}$$

As explained before, this framework can be seen as a generalization of some of the

Method	λ_c	λ_z	λ_{tr}	λ_{id_A}	λ_{id_B}	λ_{cyc}	λ_{trc}
ADDA [36]	✓	✓					
DRCN [13]	✓				✓		
I2I	✓	✓	✓	✓	✓	✓	✓

Table 2.1: Possible instances of Image to Image adaptation

previously presented techniques. table 2.1 highlights some of the models that can be

obtained as a particular instance of this. By training only f_x on the source domain and then freezing it and training the target encoder f_y and setting $\lambda_{id} = \lambda_{cyc} = \lambda_{tr} = 0$ we obtain [36]. By setting instead $\lambda_{id_A} = \lambda_{cyc} = \lambda_{tr} = \lambda_z = 0$ we recover [13]. λ_{id_A} denotes the first term of Q_{id}. Another popular and successful method that uses cycle consistency is CyCADA [18]. The approach is very similar to the one just exposed, but in addition it adapts representations at both the pixel-level and feature-level. The key point of CyCADA is that alignment at higher levels of a deep representation can fail to model aspects of low-level details, which are instead crucial for many visual tasks. Also in PixelDA there was this intuition, but here adaptation is performed at both level, rather than pixel-level only. Moreover, CyCADA uses cycle consistency together with a semantic loss to force even more the network to preserve the content of an image during translation. The first step is to solve the source task simply using the source annotated data and minimizing the cross entropy error as usually done for classification:

$$\mathcal{L}_{\text{task}}\left(f_S, X_S, Y_S\right) = -\mathbb{E}_{(x_s,y_s)\sim(X_S,Y_S)} \sum_{k=1}^{K} \mathbb{1}_{[k=y_s]} \log\left(\sigma\left(f_S^{(k)}\left(x_s\right)\right)\right)$$

f_S represents the source encoder, while σ is a softmax layer. After this pre-training step, several losses are applied to compensate the domain shift. First the image-to-image translation loss is optimized by using two generators $G_{S\to T}$ and $G_{T\to S}$ (one for each direction). For simplicity we report only the loss in one direction, from source to target:

$$\mathcal{L}_{\text{GAN}}\left(G_{S\to T}, D_T, X_T, X_S\right) = \mathbb{E}_{x_t\sim X_T}\left[\log D_T\left(x_t\right)\right] + \mathbb{E}_{x_s\sim X_S}\left[\log\left(1 - D_T\left(G_{S\to T}\left(x_s\right)\right)\right)\right]$$

This GAN loss is the one that performs adaptation by mapping one domain to the other at a pixel-level (green portion in fig. 2.18). By doing this we can also train the target encoder f_t, that is the real objective of the whole model, on the fake target images generated from source images (purple portion): $\mathcal{L}_{\text{task}}\left(f_T, G_{S\to T}\left(\bar{X}_S\right), Y_S\right)$. Note that the labels of the source images are used, hence it is fundamental to respect the semantic of the source image while translating domain. Cycle consistency loss and a semantic loss are both added to this purpose:

$$\begin{aligned}\mathcal{L}_{\text{cyc}}\left(G_{S\to T}, G_{T\to S}, X_S, X_T\right) = {} & \mathbb{E}_{x_s\sim X_S}\left[\left\|G_{T\to S}\left(G_{S\to T}\left(x_s\right)\right) - x_s\right\|_1\right] \\ & + \mathbb{E}_{x_t\sim X_T}\left[\left\|G_{S\to T}\left(G_{T\to S}\left(x_t\right)\right) - x_t\right\|_1\right]\end{aligned}$$

$$\mathcal{L}_{\text{sem}}\left(G_{S\to T}, G_{T\to S}, X_S, X_T, f_S\right) = \mathcal{L}_{\text{task}}\left(f_S, G_{T\to S}\left(X_T\right), p\left(f_S, X_T\right)\right) + \mathcal{L}_{\text{task}}\left(f_S, G_{S\to T}\left(X_S\right), p\left(f_S, X_S\right)\right)$$

$\mathcal{L}_{\text{cyc}}$ ensures that $G_{T\to S}\left(G_{S\to T}\left(x_s\right)\right) \approx x_s$ and $G_{S\to T}\left(G_{T\to S}\left(x_t\right)\right) \approx x_t$, while $\mathcal{L}_{\text{sem}}$ forces the same classification before and after the translation. The classification is done with the encoder f_s that was pre-trained on the previous step with source data only (f_s is freezed). $p(f_S, X_T)$ and $p(f_S, X_S)$ denote the predictions obtained with f_s on the target and source sample respectively. It is important to note that these are just noisy labels, although they are still sufficient to maintain content information after translation. The authors motivated the choice of using both these losses with an ablation study using SVHN [28] as source domain and MNIST [23] as tareget domain. In particular, they showed that without the semantic loss, both the GAN and cycle constraints are satisfied (image generation and image reconstruction), but the semantic content is lost during translation. This behaviour is depicted in fig. 2.17(a), where the translated digits do not correspond to the original one. On the other hand, without cycle loss, the reconstruction fails and the semantic consistency alone is successful only in some cases.

(a) Without Semantic Loss (b) Without Cycle Loss

Figure 2.17: Ablation study on the effect of the semantic and cycle consistency loss [18]

A third and fine-tuning step is done to ensure domain adaptation at the feature-level (orange portion): $\mathcal{L}_{\text{GAN}}\left(f_T, D_{\text{feat}}, f_S\left(G_{S\to T}\left(X_S\right)\right), X_T\right)$. Overall, the full loss function

is

$$\begin{aligned}
&\mathcal{L}_{\text{CyCADA}}\left(f_T, X_S, X_T, Y_S, G_{S\to T}, G_{T\to S}, D_S, D_T\right) \\
&= \mathcal{L}_{\text{task}}\ \left(f_T, G_{S\to T}\left(X_S\right), Y_S\right) \\
&\quad +\mathcal{L}_{\text{GAN}}\left(G_{S\to T}, D_T, X_T, X_S\right) + \mathcal{L}_{\text{GAN}}\left(G_{T\to S}, D_S, X_S, X_T\right) \\
&\quad +\mathcal{L}_{\text{GAN}}\left(f_T, D_{\text{feat}}\ , f_S\left(G_{S\to T}\left(X_S\right)\right), X_T\right) \\
&\quad +\mathcal{L}_{\text{cyc}}\ \left(G_{S\to T}, G_{T\to S}, X_S, X_T\right) + \mathcal{L}_{\text{sem}}\left(G_{S\to T}, G_{T\to S}, X_S, X_T, f_S\right)
\end{aligned}$$

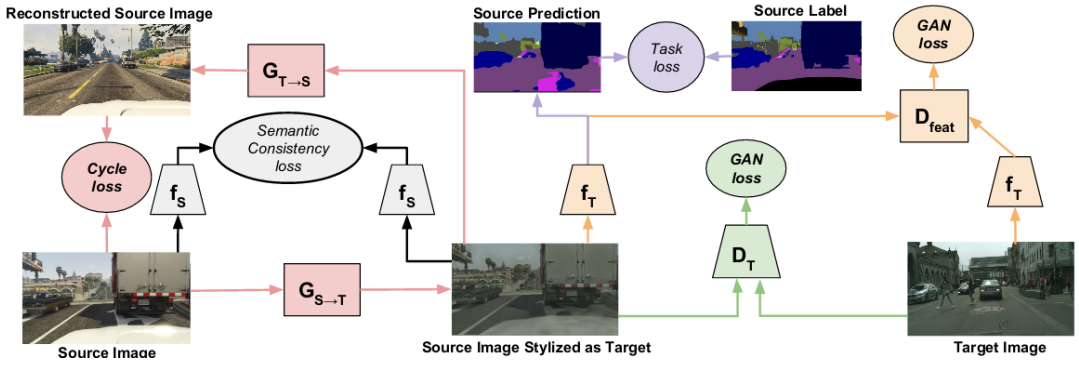
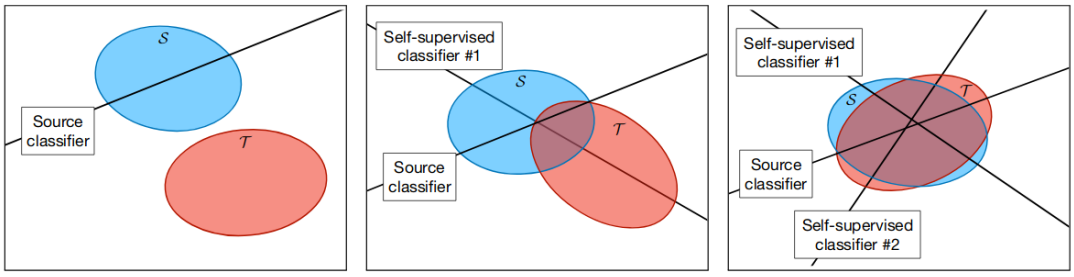


Figure 2.18: Cycle-consistent adversarial adaptation architecture [18]. Target cycle omitted

2.2.4 Self-supervised approaches for Domain Adaptation

The recent success of Self-supervised learning (or equivalently Weak-supervised) that makes use of auxiliary tasks to boost classification performances, has also gained attention in the Domain Adaptation setting. The great advantages of such tasks (i.e. the possibility to obtain free labels for abundant data), make them particularly attractive in situations in which data is only limited or not available at all as in the case of UDA. Although the potential benefit of Self-supervised tasks has not been thoroughly explored yet, some recent works try to exploit tasks such as rotation prediction, colorization and the jigsaw puzzle to reduce domain shift. On this line [40], seeks to align the two distributions by relying on weak-supervision. Self-supervised learning allows to generate synthetic labels automatically by applying simple transformations on the original dataset. The idea of this work is to accomplish the alignment by solving auxiliary

weak-supervised task(s) on both domains jointly with the main downstream task. Each self-supervised task brings the two domains closer along the direction relevant to that task:



(a) Source classifier only (b) adding a self-supervised task (c) adding more tasks

Figure 2.19: Alignment through weak supervision [40]

Both the architecture and the optimization process are simpler than the previous methods, as no GANs and discriminative losses are involved. A shared feature encoder ϕ learns to map images from different domains to a common feature space. The extracted features are then used by a classifier that solves the source task (optimized using only source images). Moreover, these features are also used by several heads, each one associated to its corresponding proxy task. These heads are composed of just one single layer so that only the features provided by the encoder are used. In this case, the optimization is performed with batches containing images from both domains. The architecture is summarized in fig. 2.20. By denoting with $\mathcal{L}_0$ the loss function computed by the first head (h_0) that solves the source task, and with $\mathcal{L}_k$ the objective function for kth auxiliary task, we can formalize the whole objective as follows:

$$\min_{\phi, h_k, k=1\ldots K} \mathcal{L}_0\left(s; \phi, h_0\right) + \sum_{k=1}^{K} \mathcal{L}_k\left(s, t; \phi, h_k\right)$$

The proxy tasks should be carefully selected. The purpose of solving them is to learn general features that are shared across domains. Indeed, if we select a task that focuses too much on low-level details, we may have the risk to separate even more the two domains rather than aligning them, since we are not capturing the high-level and shared features. Particularly unsuitable tasks for unsupervised domain adaptation are for example colorization and denoising autoencoder, while examples of tasks that can

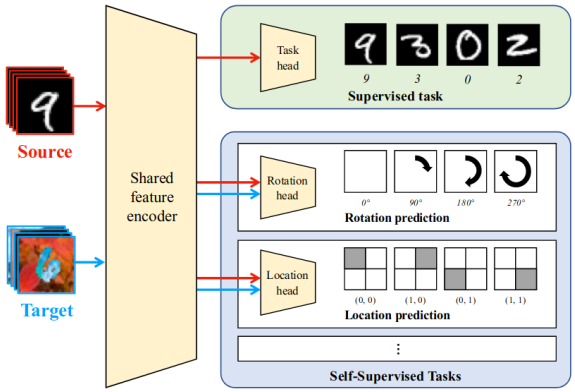


Figure 2.20: Weak supervised learning for Domain Adaptation [40]

be used successfully are predicting the degree of a random rotation or flip prediction. One may even apply domain knowledge to design a custom auxiliary task and boost the domain adaptation process. A similar idea that deploys rotation as pretext task has also been used in [39]. In this case, the architecture is enhanced with adversarial training for domain alignment in feature space rather than using multiple auxiliary tasks as before (see fig. 2.21) and batch normalization calibration, that consists in recomputing the statistics of each batch normalization, which is very similar to AdaBN, but instead of recomputing the statistics of each BN layer with a specific algorithm, it directly updates the common moving average and variance by feeding once all the target images.

