

École polytechnique de Louvain

Deep Visual Domain Adaptation Applied to Traffic Sign Detection

Author: **Aurélien BOMBO**

Supervisors: **Marco SAERENS, Arnaud JACQUES, Damien FOURURE**

Readers: **Siegfried NIJSSEN, Pierre SCHAUS**

Academic year 2019–2020

Master [120] in Computer Science

Abstract

Most machine learning problems assume that past training data follows the same distribution as future test data. However, this assumption often does not hold, such that the so-called source domain, from which a model learns, is distributed differently from the so-called target domain, which contains the data on which predictions are made. Hence, the learning algorithm fails to generalize. Domain adaptation has been proposed to address such cases, and aims to improve the performance of a model on the target domain using the knowledge learned in the source domain. In this work, we use domain adaptation to solve a traffic sign detection task with images taken under different conditions. For this purpose, we apply an adversarial training technique that is able to integrate into a deep learning algorithm and learn a domain-invariant representation of the input. We train it in an unsupervised fashion, such that only the ground-truth labels for the source domain are revealed, while those of the target domain are hidden. We show that the resulting adaptive model is effective in learning a detector that performs better on the target domain than a non-adaptive model trained solely on the source domain.

Keywords: domain adaptation, transfer learning, machine learning, deep learning, computer vision, object detection, adversarial learning, traffic sign.

Contents

List of Figures	v
List of Tables	vi
List of Symbols	vii
1 Introduction	1
1.1 Research Questions	2
1.2 Thesis Structure	3
1.3 EURA NOVA	4
2 Theoretical Basis	6
2.1 Machine Learning	6
2.1.1 Task	6
2.1.2 Performance Measure	7
2.1.3 Experience	7
2.2 Artificial Neural Networks	8
2.2.1 Activation Units	9
2.2.2 Training	10
2.3 Convolutional Neural Networks	11
2.3.1 Convolution	11
2.3.2 Pooling	12
2.4 Object Detection	12
2.4.1 YOLO: You Only Look Once	13
2.4.2 Performance Measure	15
2.5 Transfer Learning	18
2.5.1 Formalisms	18
2.5.2 Categorization	18
2.5.3 Domain Adaptation	19
2.5.4 Subspace Alignment	21
3 Data Collection and Preprocessing	24
3.1 Data Collection	24
3.2 Data Annotation	25
3.2.1 Inter-Rater Agreement	26

3.3	Data Preparation	27
3.3.1	Inter-Annotator Disagreement Reconciliation	27
3.3.2	Study of the Domain Shift	29
3.3.3	Train-Validation-Test Set Split	39
4	Baseline Models	42
4.1	Non-Adaptive Baseline	42
4.1.1	Training	42
4.1.2	Results	46
4.2	Adaptive Baseline	48
5	DANN: Domain-Adversarial Neural Networks	49
5.1	Model	50
5.2	Training	51
5.3	Gradient Reversal Layer	52
5.4	Application to Traffic Sign Classification	54
5.4.1	Baselines	54
5.4.2	Experimental Setup	55
5.4.3	Results	56
6	Experimentation	58
6.1	Setup	58
6.1.1	Retrofitting a Domain Classifier in YOLOv3	58
6.1.2	Domain Classifier Architecture	60
6.1.3	Training from Scratch vs. Fine-Tuning	61
6.1.4	Domain Regularizer	61
6.1.5	Hyperparameter Tuning	61
6.2	Results	63
6.2.1	Initial Model	63
6.2.2	Tuning the Initial Model	64
6.2.3	Fine-Tuning the Source-Only Model	64
6.2.4	Boosting the Domain Classifier	65
6.2.5	Multiple Domain Classifiers	66
6.2.6	Other Experiments	67
7	Conclusion	69
7.1	Future Work	70
A	Contingency Table Discussion	72
	Bibliography	75

List of Figures

2.1	Architecture of YOLOv3	13
2.2	Examples of different postprocessing steps employed by YOLOv3	14
2.3	Calculation of average precision using the precision-recall curve	17
2.4	Example of covariate shift	20
2.5	Example of subspace mapping	21
3.1	Uncropped examples of the images collected	25
3.2	Examples of ground-truth annotations	26
3.3	Examples of ground-truth annotations lacking specificity	29
3.4	Examples of ground-truth annotations in different lighting conditions	31
3.5	Examples of equally sharp images where our sharpness metric fails	33
3.6	Histogram of the number of images across the months of the year	34
3.7	Performance gap at different solar altitude thresholds	37
3.8	Performance under and over different solar altitude thresholds	38
3.9	Examples of images at different solar altitudes	39
3.10	Color-coded performance plot for different solar altitudes across seasons	40
4.1	Typical learning curves under a two-stage learning process	44
4.2	Examples of oversized ground-truth annotations	46
4.3	Examples of well-fit predicted bounding boxes	47
5.1	Architecture of DANN	50
5.2	Examples of the traffic sign classes of the GSTRB dataset	54
5.3	Architecture of DANN applied to the GSTRB dataset	55
6.1	Our DANN architecture with one domain classifier	59
6.2	Our DANN architecture with three domain classifiers	60
6.3	Our DANN architecture with a domain classifier directly after Darknet	67

List of Tables

2.1	Performance of different object detection algorithms	17
3.1	Distribution of the number of unique annotators per image	27
3.2	Distribution of the number of images across twilights	36
3.3	Distribution of the final split across domains and training sets	41
5.1	DANN results in image classification	56
6.1	Summary of experimental results	66
A.1	Contingency table comparing different training and test sets	72

List of Symbols

a	Italic — scalars
$\mathbf{a}$	Bold — vectors
$\mathbf{A}$	Capital bold — matrices
$\mathcal{D}$	Domain
$\mathcal{T}$	Task
$\mathcal{P}$	Performance metric
N	Number of examples
D	Number of features [†]
C	Number of class labels
$\mathcal{X}$	D -dimensional feature space
$\mathcal{Y}$	C -dimensional class label space
$\mathbf{x}$	D -dimensional feature vector representing an example
$\mathbf{y}$	Empirical C -dimensional class label vector associated to an example
$\hat{\mathbf{y}}$	Predicted C -dimensional class label vector associated to an example
d	Binary domain label associated to an example [†]
$\mathcal{L}$	Loss function
$\mathcal{I}$	Set of images
$\mathcal{B}$	Set of bounding boxes

[†]Unless otherwise specified.

1 Introduction

With most machine learning problems, the assumption is made that the distribution of the source examples used to train a model (so-called source domain) is the same as that of the target examples on which predictions are made (so-called target domain). However, in many instances, this assumption does not hold, and the predictive power of the model suffers. This is because the features are domain-dependent, and as such, the model fails to generalize.

This issue is particularly problematic in the case of object detection. In the field of computer vision, object detection is the task of locating and classifying instances of semantic objects in an image [1]. In this task, factors such as illumination, pose, or image quality affect the ability of a model to make truthful predictions [2, 3].

Domain adaptation refers to the set of techniques used to deal with such situations, and aims at compensating for the distribution shift between the training set and the test set. In particular, with the renewed interest for artificial neural networks among the scientific community, techniques have been devised which leverage deep neural networks to learn more transferable representations by embedding domain adaptation in the deep learning process. Such techniques are referred to using the term “deep domain adaptation” [2, 3].

At EURA NOVA, a research and consultancy company active in artificial intelligence and big data, one of the research projects pertains to traffic sign detection. In Belgium, some municipalities do not possess an exhaustive inventory of their traffic signs. Even more importantly, they would like to be able to know which of their traffic signs are in poor condition (i.e., damaged, discolored, dirty, etc.) and need service.

To this end, the national postal service bpost partnered with EURA NOVA. Using a camera installed inside a delivery car, and retrofitted with a GPS receiver, they are able to capture anonymized images of the municipalities’ streets, and cross-reference the detected traffic signs with the location derived from the GPS device to build a traffic sign map.

To extract traffic signs from the collected images, bpost employees are leveraged to annotate past data, and an object detection model is trained on these annotations. However, issues arise due to the method of collection of the data, that lead to a natural split of the images: those of good quality (the source domain) and those of poor quality (the target domain).

The main factor is related to the delivery patterns of a postal service: delivery rounds often start before sunrise; as such, some images come out dark and noisy. Similarly, mist, rain, and difficult weather conditions in general have an effect on how lit the area appears through the lens. Finally, the speed at which the vehicle is moving may induce motion blur.

Domain adaptation has been proposed to attend to these issues [2, 3]. In this thesis, we seek to implement a domain adaptation algorithm that is able to accurately detect the traffic signs in our target dataset using knowledge from the source data.

1.1 Research Questions

As is clear from the introduction, there are different facets to the problem at hand. Here, we clearly identify the three research questions that we aim to answer in this thesis.

What is the domain shift in our data?

This first question is sort of a meta-question which, once answered, will allow us to fully dive into the meat of the problem. We have made the assumption that there exists a domain shift in our dataset, such that the target set contains images that are harder to predict than those in the source set. Typically, this domain shift is known a priori: for example, one may use video game screenshots to learn better representations for real-life pictures [4]. In that sense, the domain shift is known prior to the data collection process. In this work, this was not the case: only after the images were collected did we notice the emergence of a domain shift, which we propose to address with domain adaptation.

This domain shift is due to the data collection process: images are collected at different times of the day (thus some appear darker) and the camera is located inside a moving vehicle (thus some images are subject to motion blur). However, we must identify an exact set of criteria and thresholds on which to establish a source-target split. It is necessary in order to determine whether a baseline model effectively struggles to process target images compared to source images, and thus if domain adaptation is needed. Furthermore, this may inform our choices of domain adaptation approaches, as some approaches may be better-suited than others for specific kinds of domain shifts. This question is addressed in Section 3.3.2.

Can domain adaptation yield better results than non-adaptive machine learning for traffic sign detection?

Recall that, from an applicative point of view, one main goal is to accurately detect traffic signs in an image. Assuming the existence of two different but related source and target datasets, the particular aim of this question is only to conclude whether any domain-adaptive algorithm is able to improve the performance of a baseline, non-adaptive model on the *target* dataset, using knowledge from the source dataset.

Importantly, this implies that we do not concern ourselves with how a domain-adaptive model performs on the source dataset. The reason is that, in the worst case that such a

model performs worse than our baseline on the source data, we can simply take the decision to always fall back to the baseline model for such data.

Furthermore, we seek to evaluate how exactly a domain-adaptive model compares and if it could be used in practice. To do so, we consider different baselines, such as non-adaptive models solely trained either on source or target data (see Chapter 6).

Is deep domain adaptation more performant than shallow adaptation approaches?

As popular as deep learning algorithms are, especially for object detection [5, 6], they all suffer significant pitfalls [7]. Among those, neural networks are notoriously hard to optimize: they are prone to overfitting and require careful regularization in order to generalize properly. Paradoxically, reaching convergence is sometimes difficult as hyperparameters are tricky to tweak. Finally, since neural networks can contain billions of parameters [8], training them is extremely resource-intensive and time-consuming.

In light of these issues making neural network training a complex problem, it is clear that there is value in first evaluating a shallow domain adaptation solution: such a solution could be much easier to implement (compared to an alternative deep method) and still provide an improvement over a baseline model trained solely on the source data. This would represent a quick win from an applicative perspective, as well as an interesting theoretical result. We address this question in Chapter 6.

1.2 Thesis Structure

This work is structured as follows:

This chapter (Chapter 1) introduces the rest of thesis by laying out the general ideas behind it and the problem it aims to solve. It does not yet go into detail about any of the proposed solutions.

In Chapter 2, we conduct a review of the literature regarding machine learning concepts, including artificial neural networks and convolutional neural networks. We then introduce the task of object detection and describe the YOLOv3 [9] algorithm, a state-of-the-art approach in object detection and a significant basis for our work.¹ We conclude by presenting formalisms for domain adaptation, while drawing parallels to the overall field of transfer learning.

Chapter 3 discusses practical aspects common to most machine learning projects, such as data collection and preparation. In particular, we detail how we established a split of our dataset into source and target data.

¹We do not consider YOLOv4 [10] as its preprint was released only near the end of this thesis.

In Chapter 4, we describe the implementation of simple non-adaptive and adaptive baseline models that will be used to benchmark the performance of a more powerful deep domain adaptation model.

In Chapter 5, we present a deep domain adaptation approach that leverages an adversarial neural network as well as a novel neural layer design to solve the adaptation problem. It works by learning features that are both discriminative for the main learning task and invariant to domain shift.

Chapter 6 is the subject of a discussion regarding the implementation of the method described in the preceding chapter. In that chapter, we detail our experimental setup as well as the results we obtained.

Finally, Chapter 7 concludes our thesis and touches on potential future work.

1.3 EURA NOVA

This thesis was conducted during a 13-week internship at EURA NOVA [11, 12], a Belgian technology company and independent research center founded in 2008. On top of its Belgian presence, it has satellite offices located in Tunisia and France. The main goal of the company is to help other organizations in achieving their digital transformation, leveraging its expertise in domains ranging from software architecture to deep learning through data management. Here, we present the three main activities of the company.

First, EURA NOVA supports companies by working alongside them and providing them with support and expertise. It employs consultants that directly work with companies across various industries (e.g., transportation, banking, communication, and more). Their missions spark from the needs of each project and the skills required; they are data scientists, software architects, project managers, researchers, and so on.

If consulting can help companies with specific aspects of their digital transformation, nearly every company shares a common issue: data management. Based on this observation, EURA NOVA provides a managed end-to-end solution called *digazu* [13] for companies to manage and leverage the data they produce. At the center of *digazu* lies a data lake based on Hadoop [14] to allow easy collection and storage of data from different parts of the company. It also allows companies to build their own pipelines, leveraging tools such as Flink [15] and Kafka [16]. Furthermore, it integrates with Jupyter Notebook [17] and Kubeflow [18] for easy prototyping and deployment.

Finally, research and development is centric to the company, in a bid to provide both internal and external projects with state-of-the-art technology. The *digazu* project is an obvious first example of this mindset. Similarly, the company recently presented a formal specification framework for event processing [19] at the 13th ACM Conference on Distributed and Event-Based Systems. At the 16th International Symposium on Neural Networks, it presented a novel loss function for neural topic models [20] that “penalizes incoherent topics during

training.” The company also presented GraphOpt, a framework for automatic parameter tuning of graph processing frameworks [21], at the 3rd IEEE International Workshop on Benchmarking, Performance Tuning and Optimization for Big Data Applications.

In the next chapter, we present a literature review of different topics regarding machine learning, computer vision, and domain adaptation, necessary to the proper understanding of the rest of this work.

2 Theoretical Basis

This chapter introduces different concepts and formalisms necessary to the proper understanding of the rest of this thesis. We first give high-level concepts related to general machine learning, before diving into deep learning and convolutional networks. We then present the problem of object detection and its approaches. Finally, we close the chapter by introducing common formalisms in transfer learning and domain adaptation.

Note that the theoretical aspects introduced in Sections 2.1–2.3 are taken from [7].

2.1 Machine Learning

Since the advent of the computer, people have been wondering whether such a machine could become intelligent, for whatever arbitrary definition of intelligence there is. Today, the field of artificial intelligence is thriving. It is concerned with the study of intelligent agents: systems that act intelligently, so as to make the appropriate choices given the circumstances of their environment, and an endgoal [22].

Over the years, different approaches have been taken toward artificial intelligence. One such approach seeks to hardcode knowledge about the world in formal languages, so as to build a knowledge base [23]. However, human supervisors struggle to devise formal rules that are complex enough to describe the real world, strongly impacting the ability of such systems to make inferences. To overcome these limitations, researchers have come up with *learning algorithms* that have the ability to acquire their own knowledge by extracting patterns from raw data. This capability is known as *machine learning*.

In short, a machine learning algorithm is an algorithm that is able to learn from data. According to [24], “a computer program is said to learn from experience $\mathcal{E}$ with respect to a task $\mathcal{T}$ and a performance measure $\mathcal{P}$, if its performance at task $\mathcal{T}$ improves with experience $\mathcal{E}$, as measured by $\mathcal{P}$.”

2.1.1 Task

Machine learning tasks are typically described in terms of how the machine learning system should process a *learning example* (or simply “example”). An example is a collection of *features* that have been quantitatively measured from some object or event that we want the

machine learning system to process. We represent an example as a vector $\mathbf{x} \in \mathbb{R}^D$ where each entry x_i is a separate feature, among D total input features. For example, the features of an image are usually the values of the pixels in the image.

Many kinds of tasks can be solved with machine learning. Some of the most common machine learning tasks include *regression* and *classification*.

In regression, the algorithm predicts a scalar given an input. To solve this task, the algorithm outputs a function $f: \mathbb{R}^D \rightarrow \mathbb{R}$. An example of a regression task is the prediction of the expected revenue of an investment.

In classification, the algorithm specifies which of C categories the input belongs to. To solve this task, the algorithm typically produces a function $f: \mathbb{R}^D \rightarrow \{1, \dots, C\}$. When $\hat{y} = f(\mathbf{x})$, the model assigns an input described by the vector $\mathbf{x}$ to a category identified by the label y . An example of a classification task is object recognition, where the input is an image, and the output is an integer identifying the single object represented by the image.

Many others different use cases exist for machine learning, such as *object detection*, which we detail in Section 2.4.

2.1.2 Performance Measure

To evaluate the abilities of a machine learning algorithm, we must design a quantitative measure of its performance. This performance measure $\mathcal{P}$ is typically specific to the task $\mathcal{T}$ being carried out by the model.

For tasks such as classification, we often measure the *accuracy* of the model, which represents the proportion of examples for which the model produces the correct output.

Usually, we are interested in how well the machine learning algorithm performs on data that it has not seen before, since this determines how well it will work when deployed in the real world. This is the *generalization power* of the algorithm. We therefore evaluate these performance measures using a test set of data that is separate from the data used for training the model.

2.1.3 Experience

Machine learning algorithms can be broadly assigned to three categories, related to the kind of experience they are allowed to have during the learning process: *supervised*, *unsupervised*, and *semi-supervised*.

Supervised algorithms are trained with knowledge of the ground-truth labels for the entire dataset. Such algorithms thus work by learning how the input features influence the form of the output.

In the unsupervised case, the targets are unknown during training, roughly meaning that the

algorithm is learning with no reference frame. To work effectively, some such algorithms learn the probability distribution that generated the dataset. Others perform clustering, which consists in dividing the dataset into clusters of similar examples (which similarity measure is defined a priori).

Semi-supervised algorithms fall in the middle and typically train using a small amount of labeled and a large amount of unlabeled data.

2.2 Artificial Neural Networks

The performance of machine learning algorithms is heavily reliant on the *representation* of the input data, that is, how the features are chosen. However, simple algorithms such as linear regression are unable to influence this representation. For example, when linear regression is used to predict the price of a house, it is fed a fixed number of features such as its age, floor space, number of rooms, or the neighborhood it is in.

This is especially problematic when it is not obvious what features should be used. The house price example was fairly straightforward and one can easily think of new features to incorporate (e.g., whether it is detached, neighborhood, lot size, presence of a garage). However, when writing a program to detect faces in pictures, things get more complicated. We know that the basic facial features of a person are their eyes, nose, and mouth, but it is difficult to express what exactly those look like in terms of pixel values. Furthermore, the image could be complicated by the person wearing headwear, sunglasses, or even a face mask.

To address this problem, one solution consists in using machine learning to discover not only the mapping (from representation to output), but also the representation itself. This approach is known as *representation learning* and often leads to better results than can be obtained through hand-designed representation. An *artificial neural network* (ANN) leverages this approach by introducing gradually more complex representations, expressed in terms of simpler representations.

We focus here on the *feedforward* neural network, as it is the most widely-studied kind of ANN, as well as the only kind that we will use in the rest of this thesis. It is called feedforward because, at inference time, the information flows through the network from the input to the output, with no feedback connections.¹ Such networks are also called *multilayer perceptrons* (MLP), and because of their multilayer architecture, the discipline that studies neural networks is called *deep learning*. A neural network defines a mapping $\hat{\mathbf{y}} = f(\mathbf{x}; \boldsymbol{\theta})$ and learns the parameters $\boldsymbol{\theta}$ that best approximate some function $f^*: \mathbf{x} \rightarrow \mathbf{y}$.

At its most basic level, a neural network is thus represented by layers of perceptrons (also called “activation units”), each layer giving a different representation of the input, based on

¹Not to be confused with *backpropagation*, as discussed later, which is necessary for *learning*.

the earlier layers. The scalar output of a single perceptron can be written as

$$\hat{y} = f(\mathbf{x}; \theta) = f(\mathbf{x}; \mathbf{w}, b) = \mathbf{w}'\mathbf{x} + b, \quad (2.1)$$

where $\mathbf{w}$ represents the weight of each input $\mathbf{x}$ and b is the intercept.

In the following, we describe various concepts and design choices encountered in designing neural network architectures.

2.2.1 Activation Units

Before one starts thinking about adding layers to a neural network, one needs to define the *width* of the network, that is, the number of units of each layer. When a layer has D inputs and C outputs, we can write its equation as

$$\hat{y}_i = \mathbf{w}_i'\mathbf{x} + b, \quad (2.2)$$

$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{x} + \mathbf{b}, \quad (2.3)$$

where $\mathbf{w}_i$ represents the weight of each input, $\mathbf{W}$ is a $C \times D$ matrix whose rows are the weight vectors of the C units, and $\mathbf{b}$ contains the intercept for each unit. Note that each layer can have a different number of units, but for simplicity's sake, we assume it is fixed here.

So far, the perceptron model we have presented is a linear one. However, this limits the expressiveness of our model, as clearly, a neural network solely made up of linear units would remain a linear function of its input. As an example, this would prevent us from approximating a function as simple as the XOR operation. This is remedied using a supplemental nonlinear mapping called an *activation function*. Our perceptron is thus defined as

$$\hat{\mathbf{y}} = g(\mathbf{W}\mathbf{x} + \mathbf{b}), \quad (2.4)$$

where g is an activation function. Recently, a popular choice for such a function has been the rectified linear unit (or “ReLU”) [25, 26, 27] which is defined as $g(z) = \max\{0, z\}$.

With that said, the *depth* of the network, that is, the number of layers that it is made of, is also an important design choice. As mentioned previously, each layer depends on the output for the previous layer, with no feedback connections. As such, the network can be considered as a series of functions that are composed together. For example, we can define a three-layer network as

$$f(\mathbf{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{x}))), \quad (2.5)$$

where $f^{(1)}$ is the first layer, $f^{(2)}$ is the second one, and so on. The final layer is also called the *output layer*.

2.2.2 Training

When we use a feedforward neural network to accept an input $\mathbf{x}$ and produce an output $\hat{\mathbf{y}}$, information flows forward through the network. The input $\mathbf{x}$ provides the initial information that then propagates up to the hidden units at each layer and finally produces $\hat{\mathbf{y}}$. This is called *forward propagation*. During training, forward propagation can continue onward until it produces a scalar cost $\mathcal{L}(\boldsymbol{\theta})$. The *backpropagation* algorithm [28] allows the information from the cost to then flow backward through the network in order to compute the gradient $\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})$.

Calculating the gradient of a composition of functions can be done efficiently using the chain rule. Let x be a real number and let f and g be functions such that $f, g: \mathbb{R} \rightarrow \mathbb{R}$. Suppose that $y = g(x)$ and $z = f(g(x)) = f(y)$. Then the chain rule states that

$$\frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx}. \quad (2.6)$$

This is further generalizable to the vector case. Suppose that $\mathbf{x} \in \mathbb{R}^m$ and $\mathbf{y} \in \mathbb{R}^n$. If we have $g: \mathbb{R}^m \rightarrow \mathbb{R}^n$ and $f: \mathbb{R}^n \rightarrow \mathbb{R}$ such that $\mathbf{y} = g(\mathbf{x})$ and $z = f(\mathbf{y})$, then

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}. \quad (2.7)$$

In vector notation, we can rewrite this result by formulating the $\nabla_{\mathbf{x}} z$ gradient as the product of the Jacobian matrix $\partial \mathbf{y} / \partial \mathbf{x}$ by the gradient $\nabla_{\mathbf{y}} z$, such that

$$\nabla_{\mathbf{x}} z = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}} \right)' \nabla_{\mathbf{y}} z. \quad (2.8)$$

The backpropagation algorithm consists in performing such a computation for every activation unit in the network, so as to end up with a final gradient $\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})$.

Once we have obtained the gradient of the loss function, we can use an algorithm called *gradient descent* [29] to perform the actual optimization. Gradient descent is an iterative algorithm that seeks the minimum of a loss function $\mathcal{L}(\boldsymbol{\theta})$ by successively updating its parameters $\boldsymbol{\theta}$ until it reaches a stationary point. At each iteration, gradient descent performs the update

$$\boldsymbol{\theta}' = \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}), \quad (2.9)$$

where $\boldsymbol{\theta}'$ represents the new parameters and η is the *learning rate*: a positive scalar used to determine the size of a step, in such a way that gradient descent does not overshoot a stationary point.

2.3 Convolutional Neural Networks

A convolutional neural network [30] (also called “CNN”) is a specific kind of feedforward neural network used for processing input that has a grid-like topology. In particular, such an architecture can be used to process time-series data, and has become especially popular in the recent years for processing image data [31, 32, 33, 34]. The name indicates that the network uses a special kind of operation called a *convolution*, which is a linear operation that is used in lieu of matrix operation in some layers. In this section, we give a high-level explanation of this concept, as well as that of another widely-used operation called *pooling*.

2.3.1 Convolution

In its most general, convolution operates on two functions that take a real-valued argument. Let us first consider a simple example to motivate the need for such an operation.

Suppose that one is tracking the location of an object through space using a sensor. The position of the object at time t is given by $x(t)$. Suppose also that the output of the sensor is rather noisy, such that, to obtain a more accurate reading of the object’s position, we would like to average multiple measurements. As we consider more recent measurements to be more relevant, we actually want a weighted average that gives more weight to such measurements. As such, we define a weighting function $w(a)$, where a is the age of a measurement. If we apply such a weighted average operation at every moment, we obtain a new function s providing a smoothed estimate of the position of the object defined by

$$s(t) = \int x(a) w(t-a) da. \quad (2.10)$$

This is the operation referred to as *convolution*. We usually denote it with an asterisk, such that the above equation is equivalently written as

$$s(t) = (x * w)(t). \quad (2.11)$$

In convolutional network terminology, the first argument (i.e., x in this case) is usually referred to as the *input*, whereas the second one (i.e. w in this case) is called the *kernel*. Researchers also commonly refer to the output as the *feature map*.

In this example, we assume that the sensor provides measurements at every instant in time. However, this assumption is not realistic when working with computers, as data is most often provided in a discretized form. Instead, if we assume that x and w are defined on integer values, we can define the discrete convolution

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a) w(t-a). \quad (2.12)$$

Furthermore, in machine learning applications for image processing, the input is usually a tensor (i.e., a multidimensional array) of pixel values, while the kernel is a tensor of param-

eters. With this in mind, it makes sense to define a convolution operation able to operate over more than one axis at a time. For an image I of size $m \times n$ and a two-dimensional kernel K , we define convolution as the function $S(i, j)$, such that

$$S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(m, n) K(i - m, j - n). \quad (2.13)$$

The use of convolution is motivated by three main ideas. First, convolution enables *sparse connectivity*: as opposed to traditional neural networks where every output interacts with every input, CNNs can leverage kernels that are smaller than the input. This reduces the memory footprint of the network and means that computing the output requires fewer operations. Second, the same kernel (as well as its parameters) is reused over different parts of the input. This is referred to as *parameter sharing* and further improves the efficiency of CNNs. Finally, convolution has a property called *translation equivariance*, meaning that a translation of the inputs results in an equivalent translation of the convolved outputs. In image processing, this helps the network to generalize feature detection in different locations of the image.

2.3.2 Pooling

Convolutional layers can generally be considered as having two stages. In the first stage, a number of convolutions are computed in parallel to obtain a set of linear activations. Secondly, each linear activation is fed to a nonlinear activation function (e.g., ReLU), sometimes referred to as the *detector stage*.

Pooling adds a third stage to this framework, by using a *pooling function*, which augments the output of the network by replacing it with a summary statistic of the nearby outputs. An example of such a function is the *max pooling* operation [35], which reports the maximum output within a rectangular neighborhood. Other popular pooling functions include the average of a rectangular neighborhood (i.e., average pooling), the L2 norm of a rectangular neighborhood, or a weighted average based on the distance from the central pixel.

In all cases, pooling helps to make the representation approximately *invariant* to small translations of the input. *Translation variance* (as opposed to translation equivariance induced by convolution) means that a translation of the inputs by a small amount does not result in a translation for most outputs. This property is useful when we care more about the presence of a feature than its exact location (e.g., if we detect two eyes within close proximity of each, the image probably represents a face).

2.4 Object Detection

Object detection designates the task of locating and identifying instances of semantic objects of certain classes (e.g., traffic signs, people, cars, etc.) in images or videos [1]. It has

been the subject of much research in the last decades, and especially in the last ten years, as deep learning and convolutional networks in particular have led to significant performance improvements [36, 6, 5].

As one of the fundamental problems of computer vision, object detection forms the basis for many other computer vision tasks, such as instance segmentation [37], scene understanding [38], or object tracking [39]. Likewise, it supports a wide range of real-world applications, including autonomous driving, robotic vision, video surveillance, content-based image retrieval, or augmented reality.

In this section, we describe YOLOv3 [9], a recent yet popular object detection approach. We then detail average precision, an important metric used to measure the performance of object detection algorithms.

2.4.1 YOLO: You Only Look Once

YOLOv3 (You Only Look Once) [9] is an object detection algorithm that uses only convolutional layers, making it a fully convolutional network (see Figure 2.1). As the name suggests, it is the third evolution of the YOLO family of algorithms [40, 41]. A distinguishing characteristic of YOLO is that it targets real-time processing speed with state-of-the-art performance.

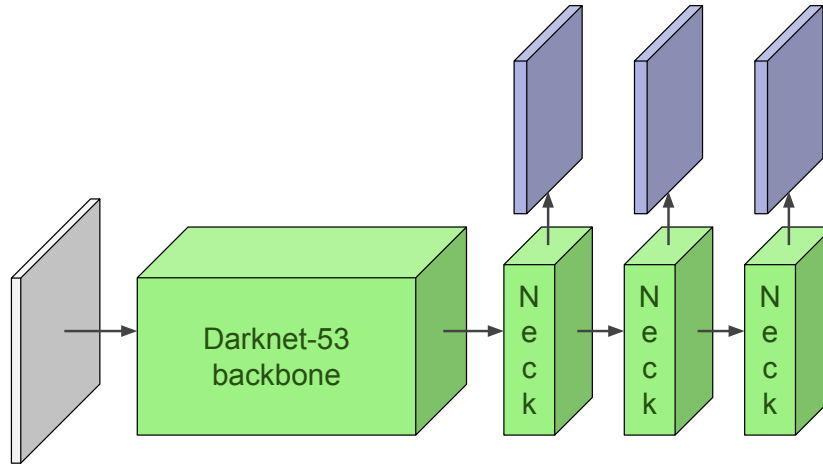


Figure 2.1: The architecture of YOLOv3 [9]. The fully convolutional network takes an image as input and first passes it through a 53-layer feature extractor (i.e., the Darknet-53 backbone). The extracted features then get scaled to three different scales by the three necks, before being fed to the final box prediction heads (blue layers).

At a high level, YOLOv3 takes an $M \times M \times 3$ tensor representing a square RGB image, extracts features using the Darknet-53 architecture, and outputs a set of bounding box predictions

each represented by

$$(p_c, b_x, b_y, b_h, b_w, \mathbf{c}), \quad (2.14)$$

where for each bounding box:

- p_c is the objectness score, i.e., the probability that it is a semantic object, rather than background;
- (b_x, b_y) is the location of its upper-left corner;
- b_h and b_w are respectively the height and weight;
- $\mathbf{c}$ is a vector of size C , where c_k is the conditional probability that the object is of class k .

It is interesting to look at exactly how many bounding boxes are predicted. To do so, it is important to understand the concept of *anchors*. YOLOv3 considers the input image as an $S \times S$ grid of cells, where each cell is of size 32×32 and predicts one object. Moreover, each cell predicts B bounding boxes, specified by the anchors. Each anchor represents a bounding box with the same centroid as its cell, but with a different aspect ratio. This is because different object classes can have different aspect ratios (e.g., a person vs. a car). As such, YOLOv3 outputs a tensor of shape $S \times S \times B \times (5 + C)$.

However, this is not the full picture. As shown in Figure 2.1, YOLOv3 incorporates three prediction heads. Each of these operates on a different scale. That is, at YOLOv3's native image resolution of 416×416 , the prediction heads operate respectively on grids of sizes 13×13 , 26×26 , and 52×52 . This is done to boost the performance of the network on small objects, which was a common complaint with previous architectures [41].

Postprocessing

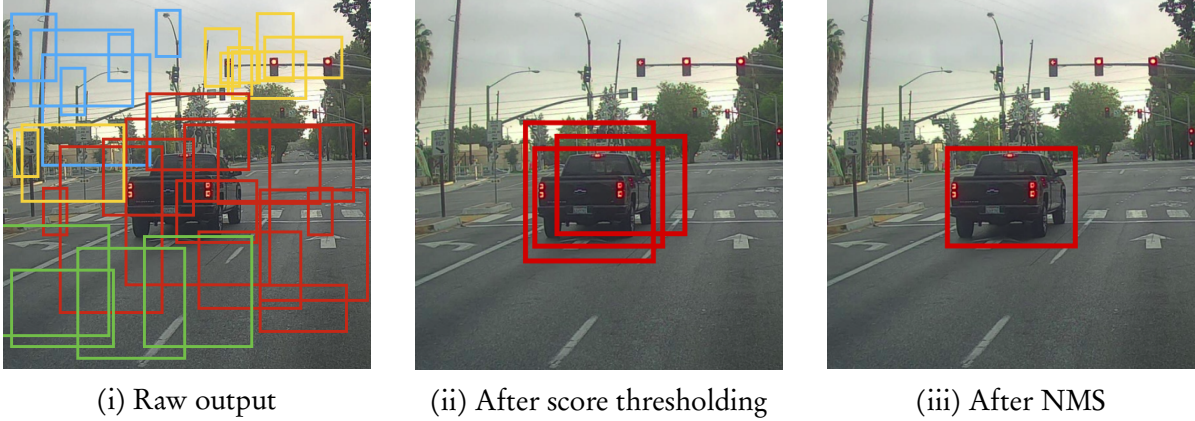


Figure 2.2: Examples of the results of different postprocessing steps employed by YOLOv3, courtesy of [42]. Different colors correspond to different object classes. In (ii), score thresholding got rid of the background boxes, as their confidence score was too low. In (iii), non-maximum suppression eliminates superfluous predictions.

From the preceding explanation, it should be clear that (i) YOLOv3 predicts a very large amount of bounding boxes for a single image, and (ii) many of those boxes overlap significantly and describe the same object.

Concerning the first point, YOLOv3 employs *score thresholding*: it calculates the confidence score of each predicted bounding box as $p_c \cdot c$ and only retains those whose score are above a certain threshold. This means that YOLOv3 only keeps predictions for which it has a certain degree of confidence (see Figure 2.2ii).

To explain how the algorithm eliminates superfluous overlapping predictions, it is necessary to first introduce the concept of *intersection-over-union* (IoU). IoU represents the degree to which two bounding boxes b_i and b_j intersect with each other, in relation to their total area. It is defined as

$$\text{IoU}(b_i, b_j) = \frac{b_i \cap b_j}{b_i \cup b_j}. \quad (2.15)$$

To eliminate bounding boxes that significantly overlap so as to obtain a minimal set of boxes that describe all objects in the image, YOLOv3 performs *non-maximum suppression* (NMS), which is a well-known technique in deep object detection [43]. At a high level, NMS works as follows: it looks at all predictions sorted by confidence score (as calculated above). For each prediction, it calculates its IoU with each other prediction. If one of the calculated IoU scores is over a threshold λ_{nms} (meaning that the two boxes overlap too much), it discards the lower-scoring prediction (see Algorithm 2.1). λ_{nms} is usually set to 0.7 [43]. Figure 2.2iii illustrates the importance of NMS.

Input: Predicted bounding boxes $\mathcal{B}$.
Output: Kept bounding boxes $\mathcal{B}$.
for $b_i \in \mathcal{B}$ **do**
 for $b_j \in \mathcal{B}$ **do**
 if $\text{IoU}(b_i, b_j) \geq \lambda_{\text{nms}}$ **then**
 if $\text{SCORE}(b_i) \geq \text{SCORE}(b_j)$ **then**
 $\mathcal{B} \leftarrow \mathcal{B} \setminus \{b_j\}$

Algorithm 2.1: Non-maximum suppression (NMS).

2.4.2 Performance Measure

Before discussing a more elaborate performance measure for object detection, it is important to first establish the basics. In classification tasks, there exists different metrics to measure the performance of a classifier.