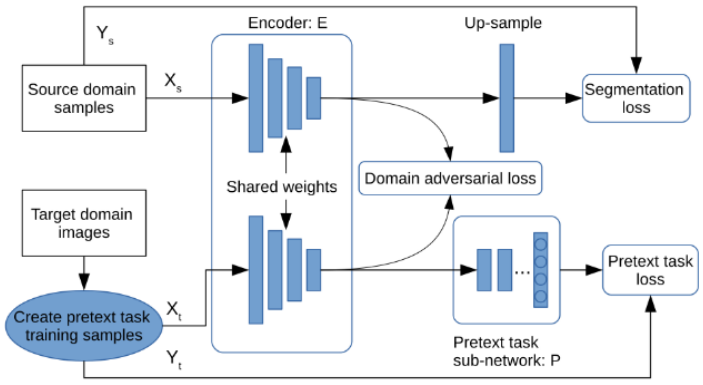


Figure 2.21: Weak supervised learning with adversarial training for Domain Adaptation [39]

2.3 Performances analysis

Due to the different settings in which all the previous method are tested is quite difficult to provide an objective evaluation. For instance, one model can be carefully fine-tuned and achieve a higher score even though in practice its performances are worse compared to another strategy. Moreover, one method can outperform the other in one specific task, while vice-versa the situation can be reversed for a different one. Another important aspect is the base architecture: some of the presented domain adaptation techniques make use of powerful backbones such as ResNet or Inception, while others just use a simple fully connected neural network. We thereby just report the official results reported on several datasets, highlighting which are the most successful methods, even though these may not be the best ones overall. Some hints are also given to explain why some architectures work or not in particular scenarios. To compare the DA algorithms explained in previous sections, we rely on two standard benchmarks:

- The Office [34] dataset includes 4652 images of the same 31 objects collected from three different domains: 2,817 images from the Amazon website, 498 high-resolution images taken with a DSRL camera and 795 low-resolution pictures taken by a web camera. Any DA method can be tested in all six pairs of domains: $A \rightarrow W$, $D \rightarrow W$, $W \rightarrow D$, $A \rightarrow D$, $D \rightarrow A$, $W \rightarrow A$.

- Digits recognition is another popular benchmark for evaluating DA. Typical datasets are in this case MNIST, SVHN, and USPS [8]. The MNIST includes 60000 training pictures and 10000 test pictures. The USPS includes 7291 training pictures and 2007 test pictures. Finally, SVHN has 73257 digits in the training set and 26032 digits for testing. Although this can be seen as a simple adaptation process at a first glance, in some cases this is not true. For instance, SVHN contains significant variations (in scale, background color, etc..) while MNIST contains only gray-scale images. For this reason, adaptation performed from MNIST to SVHN is quite a difficult task. On the other hand, as table 2.4 confirms, reducing the domain shift between MNIST and USPS in much more easier.

Let us start firstly by summarizing all the previous models by indicating whether the method is trying to find a domain invariant feature space or is learning a mapping

	Method	Generator	Adversarial Loss	Share weights	Setting
Soft Labels [35]	DI	no	feature-level	yes	Supervised
DTML [19]	DI	no	no	yes	Unsupervised
DaNN [35]	DI	no	no	yes	Unsupervised
DDC [37]	DI	no	no	yes	Unsupervised
Rozantsev et al [33]	DI	no	no	no	Unsupervised
AdaBN [24]	N	no	no	no	Unsupervised
PixelDA [3]	DM	yes	pixel-level	yes	Unsupervised
CoGAN [25]	DI	yes	pixel-level	partially	Unsupervised
DANN [11]	DI	no	feature-level	yes	Unsupervised
ADDA [36]	DI	no	feature-level	partially	Unsupervised
DRCN [13]	DI	no	feature-level	yes	Unsupervised
DSN [4]	DI	no	feature-level	partially	Unsupervised
I2I [27]	DI, DM	yes	feature-level	partially	Unsupervised
CyCADA [18]	DI, DM	yes	feature-level and pixel-level	no	Unsupervised
Sun et al[40]	DI	no	no	yes	Unsupervised
CAN [21]	DI	no	no	no BN	Unsupervised

Table 2.2: Categorization of several DA methods

between domains, if a generator is required, if the losses are adversarial based or not and finally whether some weights are shared or not. As can be seen from table 2.2, most of the recently proposed methods work under the unsupervised setting. This highlights the importance and the attention of the research community towards unlabeled data. It is also clear that most of the ideas can be combined together to obtain architectures even more robust to domain shift. For example, recent works use adversarial losses both at the feature-level and pixel-level, or they perform domain mapping while trying to find a domain invariant feature alignment. table 2.3 summarizes the results of various methods on the Office dataset. In this case, by learning a domain-invariant feature space and minimizing a contrastive domain discrepancy loss, CAN seems to be the clear winner since it outperforms all the other DA techniques in all settings. However, we should keep in mind that CAN uses ResNet50 as backbone, while most of the approaches in table 2.3 use less powerful networks. These outstanding results are probably due to the intrinsic nature of the contrastive loss, that is able to enhance the model's generalization ability for classification problems by performing class-aware alignment across domains.

DA	$A \rightarrow W$	$D \rightarrow W$	$W \rightarrow D$	$A \rightarrow D$	$D \rightarrow A$	$W \rightarrow A$
No DA[a]	62.6	96.1	98.6			
DDC [37]	59.4	92.5	91.7			
AdaBN[c] [24]	74.2	95.7	99.8	73.1	59.8	57.5
Soft Labels [35]	59.3	90.0	97.5	68.0	43.1	40.5
I2IA[b] [27]	75.3	96.5	99.6	71.1	50.1	52.1
DRCN [13]	68.7	96.4	99.0	66.8	56.0	54.9
DANN[a] [11]	72.6	96.4	99.2	67.1	54.5	57.7
ADDA[a] [36]	75.1	97.0	99.6			
Xu *et al.*[a] [39]	90.1	98.1	100.0	88.6	65.1	65.0
CAN[a] [21]	94.5	99.1	99.8	95.0	78.0	77.0

Table 2.3: Domain Adaptation on the Office dataset
(a) with ResNet50
(b) with ResNet34
(c) with Inception

The gain introduced with this kind of discriminative loss is clearly visible where other methods fail to generalize (e.g. $A \rightarrow D$, $D \rightarrow A$, $W \rightarrow A$). For other scenarios in which adaptation in slightly easier, even a simple approach such as the one based on rotation prediction as an auxiliary task (Xu *et al.*) is able to improve consistently the baseline. It is also important to note that some of these architectures are complementary. For example, one can use soft labels to fine-tune ADDA to boost performances on the target domain. CAN indeed suggests that using somehow proxy labels can be helpful. Regarding digits classification, there is no clear winner. There are however methods that outperform the ones listed in table 2.4, but they are not reported since they often require problem-specific data augmentation or hyperparameter tuning. As evidenced by table 2.4, in scenarios in which the two domains are not too far (i.e. digits classification) adversarial methods that use a generator to perform DA at pixel-level such as I2IA, PixelDA, COGAN and CyCADA seem to be effective. Again, for digit classification, even a simple method that exploits rotation as an auxiliary task is able to outperform more complex methods, at least for $M \rightarrow U$. This may not be true for tasks that require

	MNIST/USPS		SVHN/MNIST	
	$M \to U$	$U \to M$	$S \to M$	$M \to S$
COGAN [25]	91.2	89.1		
I2I[b] [27]	92.1	87.2	80.3	
PixelDA [3]	95.9			
DTML [19]	81.1		71.1	
DSN [4]	91.3		82.7	
CyCADA [18]	95.6	96.5	90.4	
DRCN [13]	91.8	73.6	81.9	40.1
Sun[c][40]	96.5	90.2	85.8	61.3
ADDA[a] [36]	89.4	90.1	76.0	

Table 2.4: Domain Adaptation on digits datasets
(a) with ResNet50
(b) with ResNet34
(c) with ResNet26

a pixel-level understanding as in semantic segmentation. The result of CyCADA in $S \to M$ advises that when mapping from a simpler dataset to a complex one pixel-level adaption may help in reducing domain shift.

Chapter 3

Learning Features across Tasks and Domains: AT/DT

After a broad overview of popular DA methods, we are ready to provide a plausible answer to the initial question we posed: is it possible to learn features from different domains and tasks simultaneously? All the previous techniques are suitable for learning features across different domains, but would it be helpful as well embedding these methods with features coming from different tasks? Recent works have proven that many relevant visual tasks are closely related one to another. A recent study referred to as *Taskonomy* [44], provides useful insights on this topic and brings to light the existence of a structure among visual tasks. For example, surface normals and depth estimation are related tasks, and solving the former can be directly useful for solving the latter. In general, a complete understanding of correlations between tasks is a valuable thing, since it allows to reduce the need for supervised training in many scenarios: Domain Adaptation, Unsupervised Learning, Self-supervised Learning, Multi-task learning, etc... Yet, only few methods are aware of these relationships and exploit this underlying structure. A novel adaptation framework that aims to do so is AT/DT [30] (Across Tasks Domain Transfer). This general framework is able to transfer features learned across tasks within a source domain in a supervised fashion, and then apply this mapping to a target domain, where only supervision is available. The key point is that the source domain can be synthetic, hence easy and cheap to generate, even with labels for complex tasks. This

framework has proven to be effective on two challenging tasks, (i.e. monocular depth estimation and semantic segmentation) and four different domains (Synthia [20], Carla [9], Kitti, and Cityscapes), although performances are not sensational. The objective of this thesis is thereby to study in detail this framework, and consequently extend the original architecture to improve performances. Although one of the strengths of AT/DT is its generality, we will focus on learning features useful for the semantic segmentation problem, starting form monocular depth estimation. This decision reflects the great attention that the research community has towards unsupervised or semi-supervised techniques able to solve semantic segmentation. In the next sections, a thorough explanation of AT/DT is provided, while the next chapter is dedicated to the proposed extensions. Finally, some effort has been done to replace monocular depth estimation with popular Self-supervised tasks.

3.1 Setting

As we already know, DA is a particular case of Transfer Learning. Since AT/DT involves both domain and task adaptation, it can be collocated in the intersection of the two categories. The setting in which this framework works is composed by two domains, $\mathcal{A}$ and $\mathcal{B}$, and two tasks, $\mathcal{T}_1$ and $\mathcal{T}_2$. For simplicity, we can fix $\mathcal{A}$ as the synthetic domain (i.e. Carla) and $\mathcal{B}$ as Cityscapes. The Carla dataset is slightly different from the one used in the original work. Again it has been generated thanks to the Carla simulator, but some parameters in the simulation have been changed to reduce the gap between synthetic and real scenes. As the original one, it contains 3500 training images in total and 500 scenes for evaluation. The Cityscapes dataset contains instead 2975 and 500 images for training and evaluation respectively. In both datasets, the image resolution is 1024×2048. From now on, $\mathcal{T}_1$ will be interpreted as monocular depth estimation, that is a regression problem where for each pixel we must estimate the distance from the camera, while $\mathcal{T}_2$ is set to be semantic segmentation, in which the objective is to classify each pixel in one of the possible category (*street*, *pedestrian*, *vehicle*, etc..). Again this is not the only possible configuration: tasks can be switched and domains can vary. It is intuitively clear though, that these two tasks are correlated. First of all, for both of them,

a complete pixel-wise understanding of the scene is required. Second, depth information can be extremely helpful for detecting the category of an object. For example, it is likely that a point that is far from the camera belongs to the *sky* category. Moreover, we assume to have complete supervision in the source domain, while only partial supervision is available for the target one. Again, thinking about our setting, it means that labels for both depth estimation and semantic segmentation are available in Carla, while only depth maps can be used in Cityscapes. One may question the utility of this framework since obtaining pixel-wise depth maps for real street images is already a complex problem by itself. The answer is that in the case of depth estimation, noisy labels can be obtained by means of off-the-shelf algorithms. For example, depth maps can be obtained by filtering SGM [17] disparities through confidence measures (left-right check). Although these algorithms are far from being optimal, they can still provide useful information from which the model can learn insightful features.

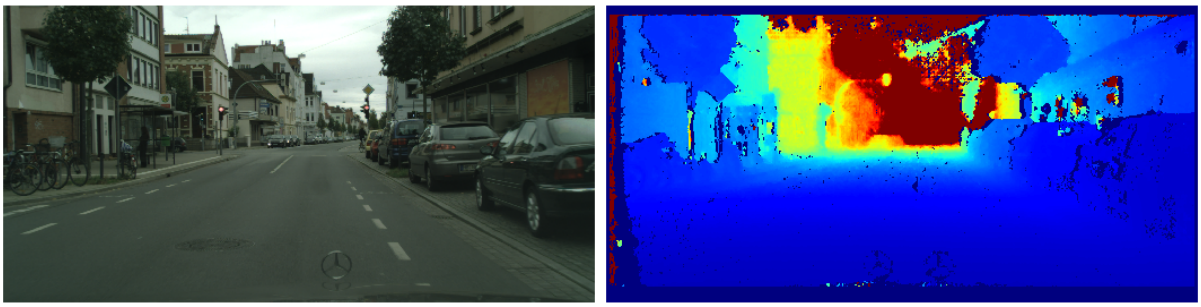


Figure 3.1: Cityscapes training data. Left side RGB input, right side proxy label.

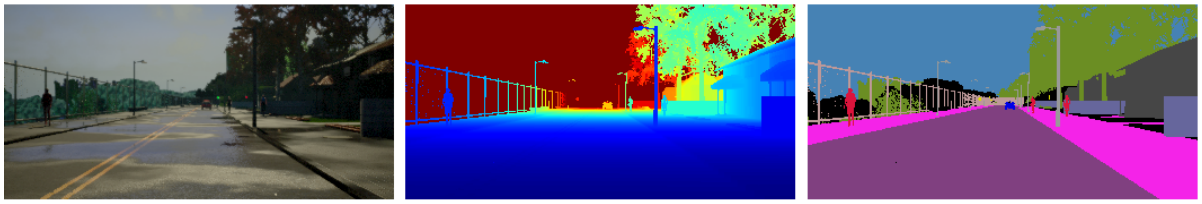


Figure 3.2: Carla dataset. For each image both semantic depth map (middle) and segmentation map (right) are given.