As we touched on before, *accuracy* calculates the proportion of truthful predictions. However, it is a relatively naive metric as it is sensitive to class imbalance. Rather, it is more

interesting to look at two other metrics in conjunction. The first one is *precision*, which informs (in the two-class case) the proportion of positive predictions that are actually positive. Inversely, *recall* refers to the proportion of positive examples that were correctly labeled as such. More formally, these metrics are defined as

$$\text{accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}, \quad (2.16)$$

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}, \quad (2.17)$$

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}, \quad (2.18)$$

where TP and TN are the number of true positives and negatives, while FP and FN are the number of false positives and negatives.

As defined by the Pascal VOC object detection challenge [44], a positive prediction is defined as one that overlaps a ground truth with an IoU of at least 0.5. It follows that a negative prediction is any one that does not match this criterion. Furthermore, if multiple predictions refer to the same ground truth, only the first prediction is considered as positive, while the rest are considered negative.

On that basis, [44] defines a performance metric for object detection algorithms referred to as *average precision* (AP). Its name stems from the fact that it calculates the average precision at different recall levels (i.e., the precision-recall curve). However, a small twist is that instead of using the exact precision measure at each point of the curve, it uses an interpolated approximation: at each point, the precision considered is the one that maximizes the right side of the curve. This is done so as to “reduce the impact of ‘wiggles’ in the precision-recall curve caused by small variations in the ranking of predictions” [44]. This interpolated curve is more easily understood with Figure 2.3. Overall, to obtain good AP, an object detection algorithm must achieve good precision at all levels of recall; AP thus penalizes algorithms which retrieve only a subset of objects with good precision.

Therefore, AP is calculated as

$$\text{AP} = \sum_{i=1}^{n-1} (r_{i+1} - r_i) p_{\text{interp}}(r_{i+1}), \quad (2.19)$$

where n is the number of predictions, r_i is the i th recall measurement and p_{interp} is the interpolated precision defined as

$$p_{\text{interp}}(r_{i+1}) = \max_{\tilde{r} \geq r_{i+1}} p(\tilde{r}). \quad (2.20)$$

Average precision, as defined by PASCAL VOC [44], has been a reference performance metric for modern object detection algorithms [6, 5, 3]. More recently, however, another metric called *mean average precision* (mAP), introduced with the Microsoft COCO challenge [46], has gained in popularity [6, 5]. The apparent stuttering comes from the fact that instead of using a single IoU threshold of 0.5, it calculates the mean AP over ten different IoU thresh-

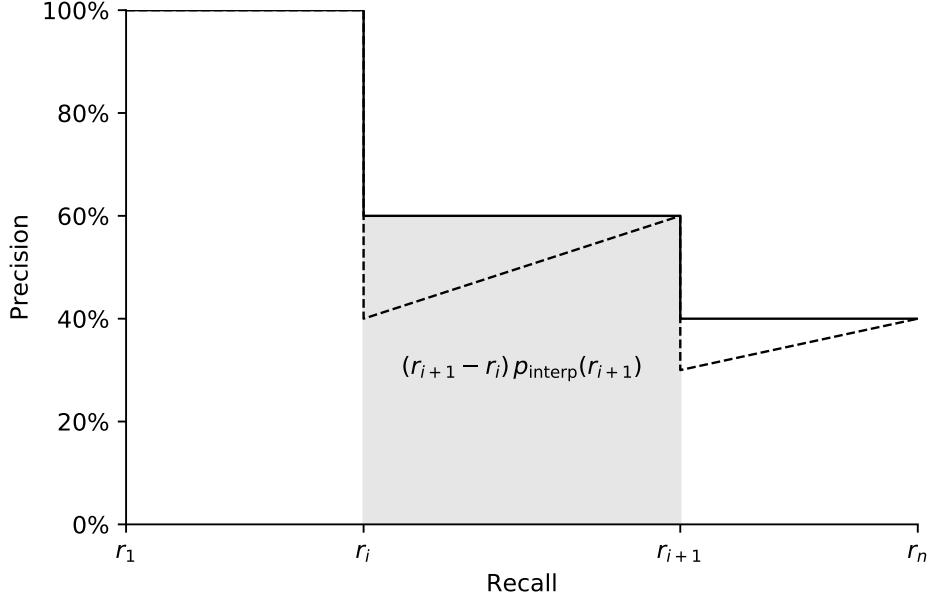


Figure 2.3: Illustration of the calculation of average precision (AP) using the precision-recall curve, adapted from [45]. The dashed line represents the original curve while the solid line represents the interpolated one.

olds (from 0.05 to 0.95, by steps of 0.1). The YOLOv3 authors note that mAP emphasizes better bounding boxes at the expense of classification accuracy [9]. Also do we only use AP in the rest of this thesis, as the quality of our ground-truth annotations is highly variable (see Chapter 3).

Table 2.1 shows the performance of three reference object detection techniques. In 2015, Faster R-CNN [43] won the COCO detection challenge, and remained the top performing algorithm for over a year. Even though YOLOv3 [9] was published three years later, it does not match Faster R-CNN’s performance, as YOLOv3 strongly focuses on speed and is able to process 30 frames per second (fps) [9], whereas Faster R-CNN only performs at 5 fps [43]. As of 2020, one state-of-the-art technique is Cascade Mask R-CNN [47], published in 2019. It is an extension of Faster R-CNN and is also able to output a contour mask for each detected object (instance segmentation).

	AP	mAP
Faster R-CNN [43]	59.1%	36.2%
YOLOv3 [9]	57.9%	33.0%
Cascade Mask R-CNN [47]	71.9%	53.3%

Table 2.1: Performance of different object detection algorithms on the COCO dataset [46].

2.5 Transfer Learning

Most learning algorithms (even recent deep learning approaches) make a major assumption: that past training and future test data are drawn from the same domain, such that the input feature space and data distribution characteristics are the same. When this assumption does not hold, the predictive power of a model can suffer [48]. However, obtaining further training data that share the same domain as the test data can be difficult and expensive. *Transfer learning* solves this problem by learning a target domain from a different but related source domain [49].

2.5.1 Formalisms

In this section, we introduce notations and definitions that we will use throughout the rest of this thesis. We start by giving the definition of a domain, and expand on our definition of a task given in Section 2.1.1. We mostly follow the formalisms used by [49] and [50].

A domain $\mathcal{D}$ consists of a feature space $\mathcal{X}$ and a marginal probability distribution $P(X)$, where $X = \{\mathbf{x}_1, \dots, \mathbf{x}_N\} \in \mathcal{X}$. X is a learning sample containing N vector-valued examples. Generally, if two domains are different, they may differ in terms of feature space, distribution, or both [49].

Given a domain $\mathcal{D} = \{\mathcal{X}, P(X)\}$, a task $\mathcal{T}$ consists of a label space $\mathcal{Y}$ and an predictive function $f(\cdot)$, such that $\mathcal{T} = \{\mathcal{Y}, f(\cdot)\}$. The predictive function is learned from the training data, denoted as $\{\mathbf{x}_i, y_i\}$ pairs, where $\mathbf{x}_i \in \mathcal{X}$ and $y_i \in \mathcal{Y}$.

In this thesis, we only consider the case where there are two domains: the *source domain* $\mathcal{D}_S$ is the one from which knowledge is transferred from, and the *target domain* $\mathcal{D}_T$ is the one to which we apply the transferred knowledge. In particular, we denote the source domain *data* as D_S , and the target domain *data* as D_T . Furthermore, we note that the size of the source sample N_S is usually much greater than that of the target sample N_T , such that $N_S \gg N_T$ [49].

Definition 2.1. Given a source domain $\mathcal{D}_S$ with a task $\mathcal{T}_S$, and target domain $\mathcal{D}_T$ with a task $\mathcal{T}_T$, *transfer learning* aims to help 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 $\mathcal{T}_S \neq \mathcal{T}_T$ [49].

As a domain is defined as a pair $\mathcal{D} = \{\mathcal{X}, P(X)\}$, the condition $\mathcal{D}_S \neq \mathcal{D}_T$ implies that $\mathcal{X}_S \neq \mathcal{X}_T$ or $P_S(X) \neq P_T(X)$. Similarly, with a task $\mathcal{T} = \{\mathcal{Y}, P(Y|X)\}$, the condition $\mathcal{T}_S \neq \mathcal{T}_T$ implies that $\mathcal{Y}_S \neq \mathcal{Y}_T$ or $P_S(Y|X) \neq P_T(Y|X)$. When both $\mathcal{D}_S = \mathcal{D}_T$ and $\mathcal{T}_S = \mathcal{T}_T$, the learning problem is a traditional machine learning problem.

2.5.2 Categorization

Based on Definition 2.1, [49] summarizes that the differences between different datasets can be expressed in terms of domain and/or task divergence. In this respect, [49] mentions the

following subsettings of transfer learning:

Inductive transfer learning. In this setting, the target task is different from the source task, no matter the similarity of the two domains ($\mathcal{T}_S \neq \mathcal{T}_T$). Then, some labeled data from the target domain is required to *induce* an predictive model $f_T(\cdot)$ for use in the target domain.

Transductive transfer learning. Here, the target task is the same as the source task, while the two domains are different but related ($\mathcal{T}_S = \mathcal{T}_T$ and $\mathcal{D}_S \neq \mathcal{D}_T$). In this case, the target domain is not labeled, while the source domain is.

Unsupervised transfer learning. This situation is similar to the inductive case and $\mathcal{T}_S \neq \mathcal{T}_T$. However, unsupervised transfer learning focuses on solving tasks such as clustering, dimensionality reduction, and density estimation [51, 52], for which there is no labeled data available in either the source or the target domain.

Note that in the literature, authors often use “transfer learning” to refer to inductive transfer learning in particular [53].

2.5.3 Domain Adaptation

Simply speaking, *domain adaptation* (DA) is a special case of transfer learning. Indeed, per [49, 50, 2, 54], domain adaptation is subject to the same assumptions as transductive transfer learning, and they are thus considered to be synonymous. In short, domain adaptation is transfer learning applied to a common task.

Definition 2.2. Given a source domain $\mathcal{D}_S$ with a task $\mathcal{T}_S$, and a target domain $\mathcal{D}_T$ with a task $\mathcal{T}_T$, *domain adaptation* aims 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 $\mathcal{D}_S \neq \mathcal{D}_T$ and $\mathcal{T}_S = \mathcal{T}_T$.

On this basis, [50, 2] split domain adaptation into two categories based on different domain divergence (i.e., distribution shift or feature space difference):

Homogeneous domain adaptation. In this category, the feature space of the target domain is the same as that of the source domain ($\mathcal{X}_S = \mathcal{X}_T$). Hence, the source and target datasets are different in terms of data distributions ($P_S(X) \neq P_T(X)$).

Heterogeneous domain adaptation. Here, we consider domains for which features spaces are different ($\mathcal{X}_S \neq \mathcal{X}_T$), with no particular regard for the data distributions.

Furthermore, [2] characterizes three levels of experience for domain adaptation methods. All levels give access to the ground-truth labels for the source data, and they only differ in terms of labels available for the target data. As such, *supervised DA* has access to a small amount of labeled target data, even though that amount is insufficient for traditional learning. On the other hand, *unsupervised DA* refers to datasets for which the target data is unlabeled. In the middle, *semi-supervised DA* has access to both labeled and unlabeled target data.

In the following, we describe two special cases of domain adaptation, as presented in [54]. These two cases are the focuses of the domain adaptation approaches that we discuss later.

Covariate Shift

Intuitively, [54] refers to *covariate shift* as a situation where the source distribution does not match the target distribution. It assumes, however, that the dependency of the response on the features in the source domain is the same as in the target domain. We give a more formal definition as follows.

Definition 2.3. Given a source marginal distribution $P_S(X)$ and a target marginal distribution $P_T(X)$, and given a source conditional distribution $P_S(Y|X)$ and a target condition distribution $P_T(Y|X)$, *covariate shift* describes a situation where $P_S(X) \neq P_T(X)$, while $P_S(Y|X) = P_T(Y|X)$.

Furthermore, [54] adds that covariate shift typically occurs when “the difficulty of picking an observation $\mathbf{x}_i$ with given features strongly impacts the probability of selecting an observation $(\mathbf{x}_i, y_i)$, thus making it impractical to replicate the target feature distribution $P_T(X)$ in the training set.” To illustrate, they give the example of a survey where some categories of individuals are more difficult to reach than others, even though the model should classify all categories equally well. Such a situation can be represented as in Figure 2.4. In Chapter 5, we present an approach to combat covariate shift in deep learning settings.

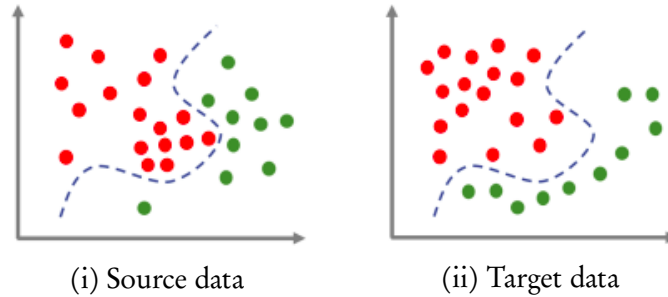


Figure 2.4: An example of covariate shift in two dimensions, courtesy of [54]. It shows that the distribution of the two classes (i.e., green and red) changes across domains. However, the dashed curve, representing the conditional probability $P(Y|X)$ is unchanged.

Subspace Mapping

[54] further discusses the case where, while the selection of examples in both the source and target domains is unbiased, there is a shift T between the features that describe those examples in the source domain, compared to the target domain. More formally, this assume

that for a *subspace mapping* problem, it holds that

$$P_S(X, Y) = P_T(T(X), Y). \quad (2.21)$$

We refer to such a situation as a *subspace misalignment*. If we decompose the preceding equation, we find that

$$P_S(X) = P_T(T(X)), \quad (2.22)$$

$$P_S(Y|X) = P_T(Y|T(X)). \quad (2.23)$$

As such, it is directly obvious that Equation (2.22) implies that $P_S(X) \neq P_T(X)$. Therefore, subspace alignment shares some similarities with covariate shift (see Definition 2.3), even though it further implies different conditional probability distributions. Figure 2.5 illustrates this special case. In Section 4.2, we present an approach to fix subspace misalignment using PCA.

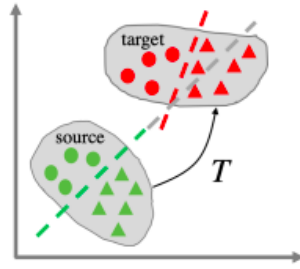


Figure 2.5: An example of a nonlinear subspace mapping from a source to a target dataset, courtesy of [54].

2.5.4 Subspace Alignment

In this subsection, we present a technique used to deal with the problem of subspace misalignment described precedingly. Introduced by [55], the technique considers one subspace (composed of d eigenvectors induced by principal component analysis, or PCA) for each domain and looks to reduce the discrepancy between the two domains by moving the source and target subspaces closer. This is achieved by optimizing a mapping function that transforms the source subspace into the target one. From this simple idea, a new domain adaptation technique is designed, based on *subspace alignment*.

The authors claims that the advantages of this technique are that (i) adapting the bases of the spaces makes it a global technique (i.e., it creates robust classifiers not subject to local perturbations) and (ii) aligning the source and target subspaces makes it a self-regularized technique (i.e., there is no need to tune further regularization parameters, as imposed by other optimization-based domain adaptation methods).

In the following, we assume that we have a source sample $\mathbf{S} \in \mathbb{R}^{N_S \times D}$ of labeled data and a target sample $\mathbf{T} \in \mathbb{R}^{N_T \times D}$ of unlabeled data, both lying in a given D -dimensional space and

drawn i.i.d. according to fixed but unknown source and target distributions $\mathcal{D}_S$ and $\mathcal{D}_T$. Note that N_S and N_T refer respectively to the number of source and target examples.

Subspace Generation

Even though both the source and target data lie in the same D -dimensional space, they have been drawn according to different marginal distributions. Consequently, rather than working on the original data themselves, it is suggested to handle more robust representations of the source and target domains and to learn the shift between these two domains.

First, we transform every source and target data in the form of a D -dimensional z-normalized vector (i.e., of zero mean and unit standard deviation). Then, we use PCA to select, for each domain, d eigenvectors corresponding to the d largest eigenvalues. These eigenvectors are used as bases of the source and target subspaces $\mathbf{X}_S$ and $\mathbf{X}_T$, with $\mathbf{X}_S, \mathbf{X}_T \in \mathbb{R}^{D \times d}$. [55] notes that $\mathbf{X}'_S$ and $\mathbf{X}'_T$ are orthonormal, such that $\mathbf{X}'_S \mathbf{X}_S = \mathbf{X}'_T \mathbf{X}_T = \mathbf{I}_d$, with $\mathbf{I}_d$ the identity matrix of size d .

Domain Adaptation

Two main strategies are used in subspace-based DA techniques. As in [56], the first one consists in projecting both source and target data to a common shared subspace. However, since this only exploits shared features in both domains, it is not always optimal. The second one aims to build a (potentially large) set of intermediate representations [57]. Beyond the fact that such a strategy can be costly, projecting the data to an intermediate common shared subspace may lead to information loss in both source and target domains.

As a result, [55] suggests to project the source sample $\mathbf{S}$ and the target sample $\mathbf{T}$ to their respective subspaces $\mathbf{X}_S$ and $\mathbf{X}_T$, through the operations $\mathbf{S}\mathbf{X}_S$ and $\mathbf{T}\mathbf{X}_T$, respectively. Then, a linear transformation function is learned that aligns the source subspace coordinate system to the target one. This step allows to directly compare source and target examples in their respective subspaces without unnecessary data projections. To achieve this task, basis vectors are aligned by using a transformation matrix $\mathbf{M} \in \mathbb{R}^{d \times d}$ from $\mathbf{X}_S$ to $\mathbf{X}_T$. $\mathbf{M}$ is learned by minimizing the following:

$$F(\mathbf{M}) = \|\mathbf{X}_S \mathbf{M} - \mathbf{X}_T\|_F^2 \quad (2.24)$$

where $\|\cdot\|_F^2$ is the Frobenius norm. We thus seek to find the optimal $\mathbf{M}^*$ such that

$$\mathbf{M}^* = \underset{\mathbf{M}}{\operatorname{argmin}} F(\mathbf{M}). \quad (2.25)$$

Since the Frobenius norm is invariant to orthonormal operations, and under our previous assumption that $\mathbf{X}'_S \mathbf{X}_S = \mathbf{I}_d$, Equation (2.24) can be rewritten as

$$F(\mathbf{M}) = \|\mathbf{X}'_S \mathbf{X}_S \mathbf{M} - \mathbf{X}'_S \mathbf{X}_T\|_F^2 = \|\mathbf{M} - \mathbf{X}'_S \mathbf{X}_T\|_F^2. \quad (2.26)$$

From this result, one can conclude that $\mathbf{M}^*$ is obtained as $\mathbf{X}'_S \mathbf{X}_T$. This implies that the new

coordinate system is equivalent to $\mathbf{X}_A = \mathbf{X}_S \mathbf{X}_S' \mathbf{X}_T$. [55] refers to $\mathbf{X}_A$ as the *target-aligned source coordinate system*. It is worth noting that if the source and target domains are the same, then $\mathbf{X}_S = \mathbf{X}_T$ and $\mathbf{M}^*$ is the identity matrix.

On that basis, Algorithm 2.2 gives the procedure to perform domain adaptation, where we project the source data onto the target-aligned source subspace via $\mathbf{X}_A$, and we project the target data into the target subspace via $\mathbf{X}_T$. In the end, a classifier is learned from the resulting projections, as well as from the source labels $\mathbf{y}_S$, to output the predicted target labels $\mathbf{y}_T$.

Input: Source data $\mathbf{S}$, target data $\mathbf{T}$, subspace dimension d , source labels $\mathbf{y}_S$.
Output: Predicted target labels $\mathbf{y}_T$.
 $\mathbf{X}_S \leftarrow \text{PCA}(\mathbf{S}, d)$
 $\mathbf{X}_T \leftarrow \text{PCA}(\mathbf{T}, d)$
 $\mathbf{X}_A \leftarrow \mathbf{X}_S \mathbf{X}_S' \mathbf{X}_T$
 $\mathbf{S}_A \leftarrow \mathbf{S} \mathbf{X}_A = \mathbf{S} \mathbf{X}_S \mathbf{X}_S' \mathbf{X}_T$
 $\mathbf{T}_T \leftarrow \mathbf{T} \mathbf{X}_T$
 $\mathbf{y}_T \leftarrow \text{CLASSIFIER}(\mathbf{S}_A, \mathbf{T}_T, \mathbf{y}_S)$

Algorithm 2.2: Unsupervised domain adaptation using subspace alignment [55].

Now that we have covered the theory required to understand the more complicated concepts developed later in this thesis, we move on to detailing the data collection and preparation aspects.

3 Data Collection and Preprocessing

This chapter gives more context and highlights the necessary preliminary steps required to solve our adaptation problem. We start by explaining the processes of data collection and annotation carried out by EURA NOVA, before going into detail about the steps required to prepare the data (i.e., dealing with multiple annotators and studying the domain shift in our data).

3.1 Data Collection

As always, the first step was data collection. Acquiring up-to-date real-life data is paramount to building an accurate traffic sign database. To this end, the Belgian national postal service bpost partnered with EURA NOVA. Thousands of bpost vans roam the streets of Belgium every day, delivering mail to its citizens. If one is able to properly leverage this network of delivery vans, it is possible to map a significant portion of Belgium’s traffic signs.

While the setup is still in its testing phase, we are able to share some details on how it works. EURA NOVA designed an electronic device made of a camera and a GPS receiver. The device has been installed in one bpost van, similar to the ones used for deliveries. It is placed on the dashboard, facing forward, and used to capture videos of the environment, just as the driver would see it (see Figure 3.1). The camera records continuously and outputs videos in 1920×1080 resolution. We worked with videos captured in the Flemish municipalities of Roeselare and Zonhoven, as well as in Brussels. Note that, for the sake of privacy, the individuals and vehicles present in the images acquired were automatically blurred out.

The dataset was built from these videos. Due to the extremely large quantity of data produced by the camera, it was necessary to sample frames from the videos. It was important to properly balance the sample rate, as under-sampling would lead to too few data points, and over-sampling would cause redundancy in the dataset.

A naive sampling strategy would be to take video frames at a fixed time interval. The main problem with this approach has to do with speed: more images would be captured when the vehicle is moving slowly, or simply stopped, than when it is moving at higher speeds. This would lead to imbalance and redundancy in the captured pictures.

Thanks to the GPS receiver attached to the camera, the video feeds are superseded with real-time location information. As such, one can use an improved strategy where the sampling rate is dependent on the position of the vehicle. The decision was made to sample video



Figure 3.1: Uncropped examples of the 31,761 total images collected.

frames at a five-meter interval, to strike a decent balance between gathering enough data and avoiding redundancy. In total, the final dataset contains 31,761 images.

3.2 Data Annotation

As the underlying problem of object detection is solved using a supervised learning algorithm, we also need to collect ground-truth information for our image dataset.

Obtaining ground-truth information is generally a costly process. In object detection, it involves having human annotators manually draw two-dimensional bounding boxes over the relevant objects (i.e., those that the model is supposed to detect), and assigning them a label.

It is crucial for the human-provided annotations to be as close to reality as possible. The reason is obvious: the model will use those annotations to learn what a traffic sign looks like. It follows that if the annotations provided are not of high quality (e.g., due to the operators annotating non-existent traffic signs, or drawing bounding boxes that are either too small or too large) the predictive power of the model will be negatively impacted.

For this process, EURA NOVA once again worked together with bpost and leveraged a number of employees of the postal service to act as annotators. First, EURA NOVA developed its own web application allowing end users to easily annotate traffic signs in images. The

application presents a user with a random sample of images from the dataset. For any image, the application asks the user a series of questions, which flow depend on the user’s answers to the previous questions. Figure 3.2 shows examples of annotations produced using the application.



Figure 3.2: Cropped examples of ground-truth image annotations provided by bpost employees. Note that, in the same image, different colors represent different annotators. Furthermore, the same color across different images does not necessarily represent the same annotator.

The first question that is asked to the user pertains to whether the current image contains at least one traffic sign. If the user answers negatively, the application skips over to the next image. Otherwise, the application allows them to draw the bounding box for each traffic sign. For each sign, the user is also invited to specify its type [58], as well as its state (e.g., dirty, discolored, damaged, etc.).

3.2.1 Inter-Rater Agreement

As mentioned previously, having high-quality annotations is crucial to the performance of the model on unseen data. Usually, in order to obtain such annotations, one would refer to domain experts (i.e., people familiar with the task at hand and able to produce annotations that are close to reality). Oftentimes, as well as in this project, it is not the case.

Even though the annotators were trained by EURA NOVA during workshops, we do not have specific knowledge of a particular annotator’s abilities to annotate traffic sign pictures properly. Although we do make the assumption that the annotators do their job conscientiously, it is naive to assume humans are inherently good at such a task, or that mistakes would not happen. For example, [59] showed that deep neural networks actually outperform humans in some image classification tasks. As such, measures have to be taken in order to mitigate such mistakes.

In statistics, *inter-rater agreement* describes the degree of agreement among raters. It is a score of “how much homogeneity or consensus exists in the ratings given by various judges” [60]. As such, obtaining redundant annotation data is helpful to this research in two ways. First, it allows us to estimate to which degree annotators generally agree with each other. Second,

and most importantly, it paves the way for allowing us to design a technique which estimates the best annotation for a given image by taking into account the opinions of multiple annotators. We describe such a technique in the next section.

While there are a plethora of methods to measure inter-rater agreement [61, 62, 63], during the data annotation process, there was no defined basis upon which to compute that score. As such, EURA NOVA made the proactive decision of designing the algorithm responsible for proposing users with images to annotate to make it so that each image which would be proposed to at least three unique annotators, as shown in Table 3.1.

Number of annotators	1	2	3	4	5	6
Frequency	68.9%	12.5%	18.5%	<0.1%	<0.1%	<0.1%

Table 3.1: The distribution of the number of unique annotators per image in our dataset of 31,761 images. Note that the majority of images have only been annotated once due to the fact that annotation was still ongoing at the time that the annotation data was collected.

3.3 Data Preparation

After the data has been collected and annotated by EURA NOVA and bpost comes the preparation phase. As such, this section reports novel contributions relating to i) the deduplication of the annotations, ii) the creation of the source-target domain split, and iii) the creation of the train-validation-test set split.

3.3.1 Inter-Annotator Disagreement Reconciliation

As mentioned in the previous section, one issue with our dataset is that it has duplication: some images have ground-truth data provided by more than one annotator. In machine learning, such a situation is problematic. The reason is simple: since those images appear multiple times in the dataset, as the model tries to learn a representation for the data, it will naturally give more weight to those repeated images and fail to generalize properly. This phenomenon is referred to as *algorithmic bias*.¹

To deal with this imbalance, a technique had to be devised to get rid of duplicates by reconciling the annotations of multiple annotators. It had to be such that, for any given image:

- (i) An object annotated by more than one person produces exactly one bounding box, this being the whole point of deduplication.

¹Not to be confused with *inductive bias*, which describes the set of a priori assumptions about the nature of the target function that are necessary for learning [24].

- (ii) The bounding box produced takes into account the information provided by all annotators, as we assume the work of all annotators to be equally valuable on average, and want to leverage it all.

Random annotator. An initial trivial solution would be to arbitrarily consider the annotation of one annotator at random for each image. However, this technique would be oversimplistic, as it ignores constraint (ii).

Average bounding box. A second technique can satisfy that last constraint. The idea is as follows: for each object in an image, retain only the bounding box which represents the average of all the bounding boxes provided by the annotators. We creatively name it the *average bounding box*. More formally, let a rectangular bounding box $\mathbf{b}^{(\cdot)} \in \mathbb{Z}^4$ be defined by

$$\mathbf{b}^{(\cdot)} = (x_1^{(\cdot)} \quad y_1^{(\cdot)} \quad x_2^{(\cdot)} \quad y_2^{(\cdot)}) \quad (3.1)$$

where the points $(x_1^{(\cdot)}, y_1^{(\cdot)})$ and $(x_2^{(\cdot)}, y_2^{(\cdot)})$ correspond respectively to the locations of the upper-left and lower-right corners of the bounding box. Let $\mathcal{B}$ be the set of all bounding box annotations for a given object. Then the average bounding box $\mathbf{b}^{(\text{avg})}$ is given by

$$\mathbf{b}^{(\text{avg})} = \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \mathbf{b}^{(i)}. \quad (3.2)$$

However, an issue arises with this method, related to the way that some operators annotate images. Among many others, the dataset contains images showing traffic signs of types F29, F31, and F33a [58]. In real life, and in the dataset, multiple instances of such signs are often attached to a single sign post, thus appearing as being being stacked on top of one another.

Such a situation appears to cause confusion among some of the annotators. While most annotate the image correctly, that is by drawing the contour of each individual sign separately, others instead annotate the whole stack of signs on the signpost as a single sign, as shown in Figure 3.3. In cases like this one, it is easy to see that the average bounding box approach fails and produces oversized boxes.

Bounding box intersection. The last technique to be presented is the one chosen to address the aforementioned problems. Intuitively, it consists in merging two or more bounding box annotations that belong to the same object by producing the box that represents their intersection.

The algorithm is detailed in Algorithm 3.1. The astute reader will notice it is inspired by non-maximum suppression (NMS, see Algorithm 2.1). As with NMS, the procedure compares all pairs of bounding boxes, and considers two boxes to represent the same object if their intersection-over-union is greater than a given threshold λ_{dedup} .



Figure 3.3: An example of ground truths provided by two different annotators for respectively the following standard traffic sign types: one N-road number sign (F23a) in blue, one F31 in green, two F29 in blue, and one F33a in white [58]. While annotator yellow correctly annotated each individual sign but the first one, annotator magenta wrongly annotated the two F29 signs as a single sign. Note that the differences in bounding box thickness are solely to aid visual analysis.

There are however two main differences with the original algorithm. First, whereas NMS only ever suppresses one box per iteration, our method suppresses both matching boxes and introduces their intersection box. The second and most important difference resides in a heuristic designed to deal with enclosed boxes. This heuristic ensures that boxes which totally enclose another one are eliminated, so as to completely squash the problem that arises with the average bounding box approach.

3.3.2 Study of the Domain Shift

As stated before, the main goal of this thesis is to improve the performance of an object detection algorithm on the target domain, using knowledge acquired from the source domain through domain adaptation. Recall from Section 1.1 that we do not have a priori knowledge of these two domains. As such, this section aims to answer our first research question, which seeks to identify the domain shift in our dataset (i.e., the criterion on which to split our dataset into a source domain and a target domain).

More formally, the task consists in finding the function $d: \mathcal{X} \rightarrow \{0, 1\}$ able to assign its *domain label* to an image in the subspace $\mathcal{X}$, as belonging to either the source domain ($d = 0$) or the target domain ($d = 1$). Recall that in domain adaptation, the domain labels are always

```

Input: All bounding boxes  $\mathcal{B}$ .
Output: Merged bounding boxes  $\mathcal{B}$ .
for  $b_i \in \mathcal{B}$  do
  for  $b_j \in \mathcal{B}$  do
    if  $\text{IoU}(b_i, b_j) \geq \lambda_{\text{dedup}}$  then
       $b_k \leftarrow \text{INTERSECTION}(b_i, b_j)$ 
       $\mathcal{B} \leftarrow \mathcal{B} \cup \{b_k\}$ 
       $\mathcal{B} \leftarrow \mathcal{B} \setminus \{b_i, b_j\}$ 
    else if  $\text{ENCLOSES}(b_i, b_j)$  then
       $\mathcal{B} \leftarrow \mathcal{B} \setminus \{b_i\}$ 

```

Algorithm 3.1: Inter-annotator disagreement reconciliation based on bounding box intersection. We designed this algorithm taking inspiration from non-maximum suppression (see Algorithm 2.1).

known during learning; this is true even in the unsupervised case, as then, only the *class* labels in the target domain are unknown.

Here, we discuss different metrics that can be measured on an image and we look for the one that is the most relevant for our dataset, that is a metric allowing us to produce a domain split such that the performance of the baseline model is significantly higher on one subset (the source domain) than on the other (the target domain). We simultaneously search for the metric’s threshold value that maximizes this performance gap.

Because the images were taken at different times of the day (including before sunrise), we first investigate image luminance as a potential metric. Sharpness is also considered to account for the motion blur induced by the vehicle’s movement. Finally, after further investigation, we rediscover the fact that the pictures were taken using two different cameras. We find that this explain a significant amount of the domain shift observed and use this as our domain split criterion.

Image Luminance

As stated above, the first step we can take to determine the characteristics of the domain shift is to simply take a look at the data, see what it looks like, and try to find significant patterns. Doing so, one can quickly identify that some images have been taken very early in the morning, before the sunrise. In such images, the sky appears dark; as a result, the overall picture quality suffers, and the traffic signs look darker and blurrier than during the daytime.

Therefore, a first idea to segregate the two domains is to classify clear images as belonging to the source domain and dark images as belonging to the target domain. A way to do this is to

compute an estimation of the luminance for each image in the dataset, and pick a luminance threshold ℓ_{thresh} as the discriminating criterion.

Computing the luminance ℓ of an image is a two-step process. First, we convert the image from a red-green-blue representation to grayscale, leveraging the Pillow library [64]. Suppose a three-channel image pixel $\mathbf{p}_{\text{rgb}} \in \mathbb{Z}^3$ is defined by

$$\mathbf{p}_{\text{rgb}} = \begin{pmatrix} r & g & b \end{pmatrix} \quad (3.3)$$

where $r, g, b \in \{0, \dots, 255\}$, then Pillow calculates its grayscale representation p_{gray} using the standard luma transform from ITU-R Recommendation BT.601 [65]:

$$p_{\text{gray}} = 0.299r + 0.587g + 0.114b. \quad (3.4)$$

We then average all the pixel levels of the image to obtain an estimation of its luminance.

In practice, for very low luminance estimations ($\ell_{\text{thresh}} = 55$), we find that images under that threshold indeed appear quite dark. Images above that threshold mostly appear to have been taken during daytime, but we notice a number of false positives.

There are, however, two more pressing issues at play with keeping this discriminating model where we consider a domain of daytime images and another one of nighttime images.

In our dataset, images that fall under the luminance threshold of 55 only represent about 10% of cases, leading to a strongly imbalanced domain assignment. In machine learning, imbalance often hinders the performance of a classifier [66] and is therefore better avoided.

Moreover, we notice that dark images are more likely to be misannotated than their brighter counterparts: the bounding boxes are of lower quality (i.e., too wide) or simply non-existent, as demonstrated in Figure 3.4. This would bias our future models against the target domain. For these reasons, we decide to explore different criteria.

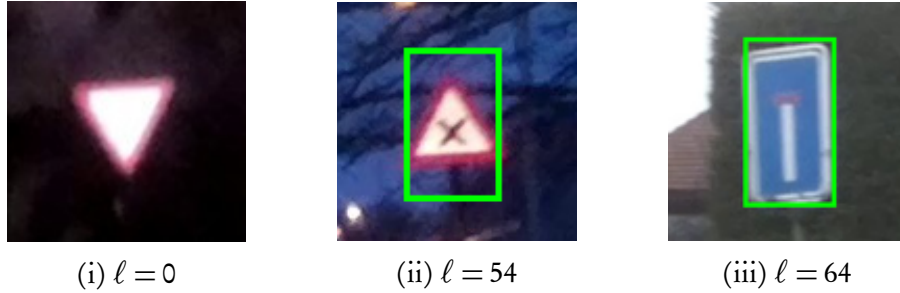


Figure 3.4: Cropped examples showing annotations from different images of the dataset. The sign in (i) was missed by all annotators, while the annotation in (ii) is too wide. (iii) demonstrates good annotating skills in daytime conditions.

Image Sharpness

The idea of studying image sharpness as a domain split criterion comes from observing the results of the previous split done with regard to image luminance. Indeed, when taken in low-light conditions, a picture appears blurrier. This is mostly caused by the camera, as it tries to counterbalance the lack of light and adds noise to the image.

Another main source of blurriness relates to the movement of the vehicle. As pictures are taken as the vehicle is moving, they may be subject to motion blur, which is a consequence of the fact that a camera does not capture a single instant in time. Rather, the image represents the scene over a period of exposure which is determined by the shutter speed [67].

To a lesser extent, weather conditions can also contribute to making the image appear blurry or noisy. Fog is an obvious cause. Rain adds to the list, as water droplets distort the image, either in the air or on the windshield.

In summary, sharpness can be seen, to some degree, as being a proxy for luminance, while also giving information on how detailed an image is in daytime settings.

In the literature, there exists multiple techniques to compute the blurriness of an image [68]. Fortunately, blurriness is an ideal surrogate for sharpness, as the two metrics are the opposite of one another. We can therefore use any blurriness metric to rank images based on their sharpness.

[69] describes a technique that outputs a scalar. This is convenient as it makes ranking images simple. The technique relies on calculating the variance of the Laplacian of an image: as the Laplacian highlights regions which contain rapid intensity changes, its variance gives a measure of how blurry an image is.