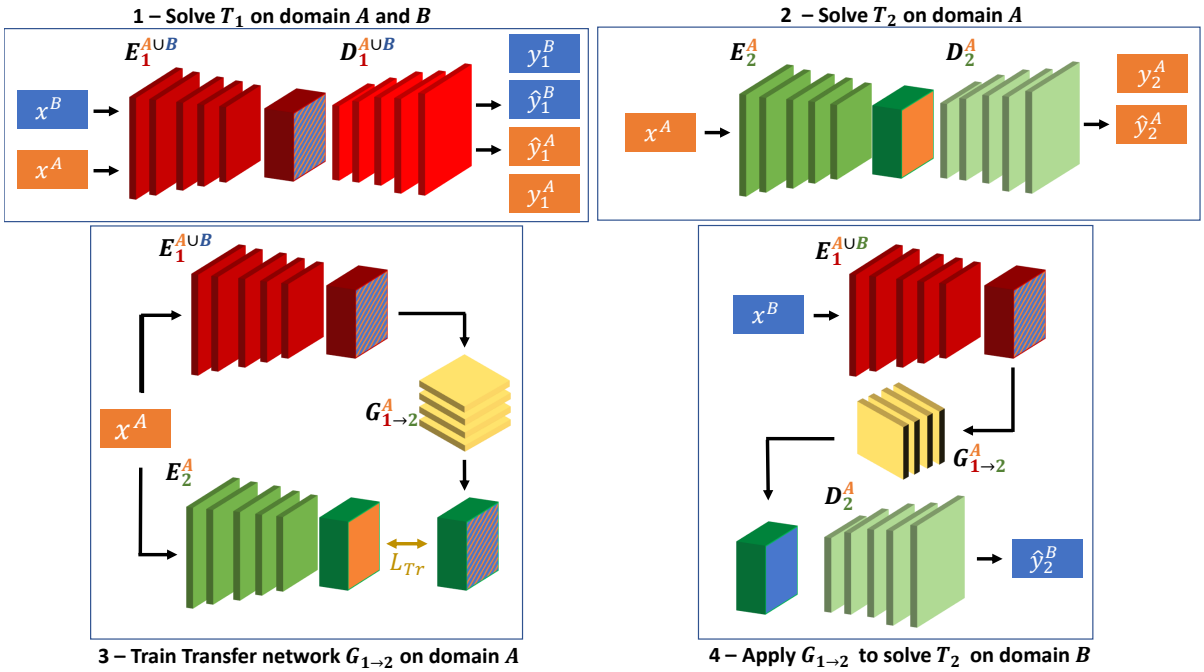


Figure 3.3: AT/DT framework [30]

3.2 Architecture

The whole idea consists in learning a mapping function $G_{1\to2}$ (colored in yellow in fig. 3.3) in feature space between two tasks in a given domain, so that the same mapping can be applied as it is in another domain. More precisely, the architecture foresees a classical encoder-decoder architecture that is used to solve independently $\mathcal{T}_1$ and $\mathcal{T}_2$. These two networks are referred as to $N_1 = D_1(E_1(x))$ (red network in fig. 3.3) and $N_2 = D_2(E_2(x))$ (green network in fig. 3.3) respectively. Since we assumed to have complete supervision for $\mathcal{T}_1$, N_1 is trained with images belonging to both domains. N_2 is of course only trained with synthetic images since we don't have labels for $\mathcal{B}$ in $\mathcal{T}_2$. Up to this point, we obtained an encoder $E_1(x)$ capable of extracting depth features given both real and synthetics images, and an encoder $E_2(x)$ able of encoding deep semantic features. The final step is thereby to train the transfer network $G_{1\to2}$ to map depth features into semantic segmentation features: $G_{1\to2} : E_1(x) \to E_2(x)$. Considering that N_2 is trained on $\mathcal{B}$, and due to the domain shift, $E_2(x)$ can only works reasonably well on the domain it has been trained on. Hence, $G_{1\to2}$ is optimized on $\mathcal{B}$ as well. To solve

$\mathcal{T}_2$, we can now extract depth features from a natural image, convert them in features for the downstream task and feed them to the corresponding decoder. The whole protocol can be summarized in the following steps:

1. Learn to solve task $\mathcal{T}_1$ on domains $\mathcal{A}$ and $\mathcal{B}$.
2. Learn to solve task $\mathcal{T}_2$ on domain $\mathcal{A}$.
3. Train $G_{1\to2}$ on domain $\mathcal{A}$.
4. Apply $G_{1\to2}$ to solve $\mathcal{T}_2$ on domain $\mathcal{B}$.

In the original version, each encoder is a dilated ResNet50 [42] that shrinks the input image by 1/16. The decoder is implemented as a stack of bilinear up-sample and convolutional layers to return to the original resolution and get the final prediction map. The same backbone architecture can be shared among tasks thanks to their similarity. The only difference is in fact the final prediction layer, which is task dependent. The transfer network ($G_{1\to2}$) is instead a simple encoder-decoder architecture that reduces the input feature map to 1/4 of the original resolution before getting back to the original scale. An important detail is that although $G_{1\to2}$ is the simplest among the networks, it is also the heavier in terms of memory requirements and parameters. N_1 and N_2 have less than 30M parameters, while $G_{1\to2}$ requires about 226M parameters. The reason lies in the high number of channels in which each encoder encodes the input image: 2048. Therefore, even a small architecture composed of 6 convolutional layers with kernel size 3×3 such as the transfer network, requires a lot of parameters. The impact of the number of channels in which $G_{1\to2}$ operates will also be experimented and reported in the following chapter. fig. 3.4 illustrates in detail the architecture of the decoder, while fig. 3.5 shows $G_{1\to2}$. The encoder is omitted due to its length.

```
Layer (type)                 Output Shape              Param #
=================================================================
InputLayer                   [(None, 32, 32, 2048)]    0
UpSampling2D                 (None, 64, 64, 2048)      0
Conv2D                       (None, 64, 64, 128)       2359424
BatchNormalization           (None, 64, 64, 128)       512
Activation(Elu)              (None, 64, 64, 128)       0
Conv2D                       (None, 64, 64, 128)       147584
BatchNormalization           (None, 64, 64, 128)       512
Activation(Elu)              (None, 64, 64, 128)       0
UpSampling2D                 (None, 128, 128, 128)     0
Conv2D                       (None, 128, 128, 64)      73792
BatchNormalization           (None, 128, 128, 64)      256
Activation(Elu)              (None, 128, 128, 64)      0
Conv2D                       (None, 128, 128, 64)      36928
BatchNormalization           (None, 128, 128, 64)      256
Activation(Elu)              (None, 128, 128, 64)      0
UpSampling2D                 (None, 256, 256, 64)      0
Conv2D                       (None, 256, 256, 32)      18464
BatchNormalization           (None, 256, 256, 32)      128
Activation(Elu)              (None, 256, 256, 32)      0
Conv2D                       (None, 256, 256, 32)      9248
BatchNormalization           (None, 256, 256, 32)      128
Activation(Elu)              (None, 256, 256, 32)      0
UpSampling2D                 (None, 512, 512, 32)      0
Conv2D                       (None, 512, 512, 11)      3179
BatchNormalization           (None, 512, 512, 11)      44
Activation(Elu)              (None, 512, 512, 11)      0
Conv2D                       (None, 512, 512, 11)      1100
=================================================================
Total params: 2,651,555
Trainable params: 2,650,637
Non-trainable params: 918
```

Figure 3.4: Decoder architecture assuming cropped images of size 512×512

```
Layer (type)                 Output Shape              Param #
=================================================================
InputLayer                   [(None, 32, 32, 2048)]    0
Conv2D                       (None, 16, 16, 2048)      37750784
Conv2D                       (None, 8, 8, 2048)        37750784
UpSampling2D                 (None, 16, 16, 2048)      0
Conv2D                       (None, 16, 16, 2048)      37750784
Conv2D                       (None, 16, 16, 2048)      37750784
UpSampling2D                 (None, 32, 32, 2048)      0
Conv2D                       (None, 32, 32, 2048)      37750784
Conv2D                       (None, 16, 16, 2048)      37750784
=================================================================
Total params: 226,504,704
Trainable params: 226,504,704
Non-trainable params: 0
```

Figure 3.5: Transfer architecture assuming cropped images of size 512×512

3.3 Training and Evaluation protocol

Regarding the single training steps, N_1 is optimized by minimizing a standard L_1 loss, while N_2 is trained using the cross entropy error. The weights of the transfer network are optimized by minimizing the reconstruction error (L_2 loss) between transformed and target features:

$$L_{Tr} = ||G_{1\to2}(E_1^{A\cup B}(x^A)) - E_2^A(x^A)||_2,$$

N_1 is trained for 100k steps with batch size 8 (each mini-batch contains random images from both domains), while N_2 is trained for 45k iterations with batch size 8. In both cases, the optimizer is Adam with initial learning rate $1e^{-4}$ and $\beta_1 = 0.9$. For a more effective training, also exponential decay with decay steps 3000 and decay rate 0.96 is applied. The transfer network is instead trained for 100k steps with batch size 1 and initial learning rate $1e^{-5}$. All three networks are trained with random crops of size 512×512. In order to asses objectively performances, we must establish a precise evaluation protocol. N_1 can be evaluated on both domains in the validation set using the standard

metrics described in [10]: Absolute Relative Error (Abs Rel), Square Relative Error (Sq Rel), Root Mean Square Error (RMSE), logarithmic RMSE (lower is better) and three δ^a accuracy scores computed as the percentage of pixels such that the maximum between the ratio and inverse ratio with respect to the ground truth is lower than 1.25^{α}. To evaluate semantic segmentation two popular global metrics are used: pixel accuracy, shortened *Acc.* (i.e the percentage of pixels with a correct label) and Mean Intersection Over Union, shortened $mIoU$ (as defined in [7]). The latter can also be reported per class to give a complete measurement of AT/DT performances. To make this metric compatible among datasets, we solve semantic segmentation on the 10 shared classes (Road, Sidewalk, Walls, Fence, Person, Poles, Vegetation, Vehicles, Traffic Signs, Building) plus the 'Sky' category defined as the set of points with infinite depth. Some of the Cityscapes classes are collapsed into one class: car and bicycle collapse into vehicle and traffic signs and traffic light into traffic sign. The remaining categories for Cityscapes are instead ignored. Computing $mIoU$ and Acc is not only important when evaluating the performances of the whole framework in the real domain, but is also fundamental as a sanity check during training of N_2 on $\mathcal{A}$. It is sensible that the more effective is N_2 on the downstream task, the higher are the general performances of AT/DT. The same reasoning can be applied on N_1 when trained on $\mathcal{A}$ and $\mathcal{B}$: better results lead to a superior DA method. During the training phase of the transfer network, the model is evaluated on the validation set of Carla. Of course, it is possible the global optimum for Carla may not be a global optimum for Cityscapes. Yet, we believe that it is important not to use data from the target domain neither for hyper-parameters tuning or early stopping, because this information would not be available in a real case scenario. The Cityscapes validation set is only used at test time to measure the real performances of the adaptation method.

Chapter 4

AT/DT Extended

In this chapter, we propose and analyze some extensions to AT/DT, motivating each of them. The problem can be tackled from several perspectives. For example, we can introduce architectural improvements or changes in the training protocol. To be able to actually implement these extensions, it is necessary to have a complete understanding of the framework, so that upgrades can be done at any level. Moreover, due to the high complexity of the project, we decided to implement everything with new powerful tools such as Tensorflow 2.0, which gives the possibility to quickly prototype complex neural networks in a short period of time (AT/DT was originally implemented in Tensorflow 1.12). Tensorflow 2.0. makes a huge step towards a simpler and cleaner deep learning framework compared to the 1.12 version. For this reason, we think that it is worthy to spend some time upgrading the existent code. This is not a trivial step since there are relevant differences between these two framework versions. Indeed, only marginal parts of the original implementation were taken, such as the data augmentation pipeline to be applied to all images before feeding them to the model. Still, using a newer version of Tensorflow, reduces substantially the time required for implementing the same architecture, considering that using another framework such as Pytorch would have required to rewrite everything from scratch. More details on Tensorflow 2.0 and other important tools used for this work are given in the following dedicated chapter. To improve AT/DT we propose a list of extensions that can be applied to the architecture. The order in which we propose them is not casual but carefully programmed. We firstly

tackle the main problems of the original version and perform many tests that serve as hyper-parameters tuning. Once we obtained some important insights on these, we go deeper and we try to understand whether more powerful architectures can be deployed for our purpose. Finally, we use some of the most common Self-supervised tasks to replace monocular depth estimation to test whether simpler tasks can be used to learn features. The outline of our work can be summarized with the following steps:

1. **Ablation study on the number of channels of the transfer network.** The first characteristic one may notice about the transfer network is its huge number of parameters. In this section, we study the impact of the number of channels and to which extent is possible to reduce this number.

2. **Batch normalization in the transfer network.** The original version does not include batch normalization layers in the transfer network. Here, we study several options such as including it or not and the effect of the batch size.

3. **Deeplab [6] vs UNET [32] as backbone network.** A classical encoder-decoder architecture may not be the best choice for complex tasks such as monocular depth estimation and semantic segmentation. For this reason, we propose different backbones.

4. **Flat transfer network.** In addition to batch normalization, other structural changes can be done in $G_{1\to 2}$. Avoiding to shrink and consequently up-sample deep features when mapping one task to another one may be highly beneficial.