More formally, consider an image I of size $m \times n$. The variance of the Laplacian ΔI is given by

$$\sigma_I = \text{Var}(\Delta I) = \sum_i^m \sum_j^n (|L(i, j)| - \bar{L})^2 \quad (3.5)$$

where $L(i, j)$ is the convolution of the image at point (i, j) with convolution kernel

$$\frac{1}{6} \begin{pmatrix} 0 & -1 & 0 \\ -1 & 4 & 1 \\ 0 & -1 & 0 \end{pmatrix} \quad (3.6)$$

typically used to detect the edges in an image, and $\bar{L}$ is the mean of the absolute values of $L(\cdot, \cdot)$. This metric is trivially computed using the OpenCV library [70].

Empirically, image sharpness as a metric suffers from the same shortcomings as the previous method, based on luminance. First, choosing a threshold able to well discriminate between blurry and sharp images ($\sigma = 45$) only yields about a 5% proportion of blurry images for the target domain. This leads to imbalance which is worse than with the previous method. Second, very blurry images tend to be more incorrectly annotated than sharp ones, leading

to bias.

On top of these issues, a new one arises due to differences in the environment surrounding the camera and the fact that Laplacian variance is not robust to images containing large regions of low noise. Consider as an example an image A taken while roaming the countryside versus an image B taken in an urban area. Assume that in both images, regions of interest (including traffic signs) appear sharp. Because the content of image A is mostly low-noise (sky mostly visible, farmland, empty roads), while that of image B has more variance (detailed buildings hiding the sky, busy roads), we generally find that $\sigma_A \ll \sigma_B$, regardless of the actual difference in blurriness among the two images (see Figure 3.5).



(i) $\sigma = 38$



(ii) $\sigma = 1,377$

Figure 3.5: Examples of two images (which are visually equally sharp) showing the nonrobustness of our sharpness metric: (i) was taken in a rural area and is measured to have low sharpness because of the prominent sky and empty road ahead, while the buildings, cars, trees, paved street and shadows in (ii) boost the metric. In reality, both images should be measured to have similar sharpness. Note that in all cases, the bottom third of an image is ignored when calculating its sharpness, so as to avoid bias from the part of the dashboard in the frame.

A mitigation for this last issue is possible in the supervised case. The idea is as follows: rather than to calculate blurriness for a whole image, we calculate it for each traffic sign bounding box region and average the results. In this way, we discard irrelevant background information. However, this idea implies to have a priori knowledge of the traffic signs present in an image, which would prevent to apply potentially interesting unsupervised domain adaptation techniques to this research problem.² Moreover, we have found empirically that it does not give significantly better results. In consequence, we do not pursue this endeavor any further.

In conclusion, image sharpness, at least as measured by Laplacian variance, acts as a poor domain splitting criterion and we thus reject it.

²Recall that in the unsupervised case, ground truths are not known for a certain proportion of images, hence we can not calculate the blurriness of the traffic signs of such images.

Solar Altitude

The discussion of the two previous metrics stems from the idea that it is possible to divide an image set into two different sets based on the illumination of the scene in each image. The point in calculating the luminance of an image is thus directly obvious and, as mentioned, it makes sense to use sharpness as a partial proxy for it. Techniques to estimate such metrics are however very dataset-dependent [68] and, as shown by the results obtained, do not always yield good estimates. In this optic, it is interesting to search for an objective metric for illumination. This helps build confidence in the fact that the technique used is reliable and gives results that will be reproducible in the future, under certain assumptions.

One might consider time as an objective metric for illumination. Indeed, as was already mentioned, the dataset contains images that were taken very early in the morning. In this case it might make sense to reason that images may be grouped solely based on their timestamp, as early-morning images would appear darker. However, this approach assumes that all images were taken approximately at the same time of the year, which is not the case, as shown in Figure 3.6. This is problematic because the length of the day (as delimited by sunrise and sunset) is about eight hours at the December solstice in Belgium, while being twice as great at the June solstice, meaning daylight appears much earlier in the summer time. Time is thus not an objective metric in this context and would need to be corrected to account for seasonal differences.

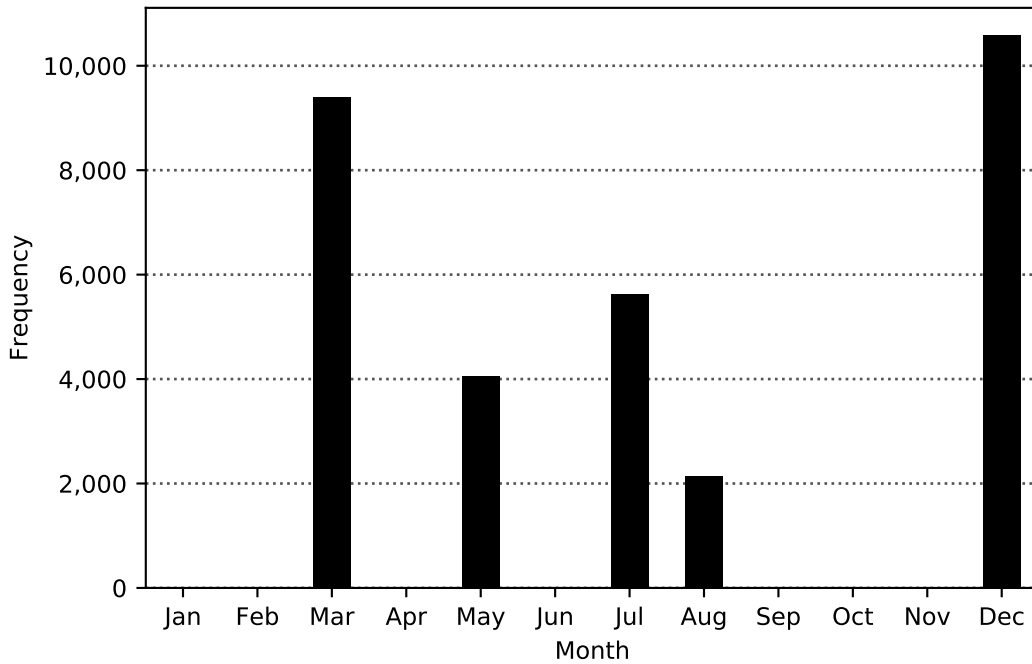


Figure 3.6: A histogram showing the distribution of the number of images across the months of the year.

Remark that in the previous paragraph, we rely on differences in the position of the sun in

the sky to argue that time is not a good metric for illumination. By deduction, one may want to look at solar altitude, which describes the angle formed by the position of the geometric center of the sun relative to the horizon [71]. It is objective because given a location and a timestamp, it is possible to precisely approximate the location of the sun in the sky, and thus its altitude. Moreover, under the assumption that the scene is not majorly obstructed from the sun, be it due to weather conditions or any object, it can be inferred that scene illumination mostly depends on solar altitude. Seasons do affect solar altitude, which is greater on average in the summer than in the winter, but the difference is not significant.

As a preparation step, we compute the solar altitude for all images in the dataset. The details of the algorithm used to compute it are outside the scope of this thesis, and the Python library `pysolar` [72] was leveraged to that effect. It takes a timestamp and a location as argument, and returns an angle in the range $[-90^\circ, 90^\circ]$. Because the images are not individually tagged with location information, Brussels was used as a mid-point between Roeselare and Zonhoven, incurring no significant loss of precision.

As always, once a metric has been defined, the time comes to search for a splitting threshold according to that metric. As solar altitude is an objective metric, one may be interested in deriving a threshold that is also objective.

An obvious first suggestion is sunrise. Although it is defined to occur when the geometric center of the sun is at approximately a -0.83° angle with the horizon [71], we will use 0° for convenience. However, looking at the data does not inspire confidence in the fact that 0° is a good threshold: the illumination of images close to that threshold, regardless of which side of the threshold they are, looks homogeneous.

The above is explained, perhaps unintuitively, by the fact that the sky starts lighting up well before sunrise. In astronomy, this illumination of the lower atmosphere that occurs before the sun is directly visible is called *morning twilight* [71], and starts when the sun reaches an altitude of -18° . It occurs because rays of sunlight scatter in the upper atmosphere and illuminate the lower atmosphere such that the Earth's surface is not completely dark. Naturally, the sky will appear progressively lit as morning twilight progresses and ends with sunrise. A similar phenomenon happens at sunset and is called evening twilight.

Furthermore, astronomers divide morning twilight in three subcategories, each corresponding to different levels of illumination [71]:

Astronomical twilight. This refers to the darkest period of twilight and lasts from the moment the sun is at an altitude of -18° until it reaches -12° . At this stage, the sky is too dark to allow for almost any sort of unguided terrestrial navigation and some stars are still visible to the human eye [71].

Nautical twilight. During this period, from -12° to -6° , [71] notes that “the horizon is generally not visible and it is too dark for marine observations.”

Civil twilight. This begins when the sun reaches an altitude of -6° and ends at sunrise. It corresponds to the most illuminated period of twilight, where solar illumination

is sufficient for the human eye to clearly distinguish terrestrial objects. In the United States, the Federal Aviation Administration (FAA) defines nighttime as ending at the beginning of morning civil twilight [73] and thus allows aircrafts to fly without position or anti-collision lights during this period [74].

Period	Solar altitude	Frequency	Cumulative
Astronomical twilight	-18° to -12°	7.9%	7.9%
Nautical twilight	-12° to -6°	5.1%	13.0%
Civil twilight	-6° to 0°	13.8%	26.8%
Daytime	0° and over	73.2%	100.0%

Table 3.2: The distribution of the number of images belonging to the different twilights, as well as those taken after sunrise. Note that there are no images taken before astronomical twilight.

Hence, the different solar altitude values given above seem to be good values to investigate as potential splitting thresholds.

Let us first consider a threshold of -12° , such that the set of images belonging to the astronomical twilight and earlier is mapped to the target domain, while the remainder is mapped to the source domain. Table 3.2 gives that, in this case, only 7.9% of images are mapped to the target domain. In addition, looking at the data it becomes clear that this threshold loosely acts like a surrogate for the threshold $\ell = 55$ that was discussed in Section 3.3.2 on image luminance, before being rejected for multiple reasons including the strong resulting imbalance.

At this stage, we decide that rather than to merely look at the images to characterize the quality of the split and choose a threshold, a more precise protocol should be defined.

Two Cameras

In Section 4.1, we describe a deep-learning model based on YOLOv3 to act as a baseline for our research. As it is a baseline, that model does not perform domain adaptation, although it was pretrained and then fine-tuned. The resulting model thus performs traffic sign detection with no specific treatment for any kind of image data.

As a result, we can define a protocol that leverages this baseline to help us pick a solar altitude threshold that produces a good split. The protocol stems from a simple idea: given a threshold t , we can use the baseline and evaluate its performance on the set $\mathcal{I}_{\geq t}$ of images above the threshold, compared to the set $\mathcal{I}_{< t}$ of images below the threshold. Thus, the performance gap $\Delta\mathcal{P} = \mathcal{P}(\mathcal{I}_{\geq t}) - \mathcal{P}(\mathcal{I}_{< t})$, where $\mathcal{P}(\cdot)$ is a performance metric, measures how hard it is for the baseline to predict both sides of the threshold relative to one another. The complete protocol consists in calculating the performance gap for several thresholds of solar altitude using the usual AP metric. As a larger performance gap means that the two resulting

sets are well-segregated, we aim to pick t such that the gap is maximized. Simultaneously, we are careful to choose t such that $|\mathcal{I}_{< t}|$ is large enough to avoid the different issues highlighted in the previous subsections.

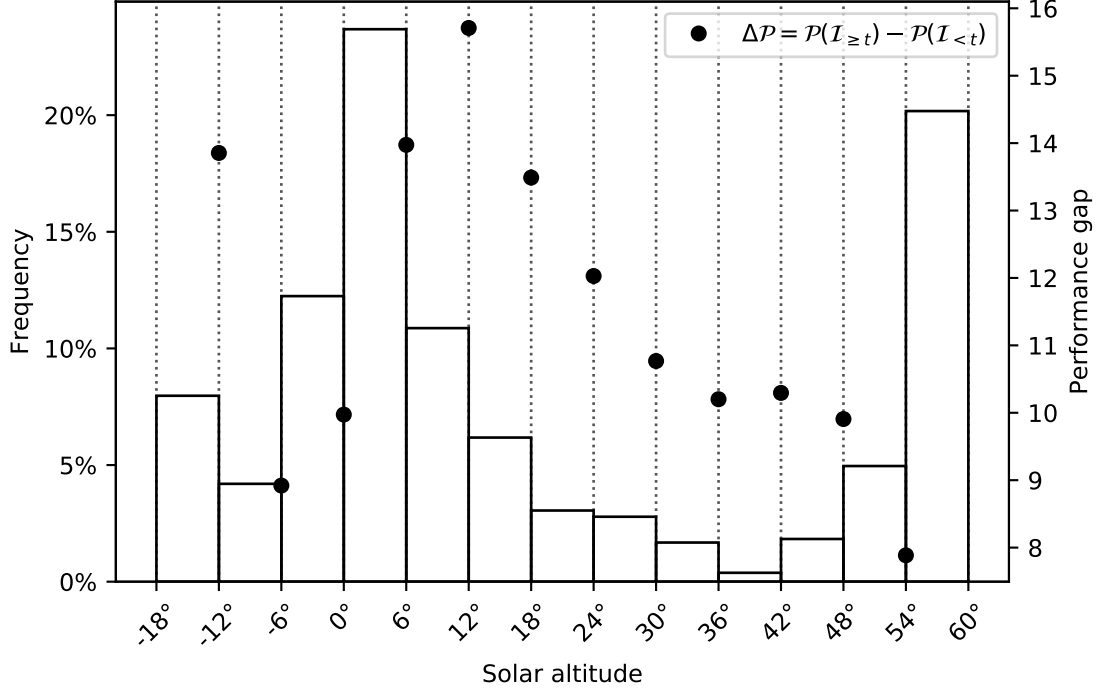


Figure 3.7: A plot showing the performance gap $\Delta\mathcal{P}$ at different solar altitude thresholds on a sample of the dataset, using the AP metric. It is overlaid over an histogram showing the distribution of solar altitude.

Figure 3.7 shows the result of executing this protocol on a random sample of 2,500 images from the dataset. Let us study the shape of the performance gap curve.

When the threshold is at its lowest at -12° , the performance gap is quite high and shows an AP difference of approximately 14 percentage points. This is expected, as images where solar altitude is under -12° are very dark and thus challenging for the model. More generally, we expect $\mathcal{P}(\mathcal{I}_{< t}) \leq \mathcal{P}(\mathcal{I}) \leq \mathcal{P}(\mathcal{I}_{\geq t})$ to hold true for any given t .

As we increase t , the performance gap is affected in two ways. First, we remove images that are progressively harder to predict from $\mathcal{I}_{\geq t}$, thus increasing $\mathcal{P}(\mathcal{I}_{\geq t})$. Simultaneously, we add progressively easier images to $\mathcal{I}_{< t}$ such that $\mathcal{P}(\mathcal{I}_{< t})$ also increases. However, because $\mathcal{I}_{< t}$ is initially empty, while $\mathcal{I}_{\geq t}$ represents the entire dataset, the performance on the former set is more sensitive to new images. As shown in Figure 3.8, the consequence is that $\mathcal{P}(\mathcal{I}_{< t})$ increases faster than $\mathcal{P}(\mathcal{I}_{\geq t})$ making the curve of the performance gap $\Delta\mathcal{P} = \mathcal{P}(\mathcal{I}_{\geq t}) - \mathcal{P}(\mathcal{I}_{< t})$ be downward-sloping. However, this behavior does not hold true indefinitely.

Unexpectedly, from $t = -6$ to $t = 12$, the opposite pattern emerges briefly, as visible in Figure 3.7: the performance gap is subject to a sharp rise from 8.9 percentage points (p.p.)

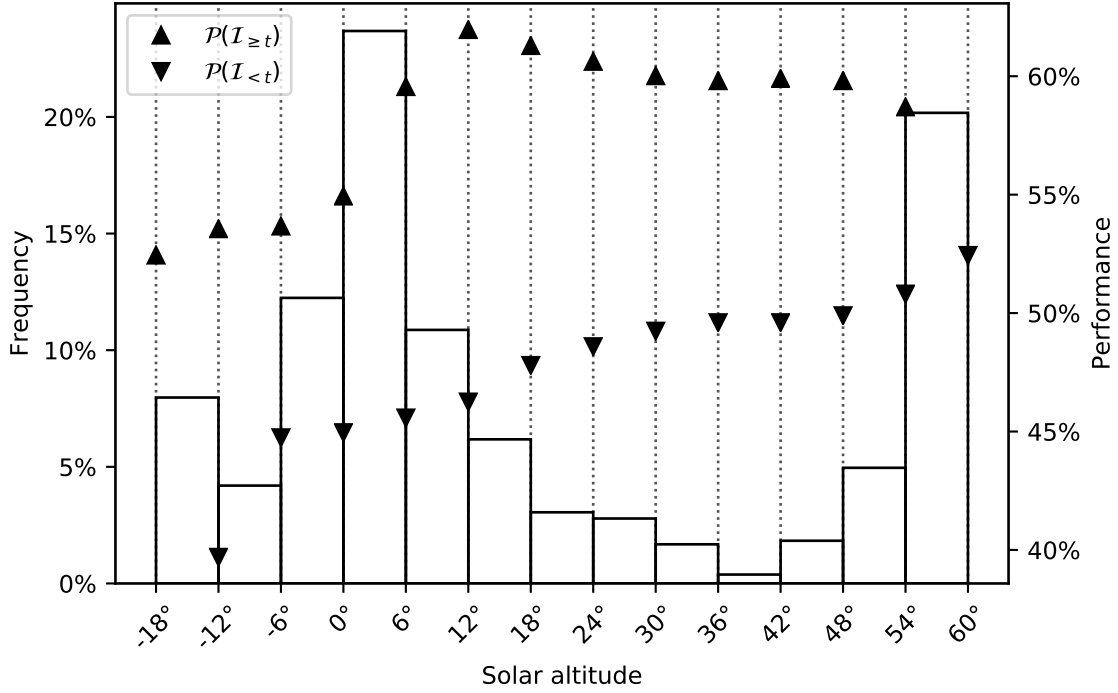


Figure 3.8: A plot splitting the performance gap shown in Figure 3.7 into its two components, namely $P(I_{\ge t})$ and $P(I_{< t})$.

to 15.7 p.p., before starting to decrease again. Figure 3.8 shows that this is due to a faster increase of $P(I_{\ge t})$, compounded with a slower increase of $P(I_{< t})$.

We hypothesize that some distribution shift takes place in the data as solar altitude increases from -6° to 12° . This seems to be true when we look at the images in this range: images at the beginning of the range appear warmer and more crisp than images at the end of the range, as shown in Figure 3.9.