5. **Adversarial training.** The transfer network is optimized by minimizing a L_2 loss. This may not be optimal given the high dimensionality of the feature space. One plausible solution is to deploy adversarial training.

6. **Self-supervised learning.** Even though there are off-the-shelf algorithms able to estimate a depth map given a scene, these labels are far optimal. In this section, we study the possibility to train N_1 with the help of Self-supervised learning. Tasks such as Autoencoder, Colorization, Edge Detection, and Rotation prediction are investigated.

We now analyze thoroughly each of the previous point. From now on, we work under the default setting: *Dep.* $\rightarrow$ *Sem.* and evaluation performed on the validation set of $\mathcal{B}$ (i.e. Cityscapes).

4.1 Ablation study on the number of channels of the transfer network

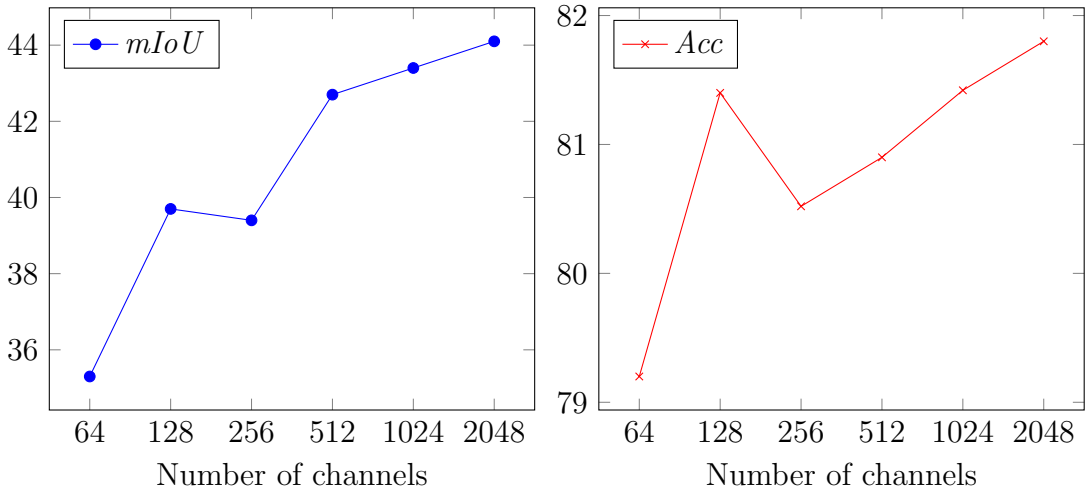


Figure 4.1: Results with different number of channels in $G_{1\rightarrow 2}$ (left side *mIoU*, right side *Acc*). Results obtained from the Cityscapes validation set.

We start by studying the most important component of the framework: $G_{1\rightarrow 2}$. As stated before, it consists of a simple CNN where each layer has 2048 input channels and 2048 output channels with a kernel of size 3. Hence, by assuming a three-channel input image with resolution 512×512, the input and output map of $G_{1\rightarrow 2}$ have size $32 \times 32 \times 2048$. The number of parameters for each layer is thereby $3 \times 3 \times 2048 \times 2048 = 37.748.736$. This huge number of parameters is probably not needed. Being able to reduce these channels has two benefits. First of all, it reduces training time. Secondly, by cutting the number of parameters we are allowed to modify the structure of $G_{1\rightarrow 2}$. For example, by saving some memory, one can use a deeper architecture or simply avoid to shrink the activation maps. This is an important step that will be considered again

later. As an ablation study, we tested the same architecture, varying only the number of channels from 2048 to 256. As fig. 4.1 shows, there is a proportional correlation between both $mIoU$ and Acc and the number of channels. Although results are noisy due to the choice of validating the model on source data only, the differences when using 512, 1024 or 2048 channels are not large. We can thereby select 1024 as it is a good compromise between memory requirements and performances. A transfer network with 1024 channels for each layer has about 96M parameters against the 226M required by the original implementation.

4.2 Batch normalization in the transfer network

Batch normalization is a popular technique for improving the speed, performance, and stability of a neural network. AT/DT original implementation does not include this type of layer in $G_{1\to2}$. It is thereby sensible to apply one batch normalization layer for each convoluational layer together with several batch sizes to study its effect during the transfer process. This layer was not utilized in the original version possibly because of the side effect that can be intruded in the DA setting by batch normalization. As explained in section 2.2.1.3, and well documented in [24], BN parameters are largely influenced by the input data. This may cause a performance drop when testing a model since, after the training phase, each layer of the network expects data belonging to a certain distribution (the one learned from the source domain), that could differ from the distribution of the target domain. Table table 4.1 summarizes results for several possible configurations. There are no evident differences introduced by batch normalization, and from our experiments it does not seem to hurt performances. It is clear though that increasing batch size is harmful. A possible explanation may be that when learning a mapping between tasks, if more than one image is used for each iteration, the network tries to learn an average mapping that works well for all of them. fig. 4.2 confirms this intuitive explanation: the higher is the batch size, the worse are small or rare objects correctly transferred among domains. For instance, when batch size is 1, the model is able to correctly classify some of the traffic signs, while this does not happen with larger batch sizes. On the other hand, when using batch size 1, the output map is noisier

(see bottom part of the top right prediction in fig. 4.2). This also explains the slight improvement in terms of *Acc* when using larger mini-batches. Given these findings, in the following extensions, we will always assume to train $G_{1\to 2}$ with batch size 1 and using batch normalization layers.

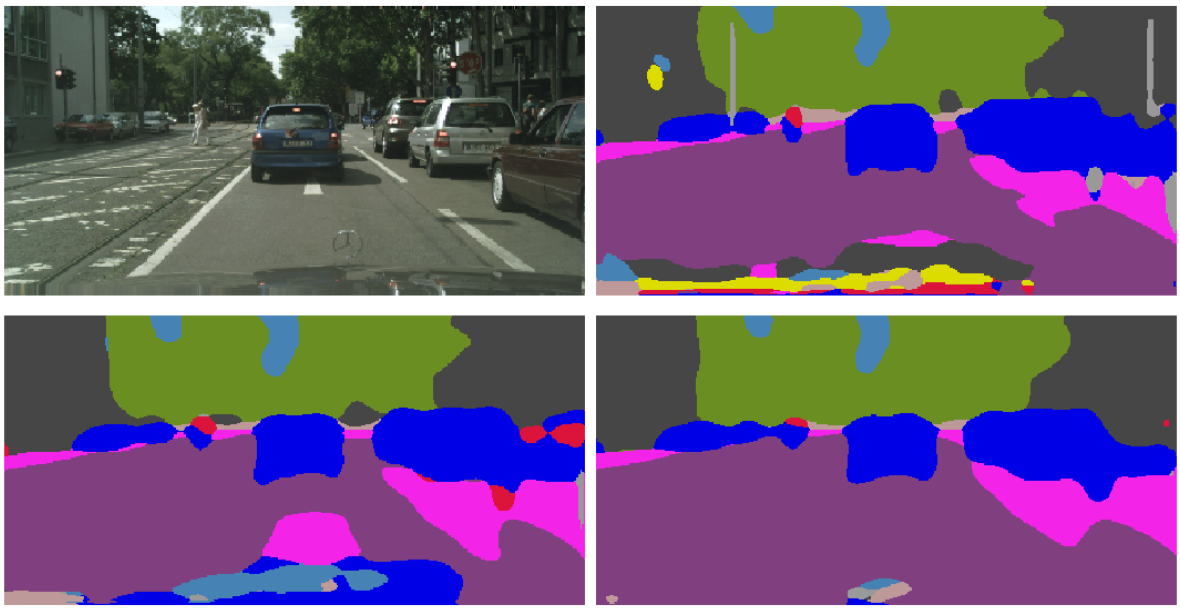


Figure 4.2: Visual effect of a different batch size. From top left to bottom right: RGB input, prediction with $BS = 1$, $BS = 8$, $BS = 16$. All predictions are obtained without batch normalization layers.

	mIoU	Acc
bs=1, BN=no	43.46	81.42
bs=1, BN=yes	**43.65**	82.46
bs=8, BN=no	42.1	82.6
bs=8, BN=yes	41.3	**83.7**
bs=16, BN=no	37.2	80.0
bs=16, BN=yes	38.1	82.1

Table 4.1: Batch Normalization effect on the transfer network

4.3 Deeplab vs UNET as backbone network

When a pixel level understanding is required, it essential to maintain as much as possible low-level details. In the down sampling path of an encoder, although rich features at different scales are captured by the network, the low-level characteristics of an image are lost due to the aggressive shrink of the input resolution. Thereby, commonly used networks such as a ResNet50 that reduces the input to be 1/32 of the original size, are not ideal in the semantic segmentation case. This is also the reason why the encoder of AT/DT is a DRN (Dilated Residual Network), which increases the resolution of output feature maps without reducing the receptive field of individual neurons. This is done by replacing the striding in the last two convolutional groups of a ResNet with dilated convolutions. This allows to shrink by only a factor of 1/16 the input image. In addition to a DRN, two more common architectures can be used to retain as much as possible low-level details, and consequently to obtain more fine-grained prediction maps. These are DeepLab and UNET. The UNET was originally developed for Bio Medical Image Segmentation but it quickly became used in many different scenarios. The architecture contains two paths. The first one is the contraction path (the encoder), which is used to extract rich features from the image, while the second one is the symmetric expanding path (the decoder) which is used to up sample the encoded features to the original input size so that a probability distribution for each pixel can be estimated. The up-sampling operation can be done with transposed convolutional layers or simply by bilinear up-sampling. The model is then augmented with skip connections between the encoder and the decoder: at each down sampling operation in the encoder, the activation maps are stored and given in input to the corresponding layer of the decoder. This has two benefits. Firstly, the skip connections allow a better flow of the gradient during back propagation. Then, thanks to these connections, at each layer of the decoder low-level details captured by the encoder can be directly integrated with the rich features modeled by the decoder, resulting in more fine-grained predictions. fig. 4.3 shows the UNET architecture.

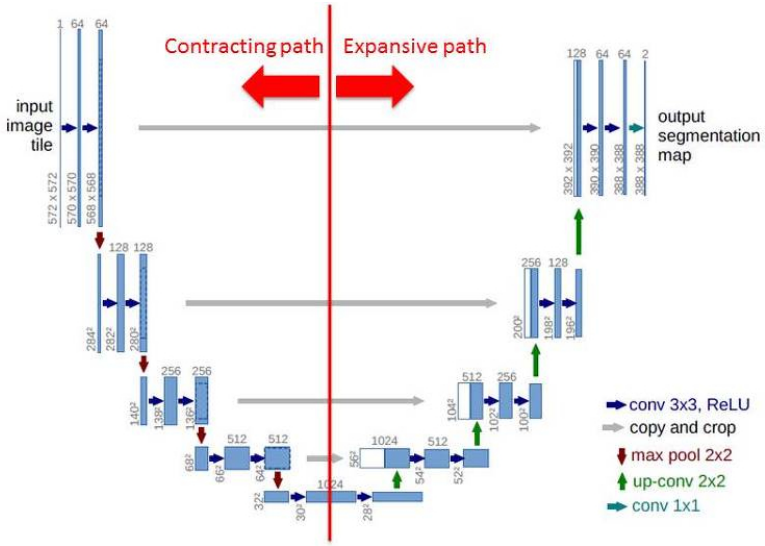


Figure 4.3: UNET architecture [32]

This architectural modification not only changes the structure of N_1 and N_2, which are now two identical UNET (the only difference is in the last layer), but also requires an update to the transfer network. This is due to the fact that by using a UNET, the decoder D_2 now expects for each layer two inputs (i.e. the output of the previous layer and the output from the layer with the same spatial resolution in E_2). This means that a transfer network must be applied to every skip connections so that the mapping between the two tasks can be learned at different scales. During training, $G_{1\to 2}$ is optimized by minimizing again the L_2, that is now computed at each skip connection level:

$$L_{Tr} = \sum_{i}^{n} L_{Tr_i}$$

$$L_{Tr_i} = ||G_{1\to 2}(E_{1_i}^{A\cup B}(x^A)) - E_{2_i}^{A}(x^A)))||_2,$$

with i denoting the $i-th$ down sampling layer of each encoder and n the total number of skip connections. The objective of Deeplab is essentially the same, although it relays on a different idea. Instead of using skip connections between encoder and decoder, it uses the so called ASPP module(Atrous Spatial Pyramid Pooling). fig. 4.4 represents the DeepLab architecture. It is very similar to the DRN architectures but enhanced with the ASPP module, which applies to the output of the encoder one 1×1 convolution, three

3×3 convolutions with rates = (6, 12, 18) and an image pooling to capture the global context. Each convolution has 256 filters and batch normalization. The outputs are then concatenated and convolved again with a 1×1 filter to reduce depth. In the context of AT/DT, both E_1 and E_2 are replaced with the architecture depicted in fig. 4.4, while the decoders remain unvaried. The mapping between tasks is thereby applied after the ASPP module, with the hope that multiple-scale features can be transferred. The L_2 loss is minimized as usual.

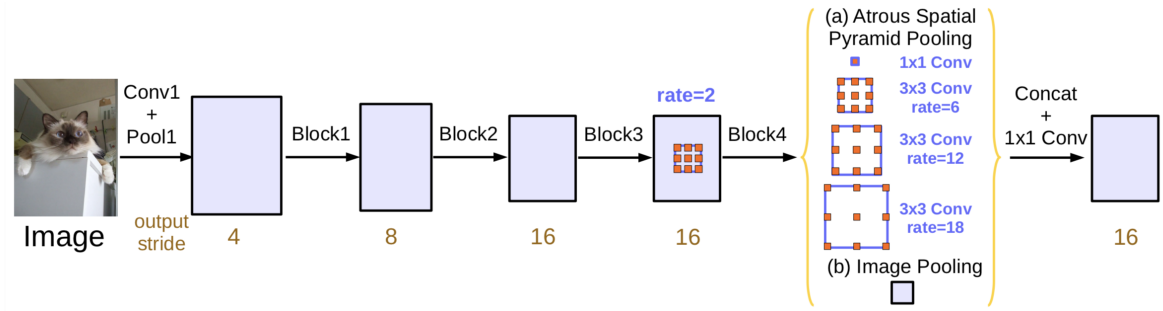


Figure 4.4: DeepLab Architecture [6]

4.4 Flat transfer network

Since the transfer network is trained on $\mathcal{A}$, it is reasonable to assume that if $G_{1\to2}$ has effectively learned to map depth features into semantic features, when applying AT/DT (i.e. $D_2(G_{1\to2}(E_1(x)))$) to an input image belonging to the Carla dataset, results should not change. Of course, this is only an ideal situation since it is not possible to perfectly convert features between tasks; still, a good result is expected. With this observation in mind, after the training process, one can test $G_{1\to2}$ on the Carla validation set, and compare this result with the one obtained by N_2. By doing this simple test, we can check the effectiveness of the transfer network in the following way: if a large gap in performances is present when applying $G_{1\to2}$ in $\mathcal{A}$, we can conclude that the learned mapping is unsatisfactory. With this observation in mind, we propose to an architectural upgrade to limit as much as possible the side effect of the transfer network, even when applied inside the domain it has been trained on. We thereby propose a transfer network composed of 6 convolutional layers with no striding and bilinear up sampling, each

followed by a batch normalization layer, so that $G_{1\to2}$ does not perform any shrinking or up sampling operation. It is important to note that performing convolution without striding increases significantly memory usage since we are not using more parameters, but we are keeping in memory bigger tensors. This solution can be applied thanks to the change done in section 4.2, which allows us to reduce the required memory to train our model. fig. 4.5 details this new architecture:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
InputLayer                   (None, 32, 32, 2048)      0
Conv2D                       (None, 32, 32, 1024)      18875392
BatchNormalization           (None, 32, 32, 1024)      4096
Activation(Elu)              (None, 32, 32, 1024)      0
Conv2D                       (None, 32, 32, 1024)      9438208
BatchNormalization           (None, 32, 32, 1024)      4096
Activation(Elu)              (None, 32, 32, 1024)      0
Conv2D                       (None, 32, 32, 1024)      9438208
BatchNormalization           (None, 32, 32, 1024)      4096
Activation(Elu)              (None, 32, 32, 1024)      0
Conv2D                       (None, 32, 32, 1024)      9438208
BatchNormalization           (None, 32, 32, 1024)      4096
Activation(Elu)              (None, 32, 32, 1024)      0
Conv2D                       (None, 32, 32, 1024)      9438208
BatchNormalization           (None, 32, 32, 1024)      4096
Activation(Elu)              (None, 32, 32, 1024)      0
Conv2D                       (None, 32, 32, 2048)      18876416
BatchNormalization           (None, 32, 32, 2048)      8192
Activation(Elu)              (None, 32, 32, 2048)      0
=================================================================
Total params: 75,533,312
Trainable params: 75,518,976
Non-trainable params: 14,336
```

Figure 4.5: Flat transfer architecture

4.5 Adversarial training

Adversarial training is widely used in many different scenarios, and it is particularly suitable in situations where the loss function to minimize is not clear, as in the case of Domain Adaptation. For example, when solving semantic segmentation on a target domain, minimizing directly the cross entropy error using source data, is likely not the best possible option. As seen in section section 2.2.2.2 in fact, Adversarial training can be used to alleviate domain shift by simply exploiting alternative loss functions. Therefore, we propose two ways of using adversarial training to train $G_{1\to 2}$, which aim to perform domain alignment on the output space of the transfer network.

4.5.1 Domain alignment through domains

The idea of performing domain alignment on the transfer network is to help $G_{1\to 2}$ to generalize better on $\mathcal{B}$. Considering that this mapping is learned using only images from $\mathcal{A}$, $G_{1\to 2}$ may also suffer of bad generalization due to domain shift. Moreover, since $G_{1\to 2}$ makes use of batch normalization, it is also important that the statistics and the parameters of each BN layer are updated taking into account images belonging do both domains. For this reason, we embed the model with a simple discriminator C that is trained to recognize whether the features obtained by applying the transfer network come from $\mathcal{A}$ or $\mathcal{B}$. On the other hand, $G_{1\to 2}$ should be able to fool the discriminator and to minimize the L_2 loss to learn an effective mapping among tasks:

$$\begin{aligned}\min_C \mathcal{L}_{\text{adv}_C}\left(\mathbf{X}_A, \mathbf{X}_B\right) = &- \mathbb{E}_{\mathbf{x}\sim\mathbf{X}_A}\left[\log C\left(G\left(E_1(x)\right)\right)\right] \\ &- \mathbb{E}_{\mathbf{x}\sim\mathbf{X}_B}\left[\log\left(1 - C\left(G\left(E_1(x)\right)\right)\right)\right]\end{aligned}$$

$$\min_G \mathcal{L}_{\text{adv}_{Tr}}\left(\mathbf{X}_B\right) = - \mathbb{E}_{\mathbf{x}\sim\mathbf{X}_B}\left[\log C\left(G\left(E_1(x)\right)\right]\right.$$

$$\min_G \mathcal{L}_{\text{Tr}}\left(\mathbf{X}_A\right) = ||G_{1\to 2}(E_1(x)) - E_2(x)||_2$$

By weighting the previous terms with appropriate weights ($\lambda_1 = 1, \lambda_2 = 0.0001, \lambda_3 = 0.0001$) we obtain the total objective function:

$$\mathcal{L} = \lambda_1 \mathcal{L}_{\text{Tr}} + \lambda_2 \mathcal{L}_{\text{adv}_{Tr}} + \lambda_3 \mathcal{L}_{\text{adv}_D}$$

4.5.2 Task mapping with adversarial training

Alternatively, adversarial training can be deployed to learn a better mapping between tasks. We can, in fact, use a discriminator to make features produced from $G_{1\to2}$ even more similar to the one obtained by the target encoder. More precisely, the discriminator has to determine whether the input feature maps come from E_2 or the transfer network, while $G_{1\to2}$ is trained with the usual L_2 loss and to fool the discriminator:

$$\begin{aligned}\min_C \mathcal{L}_{\text{adv}_C}(\mathbf{X}_A) = &- \mathbb{E}_{\mathbf{x}\sim\mathbf{X}_A}\left[\log C\left(E_2(x)\right)\right] \\ &- \mathbb{E}_{\mathbf{x}\sim\mathbf{X}_A}\left[\log\left(1 - C\left(G\left(E_1(x)\right)\right)\right)\right]\end{aligned}$$

$$\min_G \mathcal{L}_{\text{adv}_{Tr}}(\mathbf{X}_A) = - \mathbb{E}_{\mathbf{x}\sim\mathbf{X}_A}\left[\log C\left(G\left(E_1(x)\right)\right]\right.$$

$$\min_G \mathcal{L}_{\text{Tr}}(\mathbf{X}_A) = ||G_{1\to2}(E_1 - E_2(x)||_2$$

Again, we can set $\lambda_1 = 1, \lambda_2 = 0.0001, \lambda_3 = 0.0001$ and obtain the final loss:

$$\mathcal{L} = \lambda_1\mathcal{L}_{\text{Tr}} + \lambda_2\mathcal{L}_{\text{adv}_{Tr}} + \lambda_3\mathcal{L}_{\text{adv}_D}$$

In both cases, the architecture of the discriminator consists of 4 convolutional layers that shrink the input activation map to a $4 \times 4 \times 1$ map, where for each patch the output is a value between 0 and 1 (i.e. the probability of the patch to belong to domain $\mathcal{A}$ or $\mathcal{B}$). For a more stable training, we embed the discriminator with Spectral Normalization [26], which has been shown to improve the stability of GANs.

4.6 Self-supervised learning

So far we exploited the fact that monocular depth estimation and semantic segmentation are correlated. We also know by now that in order to run AT/DT partial supervision is required on $\mathcal{B}$, hence a strategy for obtaining such labels is needed. In the case of monocular depth estimation, proxy labels can be obtained by filtering SGM disparities through confidence measures. Doing this is not that simple since it requires some expertise in the specific topic: it would be much easier to deploy other general

tasks. For this reason, we study here the possibility to replace depth estimation with Self-supervised tasks. In particular, we focus on four such tasks: Autoencoder, Rotation prediction, Colorization, and Edge detection. We provide now details on how these tasks are solved since they require some adjustments to the architecture and we discuss results in the following chapter. All these tasks are trained in the same way as done for monocular depth estimation (i.e. on both $\mathcal{A}$ and $\mathcal{B}$).

4.6.1 Autoencoder

An Autoencoder is a neural network that learns to efficiently compress data and to reconstruct it back from the encoded representation. The task is unsupervised, in the sense that the label is the input itself. To reconstruct the input, the network must learn how to reduce data dimension ignoring noise and redundant information, retaining thereby only important features from the data. The decoding process is lossy, since some information is usually lost, but for our purposes this is not a problem. What we would really like to know, is instead whether the features captured by a simple autoencoder, can be transferred to another task. To change as little as possible, the autoencoder uses the same backbone deployed for monocular depth estimation, namely a DRN as encoder that reduces the input by 1/16, and a stack of convolutional and bi-linear up-sample layers as decoder. The only difference is the final prediction layer, that has to produce a 3-channel activation map. The Autoencoder is optimized by minimizing the L_2 loss between RGB input image and the 3-channel output logits. Afterward, $G_{1\to 2}$ is trained as usual to map features produced by the autoencoder into semantic segmentation features. fig. 4.6 shows a reconstruction of images belonging to both domains.

Figure 4.6: Image reconstruction with an Autoencoder. Top row input images (and ground truths), bottom row reconstructed images.

4.6.2 Rotation prediction

Rotation prediction is a Self-supervised task that has already been explored in the context of object classification, while for semantic segmentation only few works have used it (see section 2.2.4). In this case, the objective of the networks is to predict the degree of rotation in which the original image has been rotated. Only four angles of rotations are admitted (0, 90, 180, 270). In order to solve this problem, a neural network must capture complex features such as the shape of an object. Moreover, also a semantic understanding of the scene is required since in some case the shapes are not enough; for example, the degree of rotation of a vertical symmetric object can be ambiguous). These complex features can also be useful for the downstream task. Again we maintain fixed the encoder, while the decoder is only composed by a convolutional layer, an average pooling layer and a final dense layer with a softmax activation (see details in fig. 4.7). We chose such a light decoder so that the encoder is the one responsible for capturing rich features, and the decoder prediction is only based on the encoder representation.