Let us test the hypothesis that a distribution shift happens in the range $[-6^\circ, 12^\circ]$. Recall from Figure 3.6 on page 34 that the data collection phase spanned three seasons: winter, spring, and summer. As the images at the end of the range appear warmer and more crisp, it may be that a season change is taking place. To verify this, we plot, for each season, the average AP of each *individual* bin as a function of solar altitude in Figure 3.10.

We indeed observe a stark shift in the distribution of seasons, as images under -6° all belong to the winter, and images from summer and spring are progressively added as solar altitude increases, with 12° defining the cutoff point for winter images (i.e., beyond which there are no more winter images). However, the data also shows that solar altitude alone is not a good domain discriminant, as the AP for the summer and spring seasons is not significantly lower under, say, 12° .

As a consequence, one may be tempted to split the dataset according to its seasons, considering summer and spring images to be part of the source domain, and winter images to make



(i) solar altitude = -3.9°



(ii) solar altitude = 9.6°

Figure 3.9: A representative sample of what images look like when solar altitude is between -6° and 12° .

up the target domain. However, the fact that the baseline model performs worse on winter images on average, even under similar weather conditions, still raises questions.

After investigating this further with my coworkers at EURA NOVA, it came to light that the performance gap was most likely due to camera differences. Indeed, we realized that the camera used for data collection during the winter was different from the one used in the summer and the spring. Because the winter camera produces images of lower quality than the other camera, the performance of the baseline model suffers on these images.

We evaluated the performance gap on the baseline model between the high-quality camera (summer and spring) and the lower-quality (winter) camera at 12.6 points. Indeed, the baseline gives an AP of 57.2% on high-quality images and 44.6% on lower-quality images. Furthermore, such a split avoids the strong imbalance issue encountered with the previous metrics, with the high-quality set containing 21,181 images (66.7% of the complete dataset) and the low-quality set having 10,580 images (33.3%).

On this basis, the source-target split is done such that the source domain $\mathcal{D}_S$ corresponds to the high-quality camera images and the target domain $\mathcal{D}_T$ corresponds to the lower-quality camera images.

3.3.3 Train-Validation-Test Set Split

As mentioned in Section 2.2, cross-validation in deep learning requires us to split the observation dataset into a training set, a validation set, and a test set. For each of the subdomains, we split the data such that the first 80% maps to the training set, the next 10% maps to the validation set, and the final 10% maps to the test set. In our case, we must apply special care to the splitting procedure due to the nature of the problem.

In total, six subsets are created:

- The source domain training set $\mathcal{D}_S^{(\text{train})}$;
- The source domain validation set $\mathcal{D}_S^{(\text{valid})}$;

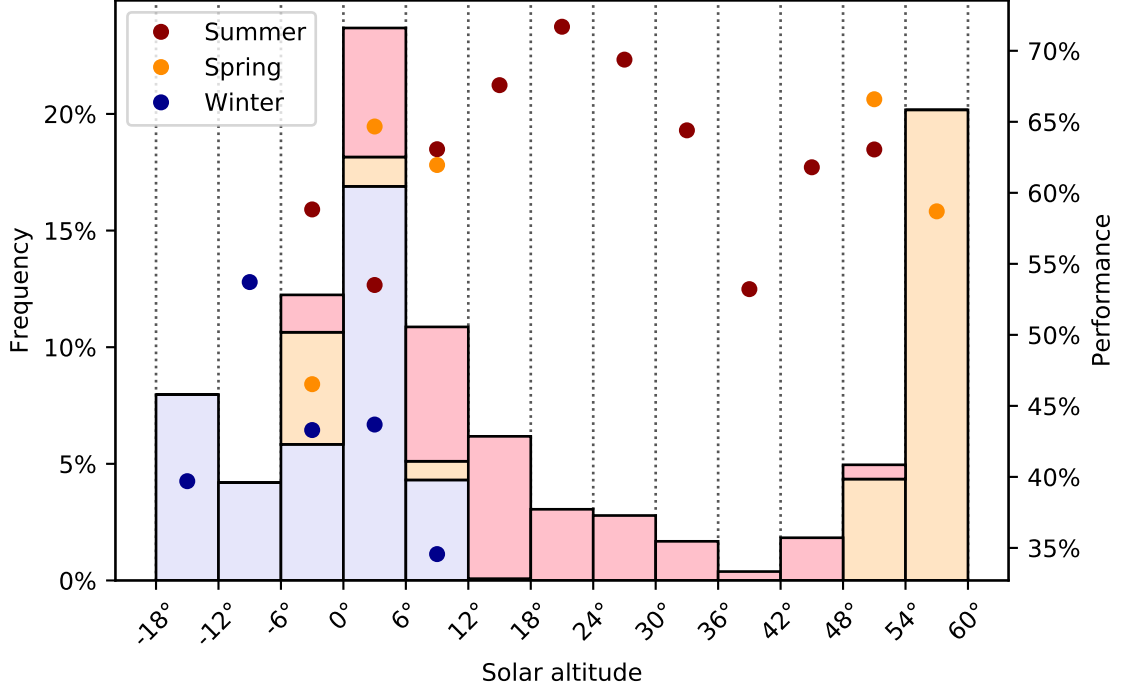


Figure 3.10: A color-coded plot in which each point represents the average AP for the *individual* bin with which it is vertically aligned that matches its color.

- The source domain test set $D_S^{(\text{test})}$;
- The target domain training set $D_T^{(\text{train})}$;
- The target domain validation set $D_T^{(\text{valid})}$;
- The target domain test set $D_T^{(\text{test})}$.

For these six subsets, we must employ stratified sampling [75] such that each subset is distributed identically along two criteria in the dataset. Let us go over these criteria.

Delivery round. As explained in Section 3.1, the images were collected during delivery rounds. As such, it is possible that the exact same traffic sign is recorded more than once in the dataset. In order to not bias the model evaluation routine toward such images, it is important to ensure that the same traffic sign does not appear simultaneously in a training set and in either a validation set or a test set (or both). Although it is not possible to strictly enforce this property in general, we aim toward that goal by using the following heuristic: when splitting the dataset, group together images that were taken during the same delivery round (and thus on the same day).

Presence of traffic signs. The second dimension pertains to the ratio of the number of images which contain traffic signs (according to the annotators) against the number of images

which do not. In the dataset, 45% of images have been annotated as containing at least one traffic sign. As such, it is important to maintain this prior probability constant across all splits so that the ratios preserve the same balance.

However, stratified sampling in both dimensions simultaneously is not possible with our dataset, as the two distributions are not strictly independent. As a result, we devise a strategy to approximate the balance that would have been guaranteed by two-dimensional stratification. First, we choose the delivery round dimension as the dimension along which it is most crucial that stratified sampling be performed. Then, we do multiple rounds of stratified sampling along that dimension, each round producing a different split of six subsets. Finally, we select the round for which the mean square error (MSE) of the proportion of images without traffic signs is minimized compared to the target of 45%, giving in our case an MSE of 5.7. Table 3.3 shows the distribution of the final split.

	Source	Target	Σ
Training	16,944	8,464	25,408
Validation	2,118	1,058	3,176
Test	2,119	1,058	3,177
Σ	21,181	10,580	31,761

Table 3.3: The distribution of the final split between the source and target domains, as well as the training, validation, and test sets.

With our data pipeline properly set up, we are now ready to dive into our main research question: in the next chapter, we establish a number of baseline models that will allow us to evaluate how exactly deep domain adaptation performs.

4 Baseline Models

Before we dive into researching domain adaptation techniques to solve our problem, it is important to define a number of baselines in order to validate any future result, so that we can verify whether the techniques that we experiment with actually improve on a simpler and more naive model (see the second research question in Section 1.1). In this chapter, we consider as a first baseline a YOLOv3 [9] model, trained in a standard, non-adaptive way. We then define a second adaptive baseline based on the subspace alignment approach described in Section 2.5.4.

4.1 Non-Adaptive Baseline

As the basis of our future models, we consider YOLOv3 [9], as described in Section 2.4.1, trained in a standard supervised fashion. The reasons behind this choice are manifold.

First and foremost, YOLOv3 has been the model used at EURA NOVA to solve the traffic sign detection problem so far. As such, if we are going to implement any domain adaptation technique, we must know how it fares against the technique currently in usage. Also, a lot of infrastructure already exists to support YOLOv3 and it is therefore easy to modify to act as baseline model.

Second, as we demonstrate in the next chapters, domain adaptation models often make use of a feature extractor component, as well as a classifier, and these two components are found in the YOLOv3 architecture. As a consequence, YOLO plugs well into those domain adaptation architectures.

The third reason may also be included in the first but is worth standing on its own: speed. Although better-performing object detection models like Faster R-CNN [43] exist, they are often an order of magnitude slower than YOLOv3. However, given the quantity of data that could be generated would this project be expanded to more cities and delivery vehicles, it is crucial to be able to process that data in a timely manner.

4.1.1 Training

Many decisions have to be taken when training deep learning algorithms, and YOLOv3 is no exception to the rule. In this section, we describe the choices made in terms of training

strategy, preprocessing, and hyperparameters. Training was done on an NVIDIA V100 GPU with 32 GB of memory [76].

We train YOLOv3 starting from pretrained weights made available by the authors of the paper. Let us note that this is in itself already a form of transfer learning, which has produced good results for multiple applications and has shown its effectiveness in computer vision tasks [77]. From that starting point, training can be decomposed into two stages.

Stage 1: Freezing the feature extractor. We first freeze all but the last three layers of the network (see Figure 2.1 on page 13) such that their weights are not updated during backpropagation. The last three layers are the heads of the network: they take the features learned by the model and output classifications (i.e., bounding boxes), and as such we refer to them as being the “classifier” part of the network. As a consequence, only the classifier’s weights are fine-tuned in this stage.

The motivation behind freezing the feature extractor in the first stage stems from the fact that it performs poorly initially. Indeed, because the model has not been trained on our dataset, the forward pass induces large errors at the end of the network, which are then backpropagated all the way to the feature extractor (i.e., the layers not included in the classifier). As such, the performance of the feature extractor, and thus the classifier, deteriorates quickly even though they were pretrained.

By first freezing the feature extractor, we let the classifier first learn good bounding boxes and prevent the feature extractor from decreasing the performance of the model. This also has the benefit of speeding up convergence.

Stage 2: Fine-tuning the feature extractor. After training the classifier for a given number of epochs, its performance generally starts to stabilize and plateau. This is a sign that it cannot learn better bounding boxes without more information. At that point, we unfreeze the feature extractor. Because the loss of the classifier is now maximized, the feature extractor is not anymore subject to the backpropagation of large errors. As such, it can learn better features in a more stable fashion. At the point where we unfreeze the feature extractor, the downward acceleration of the loss increases briefly before the loss resumes to decrease more gradually, as shown in Figure 4.1.

Data Augmentation

Before we delve into the hyperparameters chosen for each stage, let us first discuss a technique employed during both stages: data augmentation. It is a common technique in deep learning applied to computer vision tasks [78] which applies perturbations to the learning examples. Doing so increases the diversity of the data and allows a model to generalize better by reducing overfitting.

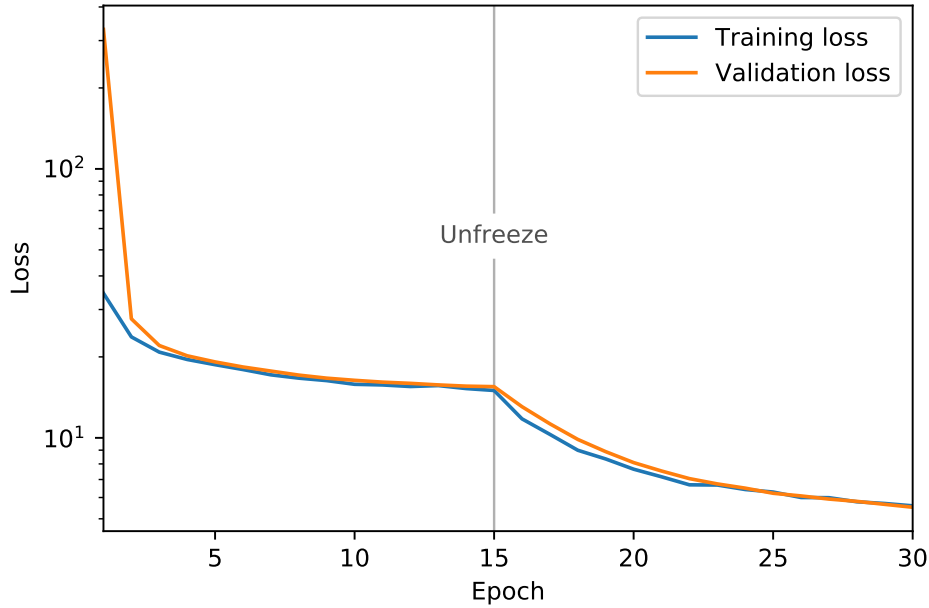


Figure 4.1: A plot showing the impact of freezing and unfreezing the feature extractor on model performance. For the first 15 epochs, the feature extractor is frozen and both learning curves converge to a plateau. After that point, the feature extractor is unfrozen and the losses start to decrease again.

Note that even though data augmentation takes place in both stages of the learning algorithm, it is only applied to the training set in our case. Indeed, as we try to increase the diversity of the data to better generalize, it makes sense to evaluate the model’s performance solely on non-augmented data (otherwise the evaluation protocol might be biased).

We implement various image augmentation transforms using the Python library `imgaug` [79]. To apply these transforms to each image, we split the transforms into four layers, each containing a set of transforms. Each layer activates randomly with an 80% probability (independently from the others) and then triggers one of its transforms, again at random.

The first layer contains the following transforms:

- *Noise*: Adds noise sampled from a Gaussian distribution in a pixel-wise fashion.
- *Dropout*: Sets the intensity of a fraction of pixels to zero.
- *Coarse Dropout*: Sets the intensity of some rectangular regions to zero.

As explained, the first layer thus activates with a 80% probability, and if so triggers one of *Noise*, *Dropout*, or *Coarse Dropout*.

The second layer pertains to geometric transforms and consists of:

- *Perspective*: Applies a perspective transformation to the image.
- *Rotate*: Applies a rotation to the image.
- *Flip*: Flips the image horizontally.

The third layer consists of a single transform that we call *Crop/Pad*. It either crops the image by extracting a subimage, or enlarges it before padding it with zeros.

The final layer applies more blur- and color- related transforms to the images:

- *Blur*: Adds blur using a Gaussian kernel.
- *Contrast*: Increases or decreases contrast by some value.
- *Brightness*: Increases or decreases brightness by some value.
- *Hue/Saturation*: Increases or decreases hue and saturation by some value.
- *Grayscale*: Converts the image to grayscale.

Hyperparameters

In this section, we first discuss hyperparameters and strategies common to most deep learning algorithms, before diving into YOLOv3-specific aspects.

You will note that, while we do make an effort to produce a good-performing baseline through techniques such as data augmentation, hyperparameter tuning in general is time-consuming, and as such we trade model performance for faster training in this section. This is not problematic as domain adaptation techniques generally incorporate a reference model (YOLOv3 in our case), hence relative performance is measured, not absolute performance.

As described earlier, training is divided into two stages, the first being focused on the classifier while the second applies to the whole network. We train both stages for 15 epochs, with learning rates of 10^{-3} for the first stage and 10^{-4} for the second one. In the second stage, though, the learning rate is subject to change as we leverage learning rate decay. This technique is quite popular [80, 81, 82] and consists in decreasing the learning rate when the performance of the model reaches a plateau. Here, we divide the learning rate by 10 if the validation loss has not improved for 5 consecutive epochs.

We use the Adam algorithm [83] for *parameter* optimization. It is an improvement of stochastic gradient descent based on AdaGrad [84] and RMSProp. It works by computing individual adaptive learning rates for different parameters and outperforms other optimization algorithms in most tasks [83]. We use the default settings $\beta_1 = 0.9$ and $\beta_2 = 0.999$ for the decay rates of the moment estimates, as recommended by the paper’s authors.

When generating batches, we use a batch size of 32 for the first stage. Due to memory usage, this corresponds approximately to the maximum batch size we can use. In the second stage, because we unfreeze the feature extractor, the algorithm has a much greater amount of parameters to optimize. As such, we decrease the batch size to 16.

The last general deep learning strategy to mention is of course early stopping [85], in which we stop training if, despite our previous efforts, model performance has not improved for 10 consecutive epochs. This is yet another attempt at preventing overfitting.

Let us now discuss the YOLOv3-specific hyperparameters present in our baseline model.

The first such hyperparameter is `ignore_threshold`. For each bounding box it predicts, YOLOv3 computes an objectness score that is equal to the highest IoU among ground-truth objects. If a bounding-box prediction does not have the highest IoU, but does overlap with a ground-truth object by more than `ignore_threshold`, the prediction is ignored such that it does not incur any confidence loss. We set `ignore_threshold = 0.5`, following the paper.

The second YOLOv3 hyperparameter, `score_threshold` controls the quality of the bounding boxes output by the model. In particular, it specifies the lower bound of the confidence score of these bounding boxes, such that bounding boxes which fall under the threshold are ignored in the final stage of the algorithm. While a threshold of 0.6 is typical [43], we fix `score_threshold` to 0.3. This is motivated by the fact that, as mentioned in Section 3.2, ground-truth bounding box annotations are often not tight, leading to artificially deflated confidence scores. This is illustrated in Figure 4.2.



Figure 4.2: Examples of ground-truth annotations where the bounding boxes drawn are much larger than the traffic signs represented.

Finally, as YOLOv3 (and object detection algorithms in general) outputs several bounding boxes that overlap each other significantly, it is important to reduce the number of overlapping boxes when they represent a single object through non-maximum suppression (NMS, see Algorithm 2.1). We tune NMS so that the threshold over which it considers that two bounding boxes represent the same object is defined as $\lambda_{\text{nms}} = 0.3$.

4.1.2 Results

As mentioned previously, we train a YOLOv3 model which does not perform domain adaptation in order to determine if our problem requires domain adaptation, or if it can be solved solely using a YOLOv3 architecture. As such, the following question arises: what protocol to implement to determine how a domain-adaptive object detection model compares with a non-adaptive object detection model, in terms of performance?

To aid our argument, we first establish some definitions. Let $\mathcal{P}_S(\mathcal{D}_T)$ be the performance on the target data of a model trained solely with *source* data (source-only model). Similarly, let $\mathcal{P}_T(\mathcal{D}_T)$ be the performance on the target data of a model trained solely with *target* data (target-only model). Finally, we define $\mathcal{P}_{\text{DA}}(\mathcal{D}_T)$ to be the performance on the target data of a model trained using domain adaptation, that is using the features from both domains (domain-adaptive model).