```
Layer (type)                 Output Shape              Param #
=================================================================
InputLayer                   (None, 32, 32, 2048)      0
Conv2D                       (None, 16, 16, 128)       2359424
GlobalAveragePooling2d       (None, 128)               0
Dense                        (None, 4)                 516
Activation(Softmax)          (None, 4)                 0
=================================================================
Total params: 2,359,940
Trainable params: 2,359,940
Non-trainable params: 0
```

Figure 4.7: Rotation Decoder

4.6.3 Image Colorization

Colorization is another simple task that has been used as a Self-supervised task. The objective in this case is to reproduce a colored image starting from its gray-scale representation. The problem is conceptually similar to an autoencoder, although in this case we are not performing data compression, but rather we ask to the network to add information such that the color is restored. There are several way to implement a network capable of solving such task, and in this work we chose the most simple one, that consists in treating it as a regression problem: from a gray scale image the network must predict 3 float numbers between 0 and 1 (as done for the autoencoder). The network is optimized my minimizing the L_2 loss between the predicted output and the original RGB image. The consequence of this choice is that the restored images will appear more grayish since an object can have different colors, and the optimal solution for a L_2 is roughly the mean value. Although this is not the best way to solve Image Colorization, our concern is to capture useful features for the target task, rather than obtaining sharp and brilliant colors. A more suitable way of tackling this problem would be to see the problem as a classification task, and predict a probability distribution for each pixel. This better represent the multimodal nature of the problem. fig. 4.8 illustrates two examples of Image Colorization performed by our network.

Figure 4.8: Example of Image Colorization. From Top to bottom: gray-scale image, prediction, colored ground truth.

4.6.4 Edge detection

Edge detection is a common and old vision problem that aims to identify pixels at which the image brightness changes sharply. Many works have been proposed to solve this task, most of them without the need for neural networks. The most simple way of computing edges is the 1D Step-Edge, which consists of computing the derivative of the signal in a point and apply a threshold; if in fact, the value is above the desired threshold, it means that a sharp change is present. However, in this work, we compute edges using one of the most advanced algorithms: Canny's Edge Detector [5]. This detector is based on the idea that edges can be detected by finding local extrema of the convolution of the signal by a first order Gaussian derivative. We rely on the OpenCV

implementation to compute edges for both domains (edges in $\mathcal{A}$ are computed starting from the semantic segmentation map rather than the RGB image itself to reduce noise), and then we use the typical encoder-decoder architecture to learn to extract edges using the output of Canny as proxy labels. We treat the task as a regression problem, hence the decoder output is a gray-scale image with values between 0 and 1 (edges are identified with white pixels). The encoder output is then mapped again into features for semantic segmentation minimizing the L_2 loss.

Chapter 5

Results

We report in this chapter all the results obtained with the proposed extensions. We start analyzing performances comparing the changes done at the architectural level. This will serve as model selection for the following tests. Then we report results obtained with adversarial training and compare it with the best result obtained without it. Finally, some thoughts and considerations are provided when replacing monocular depth estimation with several Self-supervised tasks.

5.1 Results with different architectures

To check the effectiveness of any DA method it is essential to establish a proper Baseline and an Oracle. Comparing a model with the former, allows us to determine the success or the failure of a solution, while the latter gives us the best possible results since we assume that target data is available. Our Baseline, consists of the network N_2 trained on $\mathcal{A}$ and directly tested on $\mathcal{B}$. The Oracle corresponds to the network N_2 trained and tested on $\mathcal{B}$. To set a more realistic goal, we also define a second Oracle *AT/DT Flat Oracle*, which refers to training both $G_{1\to2}$ and N_2 using target images only (normally they are instead trained using source images only). *AT/DT Flat Oracle* can be considered our real upper bound, as its score represent the best possible results achievable when mapping depth features into semantic segmentation features assuming to have labels on $\mathcal{B}$. Interestingly, the best results have been obtained with the Flat

transfer network (*mIoU* 48.04 and *Acc* 85.90), which is the simplest update we proposed. In fact, by simply avoiding to reduce the spatial dimension of the input features we are able to obtain a gain of +9.18 in terms of *mIoU* over the baseline. Also transferring features at different scales gives a noticeable gain, although not as much as the flat version. On the other hand, multi-scale features extracted with the ASPP module are less transferable. This suggests that rather than mixing information at different scales, it is better to learn a mapping among the corresponding levels of the two encoders. This may be conuter intuitive at a first glance since architectures based on DeepLab usually outperform simple encoder-decoder on the semantic segmentation task. However, our case is very different, because performances of AT/DT on the downstream task highly depends on the transferability of features extracted by $E1$ and $E2$, and the ones learned when embedding the ASPP module are probably too complex to be mapped with a simple L_2 loss. fig. 5.1, fig. 5.2, fig. 5.3, and fig. 5.4 illustrate qualitative examples. The superiority of the Flat architecture is clearly visible. For example, all the other models fail to capture small objects such as traffic signs and persons. Although transferring features at multiple levels as done with the skip connections version produces overall better results compared to the original version (which is only able to detect large blobs), it is also interesting to show that some noise is introduced (see pedestrians colored in red). It is also important to notice how our plain implementation of AT/DT obtains a gain of 3.94% in *mIoU* over the original version. This suggests that using a more real-looking synthetics dataset can noticeably improve any DA method.

Architecture	Road	Sidewalk	Walls	Fence	Person	Poles	Vegetation	Vehicles	Tr. Signs	Building	Sky	**mIoU**	**Acc**
Baseline	78.99	38.81	1.34	5.80	24.02	24.47	71.98	52.23	5.57	65.17	59.10	38.86	78.58
AT/DT [30]	76.44	32.24	4.75	5.58	24.49	24.95	68.98	40.49	10.78	69.38	78.19	39.66	76.37
AT/DT (ours)	84.66	38.52	3.95	9.19	26.99	14.09	75.72	68.35	10.49	73.90	73.71	43.60	82.86
AT/DT DeepLab	86.98	45.67	5.08	7.72	27.40	12.81	74.34	56.36	05.25	71.94	71.65	42.29	82.62
AT/DT Skips connections	**89.95**	**49.01**	4.98	**11.43**	**32.91**	18.64	74.15	66.86	10.18	72.08	**79.90**	46.37	84.60
AT/DT Flat	**89.95**	46.77	**5.16**	10.21	**28.93**	**28.92**	**77.50**	**71.37**	**19.24**	**75.29**	75.12	**48.04**	**85.90**
AT/DT Flat Oracle	89.69	48.05	11.46	29.58	59.68	35.84	85.83	85.57	34.03	78.17	85.54	58.50	88.84
Oracle	96.74	78.28	29.26	40.78	72.39	51.28	90.69	91.94	58.92	86.33	89.23	71.44	93.90

Table 5.1: Experimental results with different architectures on the **Cityscapes** validation set

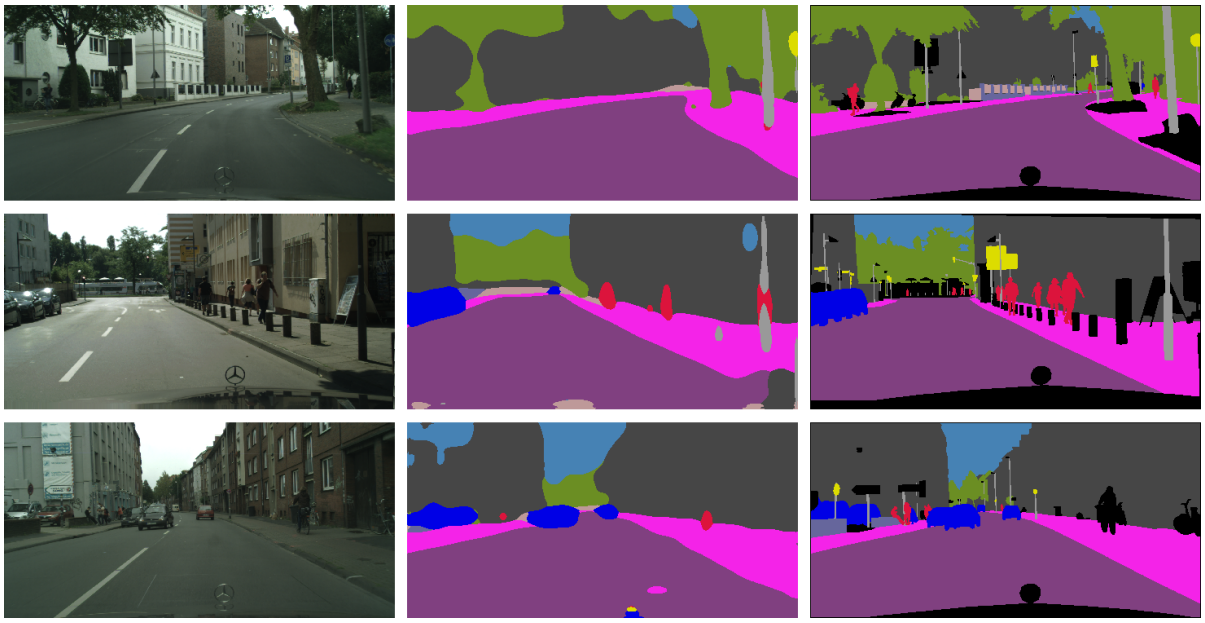


Figure 5.1: AT/DT qualitative examples. From left to right: RGB image, prediction, ground truth.

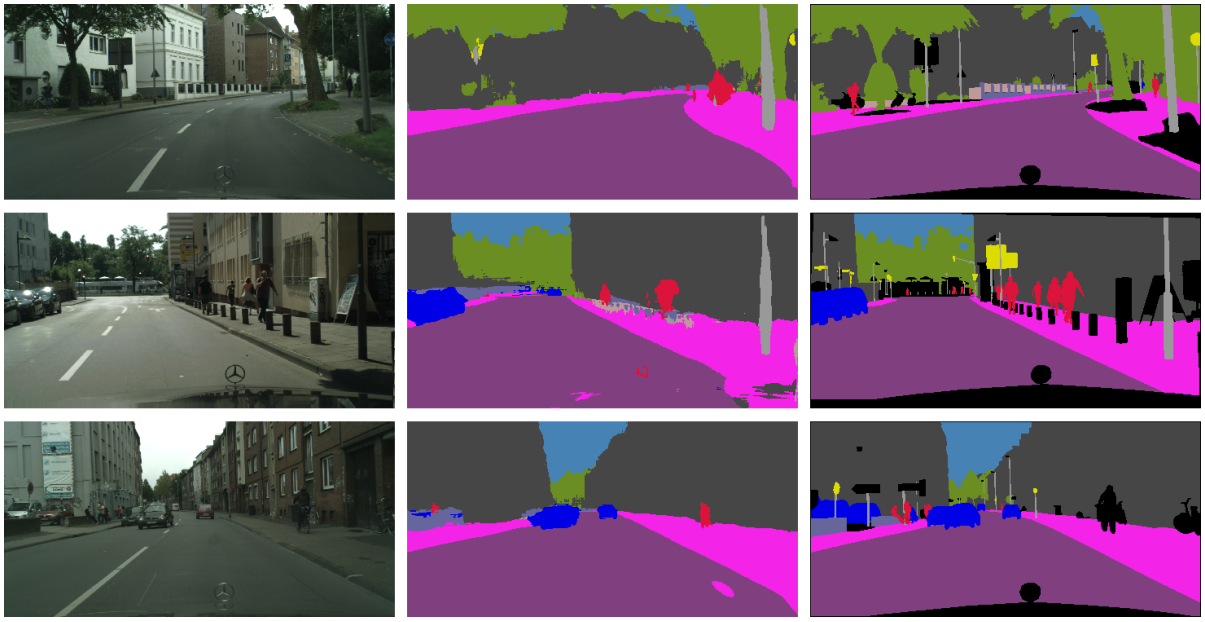


Figure 5.2: AT/DT with ASPP module qualitative examples. From left to right: RGB image, prediction, ground truth.

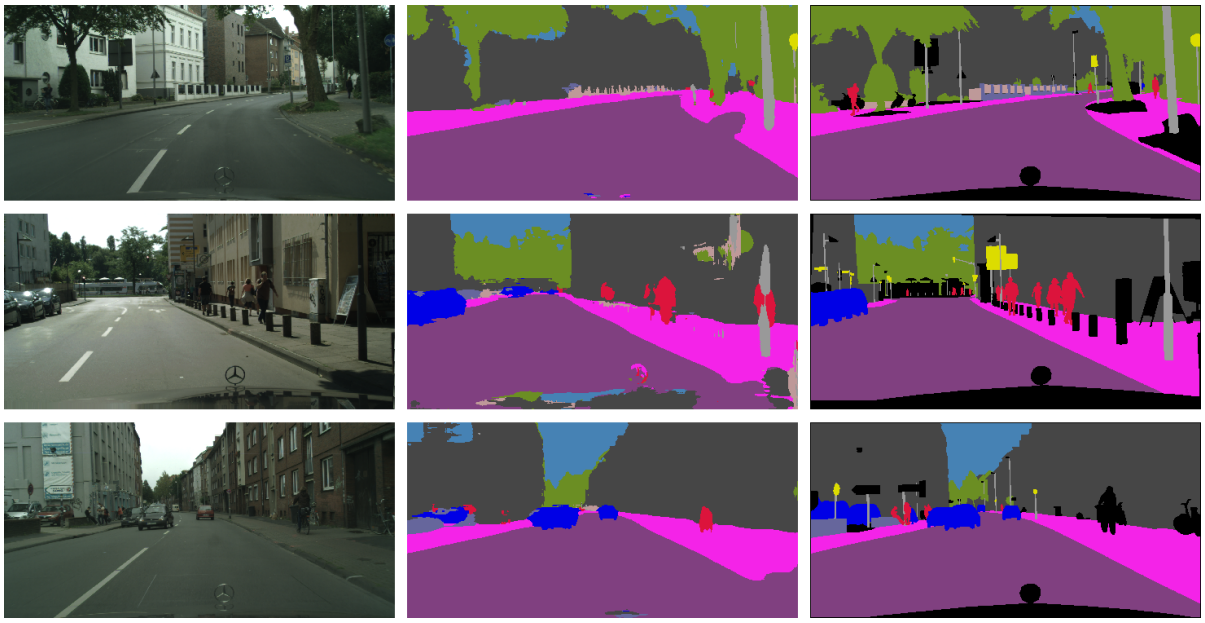


Figure 5.3: AT/DT with skips connections qualitative examples. From left to right: RGB image, prediction, ground truth.

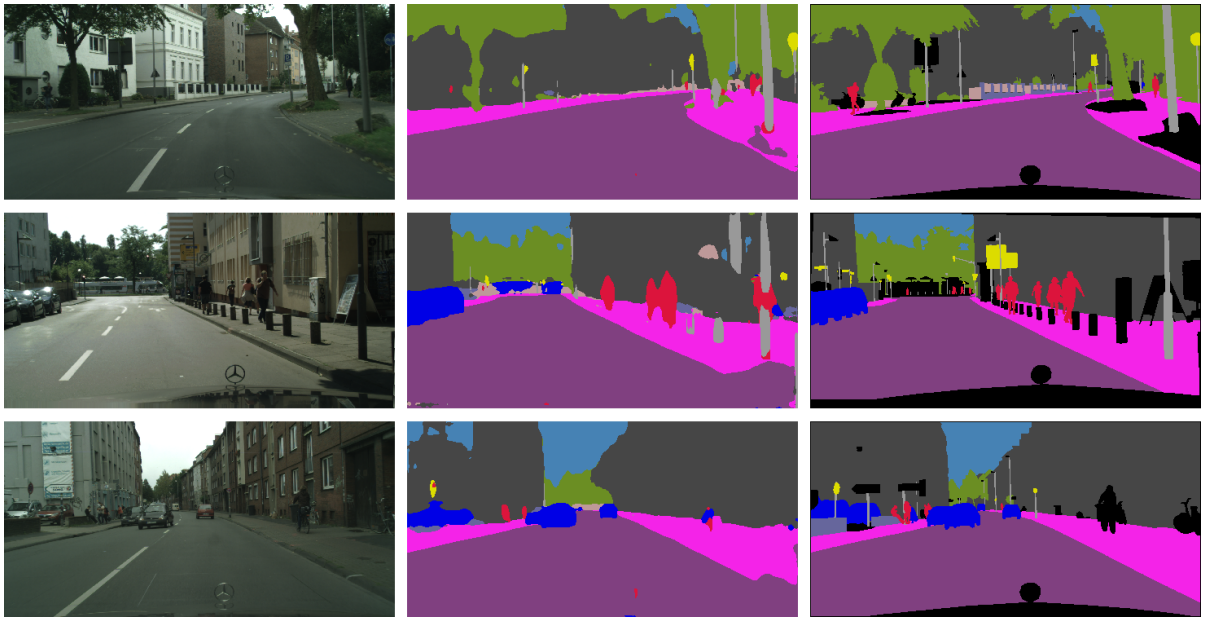


Figure 5.4: AT/DT Flat transfer qualitative examples. From left to right: RGB image, prediction, ground truth.

5.2 Results with Adversarial training

In the previous section we saw how a simple change at the architectural level was able to give us an important boost. Now we show results when deploying adversarial training together with the previous upgrade. In table 5.2 we compare AT/DT Flat with the two adversarial strategies we proposed in section 4.5. In the Domain Adversarial strategy $G_{1\to2}$ has to minimize the L_2 loss and to fool a discriminator that determines whether the output of the transfer network is computed starting from an image belonging to $\mathcal{A}$ or $\mathcal{B}$. In this case, we are thereby aligning the two domains. On the other hand, the second adversarial technique aims to improve the mapping among the two encoders by discriminating the outputs of $G_{1\to2}$ from outputs of E_2. Both solutions seem to be effective since we obtained a gain of +1.28 and (+1.47) respectively. We also combined the two strategies using two different discriminators, but it did not seem to further boost performances. By comparing these numbers with the scores obtained in table 5.1, we realized that our performances are not too far from the ones obtained by the Oracle *AT/DT Flat Oracle*. This highlights the effectiveness of our solutions, but at the same time, it suggests that more effort should be put to improve the oracle itself: improving performances when training both the transfer network and N_2 in the target domain may lead to better results in the standard setting. In fig. 5.5 and fig. 5.6 we report some qualitative samples although for humans is quite hard to appreciate such small improvements.

Strategy	Road	Sidewalk	Walls	Fence	Person	Poles	Vegetation	Vehicles	Tr. Signs	Building	Sky	**mIoU**	**Acc**
AT/DT Flat	89.95	46.77	5.16	10.21	28.93	28.92	77.50	71.37	19.24	75.29	**75.12**	48.04	85.90
AT/DT Domain Adv.	**90.80**	**48.91**	**6.16**	11.84	35.32	30.29	**78.78**	71.17	18.51	75.66	75.03	49.32	86.43
AT/DT Task Adv.	90.22	46.71	4.45	**12.35**	**37.86**	**30.73**	78.58	**73.07**	**20.21**	**76.06**	74.40	**49.51**	**86.63**

Table 5.2: Experimental results with adversarial training on the **Cityscapes** validation set

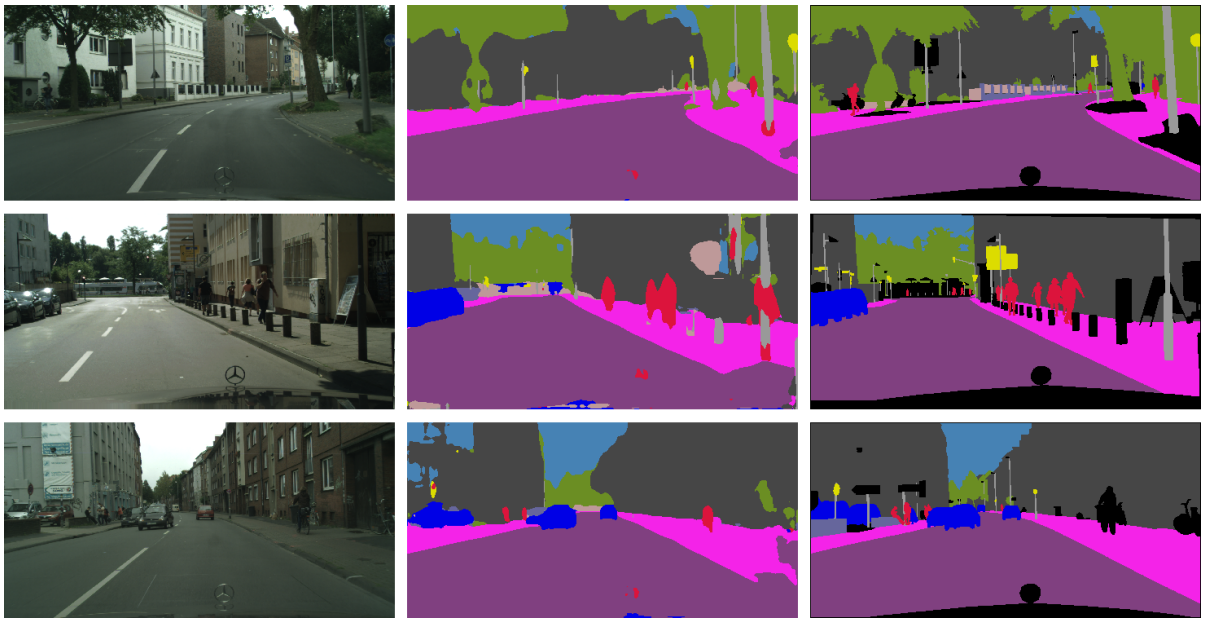


Figure 5.5: Segmentation maps obtained with Adversarial training across domains.

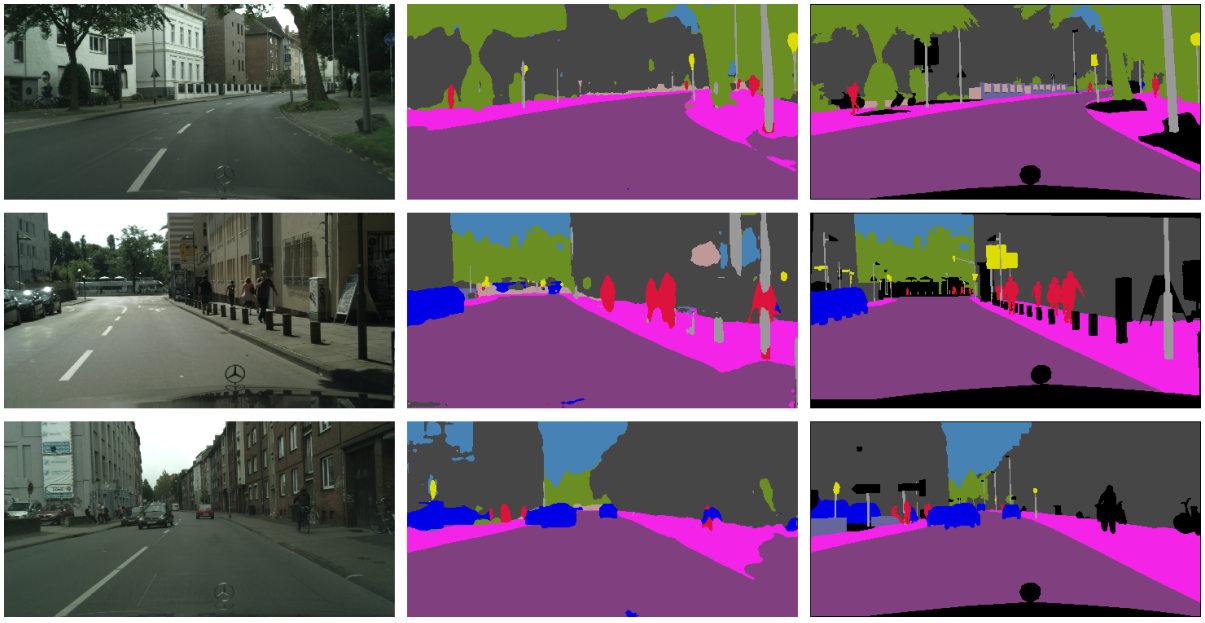


Figure 5.6: Segmentation maps obtained with Adversarial training across tasks.

5.3 Results with Self-supervised tasks

So far we focused our attention on the depth to semantic segmentation setting. However, as stated before, monocular depth estimation requires some task related knowledge since proxy labels must be generated somehow for the target domain. For this reason, we investigated different Self-supervised tasks to verify whether it is possible to transfer knowledge from a simpler task to a complex one such as semantic segmentation. From our experiments summarized in table 5.3, it seems that it is not possible (at least with these tasks and with our specific implementations) to boost performances of semantic segmentation by simply mapping features. To further confirm this hypothesis, we also reports results of AT/DT in $\mathcal{A}$ (see table 5.4). The large gaps between all the Self-supervised tasks and monocular depth estimation without even changing domain when testing suggests indeed the inadequacy of these settings. Among these, Rotation prediction seems to be the most effective when testing on the target domain. It is also the only task that requires some semantic understanding of a scene since the others (Autoencoder, Colorization, Edge detection) work with low-level details. However, we do not exclude that by combining somehow several tasks we can achieve similar results to depth estimation, even though it is not straightforward how to do it.

$\mathcal{T}_1$	Road	Sidewalk	Walls	Fence	Person	Poles	Vegetation	Vehicles	Tr. Signs	Building	Sky	**mIoU**	**Acc**
Autoencoder	60.24	19.33	1.67	1.67	4.12	8.00	33.15	10.49	0.69	17.89	62.66	19.99	52.91
Colorization	71.40	22.50	0.80	1.67	3.48	10.25	35.80	18.23	**1.27**	40.38	56.61	23.85	62.28
Rotation	**78.46**	**24.83**	**3.21**	**5.08**	**9.13**	**14.36**	**64.64**	**26.24**	0.27	**50.92**	**70.13**	**31.57**	**71.93**
Edge detection	63.82	16.60	0.67	1.37	6.55	10.26	47.62	4.42	0.11	33.90	38.87	20.38	58.33
Depth	89.95	46.77	5.16	10.21	28.93	28.92	77.50	71.37	19.24	75.29	75.12	48.04	85.90

Table 5.3: Experimental results on the **Cityscapes** validation set when mapping different tasks to Semantic Segmentation. Best results highlighted in bold.

$\mathcal{T}_1$	Road	Sidewalk	Walls	Fence	Person	Poles	Vegetation	Vehicles	Tr. Signs	Building	Sky	**mIoU**	**Acc**
Autoencoder	80.50	65.79	33.66	31.74	10.19	35.35	82.45	59.16	09.36	53.98	78.80	49.18	84.35
Colorization	90.87	68.93	34.82	36.95	**12.60**	38.81	82.50	60.45	**32.29**	62.71	**94.18**	55.92	88.89
Rotation	88.87	66.05	28.75	29.86	7.24	39.02	79.92	44.79	05.31	58.75	93.04	49.24	87.05
Edge detection	**91.55**	**72.98**	**39.59**	**45.67**	2.01	**49.50**	**85.07**	**63.03**	16.92	**68.77**	93.67	**58.81**	**90.36**
Depth	92.76	76.55	72.67	55.78	35.13	65.45	88.57	79.64	61.22	83.65	94.12	73.23	93.20

Table 5.4: Experimental results on the **Carla** validation set when mapping different tasks to Semantic Segmentation. Best results highlighted in bold.

Chapter 6

Technologies

Many frameworks are available in the Deep Learning world. The most common are probably Tensorflow, Keras and Pytorch. Each of them has its own advantages and disadvantages. For this thesis, we decided to use Tensorflow 2.0, since this new version was released right before the start of this project. Google's white paper [1] introduced TensorFlow in 2015, and it was the first choice for many Deep Learning practitioners and researchers, although it wasn't really user-friendly. Over the years, thanks to the great community behind Tensorflow, the framework has been greatly improved and recently merged with Keras, which is a high-level API that can sit on top of other Deep Learning frameworks. Essentially, Keras has become the high-level API for Tensorflow 2.0. The beauty of Keras lies in its ease of use. Defining neural networks is intuitive, simple and it provides full expressiveness. Last but not least, we have Pytorch. Pytorch is a Deep Learning framework developed by Facebook's AI research group and released in 2016. Pytorch received immediately great attention from the research community thanks to its native integration with Python. Writing code in Pytorch is essentially the same as using Python, while this is not true for Tensorflow (at least until version 2.0). In terms of coding style, we can say that Pytorch lies somewhere in between Keras and TensorFlow, even though differences have narrowed with the introduction of Tensorflow 2.0.

6.1 Tensorflow

As stated before, in this work we used Tensorflow 2.0. The are many reasons why we made this choice. Firstly, the last release allows defining very complex architectures with few lines of code, thanks to its new high-level API. Keras has been adopted for this purpose, hence all its advantages have been included in Tensorflow 2.0, together with all the tools that were already available in Tensorflow 1.x, such as tf.data (a library for creating efficient input pipeline). Another good feature is eager execution by default. Differently from the previous version in fact, Tensorflow 2.0 executes all operations eagerly (like Python normally does). This behavior simplifies the debugging process since it gives the possibility to execute operations and get the result immediately. However, this new features comes at cost: performances are considerably worse compared to the Tensorflow 1.x graph-based execution. However, TF 2.0. makes available graph mode execution too by simply using the *tf.function* decorator. The suggested pattern is thereby to implement and debug with eager execution, and use graph mode for long trainings to exploit the benefits of graph mode. Lastly, great improvements have also been done to support distributed training and easier deployment. The following snippet of code taken from the Tensorflow official documentation, shows how easy is to define a convolutional neural netowrk to solve digit classification on the MNIST dataset.