In the domain adaptation literature, a widely-used protocol [3, 86, 87, 88, 89] consists in characterizing the performance of a domain-adaptive model in terms of its ability to bridge the gap between $\mathcal{P}_S(\mathcal{D}_T)$ and $\mathcal{P}_T(\mathcal{D}_T)$. Indeed, $\mathcal{P}_S(\mathcal{D}_T)$ is interpreted as a lower bound for $\mathcal{P}_{DA}(\mathcal{D}_T)$, because a domain-adaptive model which performs worse than a source-only model is worthless, while any improvement over a source-only model is encouraging. Likewise, the protocol considers $\mathcal{P}_T(\mathcal{D}_T)$ as a soft upper bound, implying that a domain-adaptive model may perform better than a source-only model, even though a specialized target-only model may still perform better than a DA model.

With this protocol in mind, we use YOLOv3 to train a source-only model and a target-only model on the two corresponding training sets. Training takes approximately 24 hours in both cases.

As expected, we find that the target-only model performs much better than the source-only model on target data, as we have that $\mathcal{P}_T(\mathcal{D}_T) = 0.4437$ and $\mathcal{P}_S(\mathcal{D}_T) = 0.2867$, using the average precision (AP) metric. Therefore, for our research problem, we will say of a domain-adaptive algorithm that it outperforms this YOLOv3 model if $\mathcal{P}_{DA}(\mathcal{D}_T) > 0.2867$. Additionally, we assume that $\mathcal{P}_{DA}(\mathcal{D}_T) < 0.4437$ may hold.

While these numbers look unimpressive in absolute terms, it is important to keep in mind that object detection is a hard problem, along with the fact that YOLOv3 is not a top-performing model (because of its focus on speed). Furthermore, the quality of the annotations in our dataset is highly variable, as noted previously. For reference, [90] achieved 33.2% AP on the Belgium Traffic Sign Dataset (BTSD) using YOLOv2 [41]. Similarly, [91] achieved 28.3% AP on the BTSD using YOLOv1 [40].

Another interesting result pertains to the quality of the bounding boxes predicted by the model compared to that of the ground truths provided by the annotators. As shown in Figure 4.3, the predicted boxes tend to be tighter than the ground truths, comforting us in our choice of `score_threshold = 0.3`. Furthermore, the model can have better recall than the annotators and spot traffic signs that were missed by them.

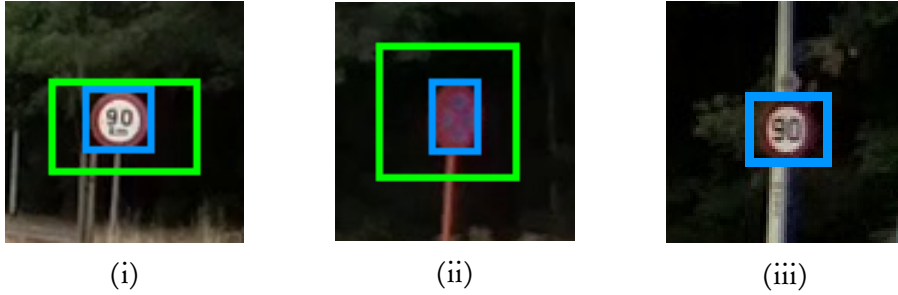


Figure 4.3: Examples of bounding boxes predicted by the model (blue) that are better than than the ground-truth annotations (green). In all cases, the predicted boxes have a tight fit around the actual traffic sign, and (iii) shows that the model can even pick up signs missed by the annotators.

Note that in this thesis we only evaluate the performance of the different models considered

on the target domain. To understand how different models (with regard to the training set used) perform on different domains, refer to Appendix A.

4.2 Adaptive Baseline

As an adaptive baseline, we purposefully choose a technique that is simple to implement and does not require the use of a neural architecture. The reason is simple: recall from the introduction to this work that we want to determine (i) whether an easily implementable domain adaptation (DA) technique can solve the adaptation challenge compared to a non-adaptive method and (ii) whether this DA technique can compete with a more elaborate deep DA scheme. In the following, we evaluate the use of the subspace alignment technique described in Chapter 2 on our dataset.

To evaluate the performance of this SA technique, we plug it into our non-adaptive baseline based on YOLOv3 introduced in the previous section. Unless stated otherwise, we retain the same training procedure, including the hyperparameters, to train this new baseline.

More precisely, we insert Algorithm 2.2 after YOLOv3’s feature extractor. As is standard, we then pass the projected samples to YOLOv3’s prediction branches. Special care is given to the tuning of the number of principal components d . In our case, Darknet outputs feature maps of size 1,071,616, we select d via a grid search through the range $\{2^7, \dots, 2^{20}\}$.

Ultimately, we find the optimal $d^* = 2^{17} = 131,072$. However, it obtains an AP of 23.46% on the test set, which is worse than the performance of the source-only model from Section 4.1, which obtains 28.67% AP. As we will see in the next chapter, for classification tasks, SA does not always perform better than a source-only model (see Table 5.1), and this could be true a fortiori for the harder problem of object detection. As such, we conclude that a simple adaptive algorithm such as the subspace alignment technique described here is not able to perform better than a non-adaptive model trained only on the source domain for our problem.

In the next chapter, we present a more powerful, deep domain adaptation algorithm based on domain-adversarial networks.

5 DANN: Domain-Adversarial Neural Networks

In this chapter, we present a technique introduced in [92] and [86] which focuses on combining domain adaptation and deep feature learning within one training process. This chapter serves as a summary of the aforementioned papers and, unless stated otherwise, all theoretical aspects mentioned here, including concepts, ideas, definitions, formulas and figures, are taken directly from these two papers.

The presented technique embeds domain adaptation into the process of deep feature learning in an *unsupervised* way, such that the final learning decisions are made based on features that are (i) discriminative for the main learning task (e.g. classification, regression, detection) and (ii) invariant to the domain shift. This ensures that the features learned have distributions in the source and target domain that are the same, or very similar. In this way, the resulting model is applicable to the target domain with minimal bias toward the source domain.

To learn features that are discriminative and domain-invariant, the presented technique simultaneously optimizes the following components of the network, shown in Figure 5.1: (i) the *feature extractor* which learns a feature mapping from the input, (ii) the *label predictor* which predicts class labels, and (iii) the *domain classifier* which discriminates between the source domain and the target domain. While the parameters of the label predictor and the domain classifier minimize their losses, the parameters of the feature extractor are optimized so as to *minimize* the label predictor loss and *maximize* the domain classifier loss, the latter encouraging the learning of domain-invariant features.

The behavior described by these three training processes can be achieved by augmenting any existing feedforward neural network architecture (i.e., with no cycle [93]) with a domain classifier component made of standard layers and loss functions, as well as a so-called *gradient reversal layer* (GRL), which acts as the identity function during forward propagation, and reverses the gradient by multiplying it by a negative scalar during backpropagation (see Figure 5.1). In practice, the GRL is trivially implemented and the resulting model can still be trained using gradient descent.

Because of the adversarial nature of the training procedure, where the domain classifier is simultaneously seeking to discriminate the features with respect to the domain, while also being confused by the feature extractor, the resulting model is referred to as a domain-adversarial neural network (DANN). In the rest of this thesis, we also simply refer to the overall technique as “DANN.”

As presented, DANN is trained on a large amount of labeled data from the source domain and a large amount of unlabeled data from the target domain. As no labeled target domain data is necessary, it is an *unsupervised* domain adaptation technique. However, as will become apparent in the next sections, the technique can be generalized to the *semi-supervised* and *fully-supervised* cases straightforwardly, yet we focus on the harder and more interesting unsupervised case. Generally, this can be seen as a way of future-proofing a learning algorithm, as further target domain data collected need not be annotated before being incorporated into the machine learning pipeline, hence making data collection less expensive [86].

5.1 Model

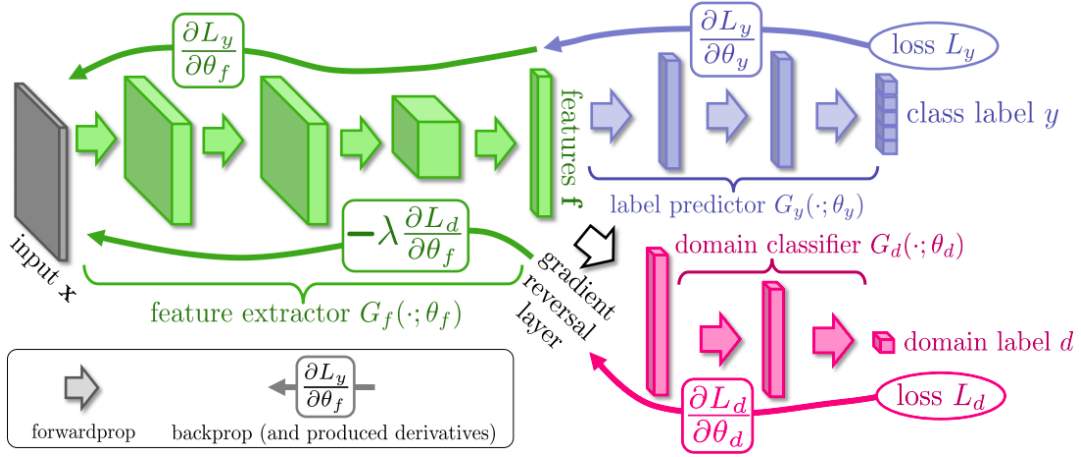


Figure 5.1: The architecture of DANN [92, 86].

We assume that the model works with input examples $\mathbf{x} \in \mathcal{X}$, where $\mathcal{X}$ is an input space, and output labels $y \in \mathcal{Y}$, where $\mathcal{Y}$ is a label space. In the following, we assume $\mathcal{Y} = \{1, \dots, C\}$ (where C is the number of output labels). However, DANN is generic and trivially handles any output label space used by a pre-existing feedforward neural network model. We also consider two domains: the source domain $\mathcal{D}_S$ and the target domain $\mathcal{D}_T$, which are assumed similar but different (i.e., there is a domain shift). We define the total number of examples to be $N = N_S + N_T$, with N_S and N_T being respectively the number of examples in the source and target domain.

The goal is to be able to predict the labels y given the input $\mathbf{x}$ for the *target domain*. At training time, we have access to a large set of training examples

$$\{\mathbf{x}_1, \dots, \mathbf{x}_N\} = \{\mathbf{x}_1, \dots, \mathbf{x}_{N_S}, \mathbf{x}_{N_S+1}, \dots, \mathbf{x}_{N_S+N_T}\} \quad (5.1)$$

from both the source and the target domains according to the marginal distributions $P_S(\mathbf{x})$ and $P_T(\mathbf{x})$ respectively. We denote with d_i the binary variable for the i th example, which indicates whether $\mathbf{x}_i$ comes from the source domain ($\mathbf{x}_i \sim \mathcal{D}_S(\mathbf{x})$ if $d_i = 0$) or from the target domain ($\mathbf{x}_i \sim \mathcal{D}_T(\mathbf{x})$ if $d_i = 1$). For the examples from the source domain, the corresponding

labels $y_i \in \mathcal{Y}$ are known at training time. For the examples from the target domain, we do not know the labels at training time and we want to predict such labels at test time.

We now define a deep feedforward architecture that, for each input $\mathbf{x}$, predicts its label $y \in \mathcal{Y}$ and its domain $d \in \{0, 1\}$. We decompose such mappings into three parts (see Figure 5.1):

- (i) We assume that the input $\mathbf{x}$ is first mapped by a mapping G_f (the feature extractor) to a D -dimensional feature vector $\mathbf{f} \in \mathbb{R}^D$. The feature mapping may include several layers and we denote the parameter vector of all layers in this mapping as θ_f (i.e., $\mathbf{f} = G_f(\mathbf{x}; \theta_f)$).
- (ii) The feature vector $\mathbf{f}$ is then mapped by a mapping G_y (the label predictor) to the label y , and we denote the parameters of this mapping with θ_y .
- (iii) Finally, the same feature vector $\mathbf{f}$ is mapped to the domain label d by a mapping G_d (the domain classifier) with the parameters θ_d .

5.2 Training

During training, we aim to minimize the label prediction loss on the annotated part of the training set (namely $D_s^{(\text{train})}$). The parameters of both the feature extractor and the label predictor are thus optimized to minimize the empirical loss for the source domain examples. This ensures that the features $\mathbf{f}$ are discriminative and that the combination of the feature extractor and the label predictor performs well on the source domain.

Simultaneously, we want to make the features $\mathbf{f}$ domain-invariant. In other words, we want to make the domains $\mathcal{D}_s(\mathbf{f}) = \{G_f(\mathbf{x}; \theta_f) \mid \mathbf{x} \sim \mathcal{D}_s(\mathbf{x})\}$ and $\mathcal{D}_T(\mathbf{f}) = \{G_f(\mathbf{x}; \theta_f) \mid \mathbf{x} \sim \mathcal{D}_T(\mathbf{x})\}$ to be similar. Under the covariate shift assumption (see Definition 2.3), making the features $\mathbf{f}$ domain-invariant makes the label prediction accuracy on the target domain to be the same as on the source domain [48]. Measuring the dissimilarity of the domains $\mathcal{D}_s(\mathbf{f})$ and $\mathcal{D}_T(\mathbf{f})$ is however non-trivial, given that $\mathbf{f}$ is high-dimensional and that the domains are themselves constantly changing as training progresses. One way to estimate the dissimilarity is to look at the loss of the domain classifier G_d , provided that the parameters θ_d of the domain classifier have been trained to discriminate between the two feature domains in an optimal way.

This observation leads to the following idea. At training time, in order to obtain domain-invariant features, we seek the parameters θ_f of the feature mapping that *maximize* the loss of the domain classifier (i.e., by making the two feature domains as similar as possible), while simultaneously seeking the parameters θ_d of the domain classifier that *minimize* the loss of the domain classifier. In addition, we seek to *minimize* the loss of the label predictor.

More formally, we consider the loss function

$$\mathcal{L}(\theta_f, \theta_y, \theta_d) = \sum_{i=1}^{N_s} \mathcal{L}_y^i(\theta_f, \theta_y) - \lambda \sum_{i=1}^N \mathcal{L}_d^i(\theta_f, \theta_d) \quad (5.2)$$

where the label prediction loss (e.g. YOLOv3) for the i th training example is defined as

$$\mathcal{L}_y^i(\theta_f, \theta_y) = \mathcal{L}_y(G_y(G_f(\mathbf{x}_i; \theta_f); \theta_y), \mathbf{y}_i) \quad (5.3)$$

and the domain classification loss (e.g. logistic) for the i th training example is defined as

$$\mathcal{L}_d^i(\theta_f, \theta_d) = \mathcal{L}_d(G_d(G_f(\mathbf{x}_i; \theta_f); \theta_d), d_i). \quad (5.4)$$

Based on this idea, we seek the parameters $\hat{\theta}_f$, $\hat{\theta}_y$, and $\hat{\theta}_d$ that deliver a stationary point of Equation (5.2):

$$(\hat{\theta}_f, \hat{\theta}_y) = \underset{\theta_f, \theta_y}{\operatorname{argmax}} \mathcal{L}(\theta_f, \theta_y, \hat{\theta}_d), \quad (5.5)$$

$$\hat{\theta}_d = \underset{\theta_d}{\operatorname{argmax}} \mathcal{L}(\hat{\theta}_f, \hat{\theta}_y, \theta_d). \quad (5.6)$$

At the stationary point:

- (i) The parameters θ_d of the domain classifier minimize the domain classification loss, since it enters into Equation (5.2) with the minus sign.
- (ii) The parameters θ_y of the label predictor minimize the label prediction loss.
- (iii) The feature mapping parameters θ_f *minimize* the label prediction loss (i.e., the features are discriminative), while *maximizing* the domain classification loss (i.e., the features are domain-invariant)

Furthermore, the parameter λ controls the trade-off between the two objectives that shape the features during learning.

Below, we demonstrate that gradient descent solvers can be adapted for the search of the stationary point (5.5)–(5.6).

5.3 Gradient Reversal Layer

A stationary point (5.5)–(5.6) can be found with the stochastic updates

$$\theta_f \leftarrow \theta_f - \eta \left(\frac{\partial \mathcal{L}_y^i}{\partial \theta_f} - \lambda \frac{\partial \mathcal{L}_d^i}{\partial \theta_f} \right), \quad (5.7)$$

$$\theta_y \leftarrow \theta_y - \eta \frac{\partial \mathcal{L}_y^i}{\partial \theta_y}, \quad (5.8)$$

$$\theta_d \leftarrow \theta_d - \eta \frac{\partial \mathcal{L}_d^i}{\partial \theta_d}, \quad (5.9)$$

where η is the learning rate.

The updates (5.7)–(5.9) are very similar to stochastic gradient descent (SGD) updates for a feedforward deep model that is made of a feature extractor fed into a label predictor and into a domain classifier. The difference lies in the $-\lambda$ factor in Equation (5.7), which is important, as otherwise SGD would try to make features dissimilar across domains in order to minimize the domain classification loss.

Although direct implementation of (5.7)–(5.9) as SGD is not possible, it is highly desirable to reduce these updates to some form of SGD, since SGD and its variants are the main learning algorithms implemented in most deep learning libraries (e.g. TensorFlow, PyTorch).

Such a reduction can be achieved by introducing a novel *gradient reversal layer* (GRL) defined as follows. The gradient reversal layer has no parameters associated with it (apart from the hyperparameter λ , which is not updated by backpropagation). During forward propagation, the GRL acts as the identity function. During backpropagation though, the GRL takes the gradient from the subsequent level, multiplies it by $-\lambda$ and passes it to the preceding layer. Implementing such a layer using existing deep learning libraries is simple, as defining procedures for forward propagation (identity function), backpropagation (multiplying by a scalar), and parameter update (none) is trivial.

The GRL as defined above is inserted between the feature extractor and the domain classifier, resulting in the architecture depicted in Figure 5.1. As the backpropagation process passes through the GRL, the partial derivatives of the loss that is downstream the GRL (i.e., $\mathcal{L}_d$) w.r.t. the layer parameters that are upstream the GRL (i.e., θ_f) get multiplied by $-\lambda$. Therefore, running SGD in the resulting model implements the updates (5.7)–(5.9) and converges to a stationary point of Equation (5.2).