```
import tensorflow as tf
mnist = tf.keras.datasets.mnist

(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', stride=2,
    input_shape=(32, 32, 3)))
model.add(layers.Conv2D(64, (3, 3), activation='relu', stride=2))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.Flatten())
```

```
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))


model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```

6.2 GCP

The GCP (Google Cloud Platform) is a cloud service that consists of a set of physical resources distributed in Google's data centers and offered as virtual resources to the final user. Among the available resources, there are Computing and hosting capabilities, Storage and Networks Infrastructure, and more importantly for this project Big Data and Machine Learning services. We chose to relay on the GCP ecosystem thanks to its great compatibility with Tensorflow and the wide support for Deep Learning applications. Like other public cloud platforms, most of the Google Cloud Platform services follow a pay-as-you-go model in which there are no upfront payments, and users only pay for the cloud resources they consume.

6.2.1 Compute Engine

Compute Engine can be thought as an infrastructure as a service (IaaS), and is one of the main services provide by Google thanks to its flexibility and utility. With regard to this thesis, it has been mainly used to instantiate and run fully customizable virtual machines. Thanks to Compute Engine in fact, one can quickly spin up a machine with all the required hardware and package configuration. This is done through a simple menu in which it is possible to select the desired options among the available solutions. For

example, in our case, we used a virtual machine with 16GB of RAM and a NVIDIA T4 GPU to run our trainings.

6.2.2 Google Cloud Storage

Google Cloud Storage is a flexible, scalable, and durable storage service. It provides a storage option for all the instanced virtual machine instances, so that the same data can be accessed by any instance. This is very handy in Deep Learning projects. For example, in our context, we used Google Cloud Storage to store our datasets, and run many virtual machines at the same to train different models completely in parallel. To use such storage service, it is sufficient to define a global object, called *Bucket*. There is a single global namespace shared by all buckets, hence the name must be unique. Once a bucket is defined, it provides a hierarchical structure in which is possible to create, store, read and write folders and files.

6.2.3 Big Data Services

The GCP offerse many Big Data Services, such as Google BigQuery, AI Platform, Google Cloud Dataproc and Dataflow. We briefly present Dataflow, since it is the only service in this category that we used for this project. Dataflow provides a managed service and set of SDKs to perform batch and streaming data processing tasks. For example, it can be used to run pipelines written using the Apache Beam library. Once a job is started, Cloud Dataflow automatically spins up a cluster of virtual machines, distributes the tasks among them, and dynamically scales the cluster based on how the pipeline is performing. These characteristics makes it very attractive for high-volume computation, especially when the processing tasks can clearly and easily be divided into parallel workloads. This is the case of data pre-processing for instance. As every data scientist knows, one of the most difficult phase of an end-to-end project is the data preparation step, that consists in understanding and modelling data to make it suitable for machine learning models. To this purpose, we used Dataflow to run a pipeline to convert our datasets in TFRecords files. A TFRecord is a binary file optimized for use with Tensorflow, and it helps in improving training time performances. Files converted

into this format can be efficiently read from disk and thereby constantly fed to the GPU without wasting time. Most of the times in fact, the input pipeline is the bottleneck of the training. fig. 6.1 shows a successful execution of a pipeline in Dataflow used to generate TFRecords from the Carla dataset. Each block shows the amount of time that would have been necessary to execute it without horizontal scaling. In total, to convert the Carla dataset into TFRecords, about 6 hours are needed, while using Dataflow we accomplished this in less than 30 minutes.

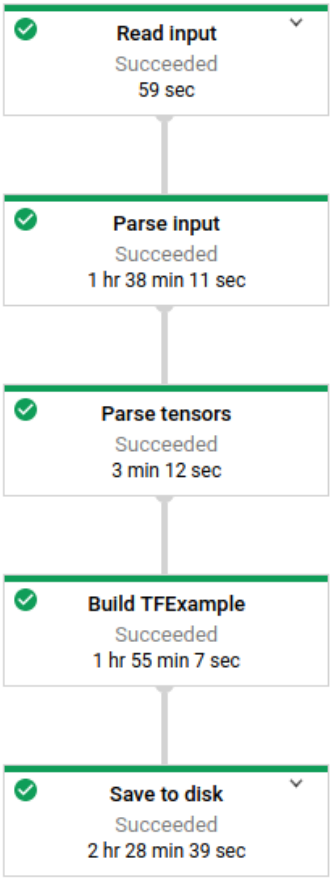


Figure 6.1: Example of a Beam pipeline executed in Dataflow

Chapter 7

Conclusions and future work

In this work we first introduced some of the most important techniques to perform domain adaptation. Then, we analyzed thoroughly a recent general framework called AT/DT, that aims to explicitly use the correlation between visual tasks to perform domain adaptation. This method learns to transfer knowledge across tasks in a fully supervised domain and exploits this mapping on a different domain where only partial supervision is available. Inspired by the domain adaptation literature, we proposed some effective upgrades that can be applied to the framework, and evaluated them using monocular depth estimation as source task and semantic segmentation as target task. We started with two ablation studies to find a good compromise between performances and memory requirements. These allowed us to modify the architecture of the transfer network, which highly affects the transferability of features among tasks. In particular, the most important factor is to maintain spatial information when learning such mapping. Afterward, by exploiting adversarial training, we were able to further improve performances. Lastly, we studied the possibility to transfer features from popular Self-supervised tasks to semantic segmentation, showing that an effective mapping is learned only when the two tasks are strongly connected. In future work, we plan to apply our augmented framework on other popular datasets to make a comparison with advanced domain adaptation techniques. Finally, we would like to test our framework with other important vision tasks.

Bibliography

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] H. Alhaija, S. Mustikovela, L. Mescheder, A. Geiger, and C. Rother. Augmented reality meets computer vision: Efficient data generation for urban driving scenes. *International Journal of Computer Vision (IJCV)*, 2018.

[3] K. Bousmalis, N. Silberman, D. Dohan, D. Erhan, and D. Krishnan. Unsupervised pixel-level domain adaptation with generative adversarial networks. *CoRR*, abs/1612.05424, 2016.

[4] K. Bousmalis, G. Trigeorgis, N. Silberman, D. Krishnan, and D. Erhan. Domain separation networks. *CoRR*, abs/1608.06019, 2016.

[5] J. Canny. A computational approach to edge-detection. *Ieee transactions on pattern analysis and machine intelligence*, 8(6):679–698, Nov 1986.

[6] L. Chen, G. Papandreou, F. Schroff, and H. Adam. Rethinking atrous convolution for semantic image segmentation. *CoRR*, abs/1706.05587, 2017.

[7] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele. The cityscapes dataset for semantic urban scene understanding. *CoRR*, abs/1604.01685, 2016.

[8] J. S. Denker, W. R. Gardner, H. P. Graf, D. Henderson, R. E. Howard, W. Hubbard, L. D. Jackel, H. S. Baird, and I. Guyon. Neural network recognizer for handwritten zip code digits. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 1*, pages 323–331. Morgan-Kaufmann, 1989.

[9] A. Dosovitskiy, G. Ros, F. Codevilla, A. López, and V. Koltun. CARLA: an open urban driving simulator. *CoRR*, abs/1711.03938, 2017.

[10] D. Eigen, C. Puhrsch, and R. Fergus. Depth map prediction from a single image using a multi-scale deep network. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2366–2374. Curran Associates, Inc., 2014.

[11] Y. Ganin, E. Ustinova, H. Ajakan, P. Germain, H. Larochelle, F. Laviolette, M. Marchand, and V. Lempitsky. Domain-adversarial training of neural networks. *J. Mach. Learn. Res.*, 17(1):2096–2030, Jan. 2016.

[12] M. Ghifary, W. B. Kleijn, and M. Zhang. Domain adaptive neural networks for object recognition. *CoRR*, abs/1409.6041, 2014.

[13] M. Ghifary, W. B. Kleijn, M. Zhang, D. Balduzzi, and W. Li. Deep reconstruction-classification networks for unsupervised domain adaptation. *CoRR*, abs/1607.03516, 2016.

[14] S. Gidaris, P. Singh, and N. Komodakis. Unsupervised representation learning by predicting image rotations. *CoRR*, abs/1803.07728, 2018.

[15] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, pages 2672–2680, Cambridge, MA, USA, 2014. MIT Press.

[16] G. Hinton, O. Vinyals, and J. Dean. Distilling the knowledge in a neural network. In *NIPS Deep Learning and Representation Learning Workshop*, 2015.

[17] H. HirschmÃ$\frac{1}{4}$ller. Stereo processing by semi-global matching and mutual information. *in IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30:328–341, 02 2008.

[18] J. Hoffman, E. Tzeng, T. Park, J. Zhu, P. Isola, K. Saenko, A. A. Efros, and T. Darrell. Cycada: Cycle-consistent adversarial domain adaptation. *CoRR*, abs/1711.03213, 2017.

[19] J. Hu, J. Lu, Y.-P. Tan, and J. Zhou. Deep transfer metric learning. *Trans. Img. Proc.*, 25(12):5576–5588, Dec. 2016.

[20] D. H. Juárez, L. Schneider, A. Espinosa, D. Vázquez, A. M. López, U. Franke, M. Pollefeys, and J. C. Moure. Slanted stixels: Representing san francisco's steepest streets. *CoRR*, abs/1707.05397, 2017.

[21] G. Kang, L. Jiang, Y. Yang, and A. G. Hauptmann. Contrastive adaptation network for unsupervised domain adaptation. *CoRR*, abs/1901.00976, 2019.

[22] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[23] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, volume 86, pages 2278–2324, 1998.

[24] Y. Li, N. Wang, J. Shi, J. Liu, and X. Hou. Revisiting batch normalization for practical domain adaptation. *CoRR*, abs/1603.04779, 2016.

[25] M. Liu and O. Tuzel. Coupled generative adversarial networks. *CoRR*, abs/1606.07536, 2016.

[26] T. Miyato, T. Kataoka, M. Koyama, and Y. Yoshida. Spectral normalization for generative adversarial networks. *CoRR*, abs/1802.05957, 2018.

[27] Z. Murez, S. Kolouri, D. J. Kriegman, R. Ramamoorthi, and K. Kim. Image to image translation for domain adaptation. *CoRR*, abs/1712.00479, 2017.

[28] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng. Reading digits in natural images with unsupervised feature learning. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning 2011*, 2011.

[29] S. J. Pan and Q. Yang. A survey on transfer learning. *IEEE Trans. on Knowl. and Data Eng.*, 22(10):1345–1359, Oct. 2010.

[30] P. Z. Ramirez, A. Tonioni, S. Salti, and L. di Stefano. Learning across tasks and domains. *CoRR*, abs/1904.04744, 2019.

[31] A. S. Razavian, H. Azizpour, J. Sullivan, and S. Carlsson. CNN features off-the-shelf: an astounding baseline for recognition. *CoRR*, abs/1403.6382, 2014.

[32] O. Ronneberger, P. Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation. *CoRR*, abs/1505.04597, 2015.

[33] A. Rozantsev, M. Salzmann, and P. Fua. Beyond sharing weights for deep domain adaptation. *CoRR*, abs/1603.06432, 2016.

[34] K. Saenko, B. Kulis, M. Fritz, and T. Darrell. Adapting visual category models to new domains. In K. Daniilidis, P. Maragos, and N. Paragios, editors, *ECCV (4)*, volume 6314 of *Lecture Notes in Computer Science*, pages 213–226. Springer, 2010.

[35] E. Tzeng, J. Hoffman, T. Darrell, and K. Saenko. Simultaneous deep transfer across domains and tasks. *CoRR*, abs/1510.02192, 2015.

[36] E. Tzeng, J. Hoffman, K. Saenko, and T. Darrell. Adversarial discriminative domain adaptation. *CoRR*, abs/1702.05464, 2017.

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

[38] M. Wang and W. Deng. Deep visual domain adaptation: A survey. *CoRR*, abs/1802.03601, 2018.

[39] J. Xu, L. Xiao, and A. M. López. Self-supervised domain adaptation for computer vision tasks. *CoRR*, abs/1907.10915, 2019.

[40] Y. yan Sun, E. Tzeng, T. Darrell, and A. A. Efros. Unsupervised domain adaptation through self-supervision. *ArXiv*, abs/1909.11825, 2019.

[41] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson. How transferable are features in deep neural networks? *CoRR*, abs/1411.1792, 2014.

[42] F. Yu, V. Koltun, and T. A. Funkhouser. Dilated residual networks. *CoRR*, abs/1705.09914, 2017.

[43] P. Zama Ramirez, A. Tonioni, S. Salti, and L. Di Stefano. Learning across and domains. In *International Conference on Computer Vision (ICCV)*, 2019.

[44] A. R. Zamir, A. Sax, W. B. Shen, L. J. Guibas, J. Malik, and S. Savarese. Taskonomy: Disentangling task transfer learning. *CoRR*, abs/1804.08328, 2018.