More formally, we can treat the GRL as a function $R_\lambda(\mathbf{x})$ defined by two equations describing its forward and backward propagation behavior:

$$R_\lambda(\mathbf{x}) = \mathbf{x} \quad (5.10)$$

$$\frac{dR_\lambda}{d\mathbf{x}} = -\lambda \mathbf{I} \quad (5.11)$$

where $\mathbf{I}$ is the identity matrix. We can then define the objective function of $(\theta_f, \theta_y, \theta_d)$ that is being optimized by SGD as

$$\tilde{\mathcal{L}}(\theta_f, \theta_y, \theta_d) = \sum_{i=1}^{N_S} \mathcal{L}_y^i(\theta_f, \theta_y) + \sum_{i=1}^N \tilde{\mathcal{L}}_d^i(\theta_f, \theta_d), \quad (5.12)$$

where the domain classification loss for the i th training example $\tilde{\mathcal{L}}_d^i$ is defined as

$$\tilde{\mathcal{L}}_d^i(\theta_f, \theta_d) = \mathcal{L}_d(G_d(R_\lambda(G_f(\mathbf{x}_i; \theta_f)); \theta_d), d_i). \quad (5.13)$$

Note that we do not anymore multiply the second term of Equation (5.12) by λ , as we did in Equation (5.2), since this factor is instead now reflected in Equation (5.13) through Equation (5.11).

Running updates (5.7)–(5.9) can then be implemented as doing SGD for Equation (5.12) and

leads to the emergence of features that are simultaneously domain-invariant and discriminative. After training, the label predictor

$$y(\mathbf{x}) = G_y(G_f(\mathbf{x}; \theta_f); \theta_y) \quad (5.14)$$

can be used to predict labels for examples from the target domain (as well as from the source domain).

5.4 Application to Traffic Sign Classification

A variety of experimental results are reported in the original DANN papers [92, 86], dealing with different *classification* tasks in both shallow and deep neural network architectures. However, the authors do not experiment with object detection tasks. As such, in this section we relay an experiment on traffic sign *classification* carried out by [92, 86], as well as the results that they achieved.

In this experiment, the label predictor predicts the 43 classes of the German Traffic Sign Recognition Benchmark (GTSRB) [94], which are illustrated in Figure 5.2. For the source domain, 100,000 synthetic images were obtained, simulating various imaging conditions. For the target domain, out of the more than 50,000 images of the GTSRB, 31,367 were sampled at random for training, while the rest was used for evaluation.



Figure 5.2: Examples of the 43 classes of the German Traffic Sign Recognition Benchmark (GSTRB) dataset [94].

5.4.1 Baselines

As discussed previously in Section 4.1.2 on page 46, the following baselines are evaluated in these experiments. The *source-only* model is trained without consideration for target-domain

data (no domain classifier branch included into the network). This model serves as a lower bound. The *target-only* model is trained on the target domain with labels revealed. This model serves as an upper bound.

Furthermore, DANN is compared against the subspace alignment (SA) technique that we describe in Section 2.5.4 on page 21. To boost the performance of this baseline, its only hyperparameter (i.e., the number of principal components) is picked from the range $\{2, \dots, 60\}$, so that the test performance on the target domain is maximized. To apply SA in this setting, a source-only model is trained and then the activations of the last hidden layer in the label predictor (before the final classifier) are considered as features. Then, the mapping between the source and the target domains is learned.

5.4.2 Experimental Setup

Figure 5.3 shows the neural architecture used for deep domain adaptation. For feature extraction and classification, the authors take inspiration from [95]. For the domain classifier, three fully-connected layers ($f \rightarrow 1024 \rightarrow 1024 \rightarrow 1$) are used: two ReLU-activated hidden layers followed by a sigmoid unit. Per the authors' admission, these architecture choices for the domain classifier are arbitrary and better adaptive performance might be attained with more careful choices.

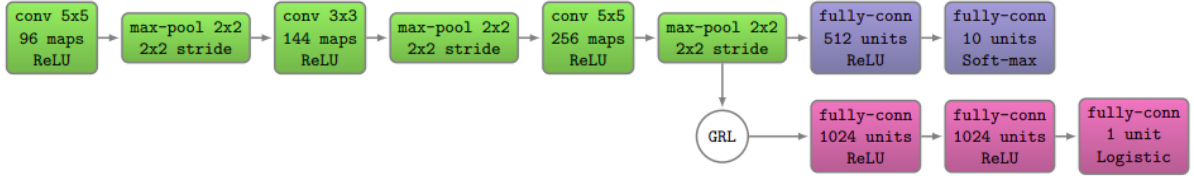


Figure 5.3: The architecture of DANN as used for the GSTRB dataset. Note that the color-coding is the same as in Figure 5.1.

As it pertains to the loss functions, the label prediction loss $\mathcal{L}_y$ is set to be the multinomial cross-entropy loss, such that

$$\mathcal{L}_y = \frac{1}{N_S} \sum_{i=1}^{N_S} \sum_{k=1}^C -y_{ik} \log(\hat{y}_{ik}). \quad (5.15)$$

Similarly, the domain classification loss $\mathcal{L}_d$ is defined as the binomial cross-entropy loss, with

$$\mathcal{L}_d = \frac{1}{N} \sum_{i=1}^N -d_i \log(\hat{d}_i) - (1 - d_i) \log(1 - \hat{d}_i). \quad (5.16)$$

With regard to the scalar hyperparameters, the learning rate is adjusted during gradient descent such that

$$\eta_p = \frac{\eta_0}{(1 + \alpha p)^\beta}, \quad (5.17)$$

where p is the training progress linearly changing from 0 to 1, $\eta_0 = 0.01$, $\alpha = 10$, and $\beta = 0.75$. This schedule promotes convergence and low error on the source domain. A momentum term of 0.9 is also used.

The domain regularizer λ is initialized at 0 and gradually changed to 1 using the schedule

$$\lambda_p = \frac{\lambda_0}{1 + \exp(-\gamma p)} - 1, \quad (5.18)$$

where $\lambda_0 = 2$ and $\gamma = 10$. This schedule allows the domain classifier to be less sensitive to noisy signal at the early stages of the training procedure. Note that these λ_p are only used for updating the feature extractor G_f , while the domain classifier G_d uses a constant $\lambda = 1$. This ensures that it trains as fast as the label predictor G_y .

5.4.3 Results

Table 5.1 compares the baselines to DANN in terms of the results obtained. For each algorithm, the classification accuracy is reported for the target domain, as well as the proportion of the gap covered between the lower-bound and upper-bound models, which is computed as

$$\text{gap} = \frac{\mathcal{P}_{\text{DA}} - \mathcal{P}_{\text{lower}}}{\mathcal{P}_{\text{upper}} - \mathcal{P}_{\text{lower}}}, \quad (5.19)$$

where $\mathcal{P}_{\text{DA}}$ is the performance of the domain adaptation model, while $\mathcal{P}_{\text{lower}}$ and $\mathcal{P}_{\text{upper}}$ are respectively the performance of the lower-bound and upper-bound models.

As the main focus of this section, the experiment pertaining to traffic sign classification sees DANN outperforming both the lower bound and the SA baseline, with an accuracy of 0.8865, meaning it covers 46.4% of the gap between the source-only model (lower bound) and the target-only model (upper bound). Interestingly though, even as a relatively simple technique, SA is able to outperform the source-only model and covers 12.7% of the gap.

Model	Traffic signs		House numbers	
	Accuracy	Gap	Accuracy	Gap
Source only	79.00%	—	86.74%	—
SA	81.65%	12.7%	86.44%	—5.5%
DANN	88.65%	46.4%	91.09%	79.7%
Target only	99.80%	—	92.20%	—

Table 5.1: Performance in classification tasks for different models reported on the target domains. On top of accuracy, the proportion of the gap covered between the source-only and the target-only models is also reported. Furthermore, we include the results of another experiment to adapt synthetic numbers to the Street View House Numbers (SHVN) dataset [96].

The last two columns of Table 5.1 show the results of a different experiment, in which synthetic numbers are adapted to images taken from the Street View House Numbers (SHVN) dataset [96]. In this experiment, SA performs worse than the source-only model and is -5.5% off from entering the gap. Meanwhile, DANN covers the gap by 79.7% with an accuracy of 0.9109.

Combined with the results of other experiments, this shows that, when applied to image classification, not only does DANN provides good adaptive performance, but it is also able to largely outperform SA, even when the latter performs worse than a source-only model.

In the next chapter, we put DANN in practice and explore how we can leverage it to improve the performance of a YOLOv3 model.

6 Experimentation

In this chapter, we discuss the different experiments that we conducted with DANN and the results obtained on the target domain.

Recall from Section 4.1 that a YOLOv3 model trained on the source domain has an average precision (AP) of 28.67%, while the same model trained on the target domain has an AP of 44.37%. These values respectively represent the non-adaptive lower-bound and upper-bound baselines of our research task, and we expect a good DANN-based model to perform at least as well as the lower bound.

Similarly, Section 4.2 gives a simple adaptive baseline based on subspace alignment, which gets a 23.46% AP. Given that its performance is lower than that of a source-only model, we concluded that it is not interesting to put it in practice in our specific case. However, we can still leverage it as a reference point for any future deep adaptive model: if such a model is more complex than subspace alignment (SA) and yet performs worse, we can trivially argue that it is better to fall back to SA, as SA both performs better and is simpler to implement.

Note that we focus here solely on the unsupervised case. As such, bounding box annotations for the target domain data are hidden during training.

6.1 Setup

This section describes how we modified an existing YOLOv3 implementation to incorporate a domain classifier and train it following a DANN architecture.

6.1.1 Retrofitting a Domain Classifier in YOLOv3

As described in Section 2.4.1, YOLOv3 uses a Darknet-53 body to extract features from the input. These features are then processed by three different necks (corresponding to three different scales), before being fed into the final box predictors (see Figure 2.1). Following this, there exists different possible choices regarding where and how to insert a domain classifier into this base architecture.

Here, we present three architectural configurations we experimented with. As you will see, all configurations rely on placing the domain classification component at the exit of the three necks.

One domain classifier. In this first configuration, we flatten the output of each prediction head before concatenating all three outputs together. The resulting vector is then passed to the domain classifier. Figure 6.1 illustrates this configuration.

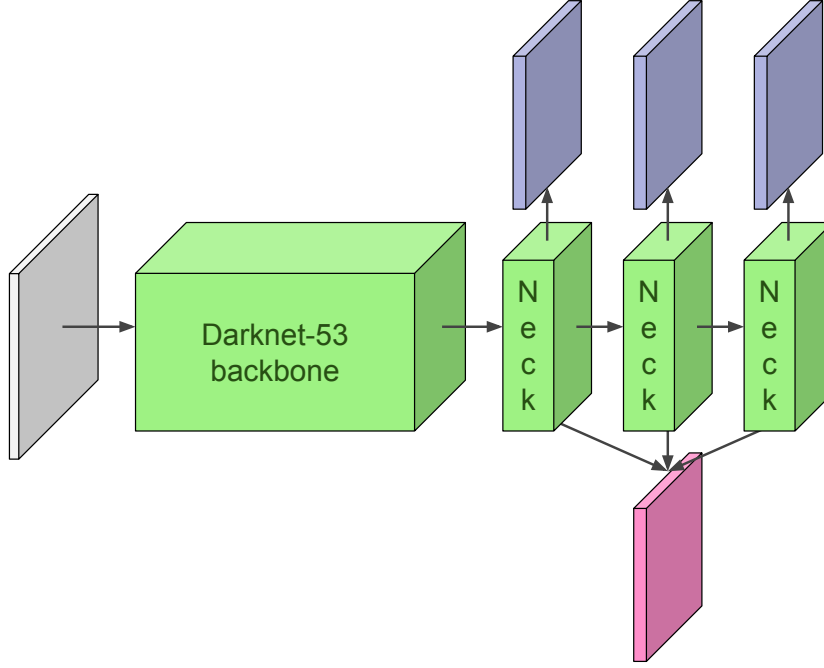


Figure 6.1: Our DANN architecture with one domain classifier connected to all three prediction heads (which outputs are concatenated together). For clarity, the gradient reversal layer is not included in the illustration. Also note that the color-coding is the same as in Figure 5.1: feature extraction layers are green, box predictors are blue, and the domain classifier is magenta.

Three domain classifiers. This configuration is inspired from the architecture of YOLOv3 itself: since YOLOv3 uses one convolutional layer per scale for bounding box prediction, it might be beneficial to apply the same idea to domain classification. As shown in Figure 6.2, instead of concatenating the outputs of the three necks into one vector, we feed these outputs into three different instances of the domain classifier. The resulting architecture is such that three domains are predicted for a given input image (i.e., one per scale). Note that in this case, we optimize the average loss of the three classifiers.

Three domain classifiers with shared weights. Depending on the complexity of the domain classifier, the previous configuration may turn out to be memory-intensive and slow to converge, due to the large number of parameters to train. As such, a simple modification involves sharing the trainable parameters across the three domain classifier instances. This way, only one set of weights (instead of three) remains for all three instances.

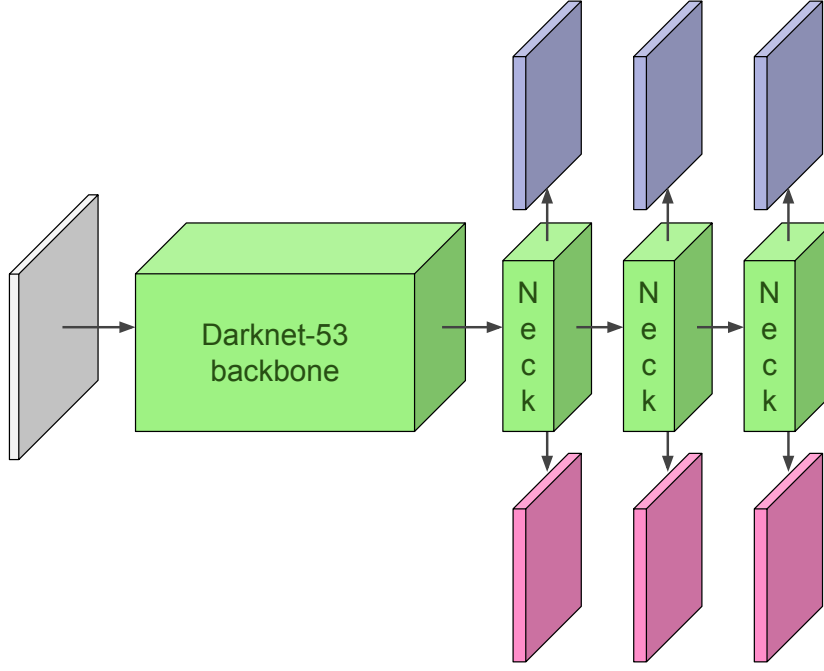


Figure 6.2: Our DANN architecture with three domain classifiers, each one connected to a different neck component.

6.1.2 Domain Classifier Architecture

Another important choice relates to the internal architecture of the domain classifier. The goal is to build a domain classifier such that it is indeed able to discriminate between the two domains while simultaneously being trainable in a reasonable amount of time and avoiding overfitting.

Note that for all architectures, we use the binomial cross-entropy loss function as defined by Equation (5.16). In architectures with three domain classifiers, we minimize the average loss of the classifiers.

Two-layer domain classifier. This first architecture is the simplest of all and consists in two fully-connected hidden layers with 256 units each. Both layers employ ReLU activation and L2 regularization. At the end of the network, a final sigmoid-activated unit outputs a domain prediction. Furthermore, we add a dropout layer between each fully-connected layer, following [86].

Three-layer domain classifier. This is a simple modification of the previous architecture where we add a new 256-unit fully-connected layer to the existing architecture. As is explained later, complexification of the domain classifier helps it learn better.

Five-layer domain classifier. Here, we significantly complexify the previous infrastructures by replacing it with five fully-connected hidden layers. On top of this, each layer now has 512 units instead of 256.

6.1.3 Training from Scratch vs. Fine-Tuning

Another contribution relates to the starting point of the weights of the network and significantly impacts performance.

In the original papers, DANN is considered as a standalone model, that is, such that it does not rely on any previously trained model to adapt to the target domain. In this sense, DANN effectively learns the target domain in parallel with the source domain; we say it trains **from scratch**. However, this approach means that we do not take advantage of any pre-existing knowledge learned from the source domain.

Following the above observation, we design a new training procedure such that, instead of optimizing the model from scratch, we import and **fine-tune** the weights from our source-only model (on top of learning a new domain classifier). This has the advantage of initially boosting the performance of our model on source data and thus reduces the amount of “noisy signal at the beginning of the training procedure,” which is an important consideration with DANN [86], as is explained later in this chapter. Of course, in this case no layer is frozen when fine-tuning.

6.1.4 Domain Regularizer

A crucial hyperparameter of DANN is the domain regularizer λ , which controls the trade-off between the box predictor and the domain classifier. As noted in the previous chapter, the DANN authors use the schedule

$$\lambda_p = \frac{\lambda_0}{1 + \exp(-\gamma p)} - 1, \quad (5.18)$$

where $\lambda_0 = 2$ and $\gamma = 10$. We use the same schedule, but tune λ_0 and γ during learning.

6.1.5 Hyperparameter Tuning

In this subsection, we first present the cross-validation protocol employed with DANN, before describing how we tune the hyperparameters. It is important to note that all aspects not otherwise discussed may be assumed to mirror the training protocol for our YOLOv3 baseline, described in Section 4.1.1.

Cross-Validation

The main modification pertains to the number of training examples seen per epoch. Recall that for our YOLOv3 baseline, the entirety of the training set is seen by the learning algorithm during each epoch, taking approximately 24 hours in total over 30 epochs. This means that the number of iterations done per epoch is given as

$$\text{number of iterations} = \left\lceil \frac{\text{training set size}}{\text{batch size}} \right\rceil = \left\lceil \frac{25,408}{16} \right\rceil = 1,588. \quad (6.1)$$

This approach is typical and is used by most deep learning algorithms. However, it is obviously very time-consuming when doing hyperparameter tuning, as testing e.g. 10 sets of hyperparameters would take about 10 days!

To solve this problem, we fix the number of iterations to 64 such that it allows us to train a single hyperparameter set in about 1 hour. This indeed implies that the learning algorithm sees $64 \cdot 16 = 1,024$ training examples per epoch, instead of 25,408. To ensure that all training examples are seen at least once over the 30-epoch training session, the content of each batch is sampled uniformly at random from the training set.

Search

Before we describe the hyperparameters used in our models, it is important to discuss how they were selected. Indeed, instead of using an exhaustive grid search, we selected the parameters through random search. The motivation is twofold:

First, random search allows us to specify exactly the number of hyperparameter sets tested, and thus gives us control over the duration of the tuning phase. Compare this to grid search, which duration gets greatly impacted if we modify the range of one of the parameters, or worse yet, if we add a new hyperparameter. Using grid search would effectively limit our willingness to try new things, as they would significantly slow down our work; this would have been especially true at the beginning of the project, when the hyperparameters used were not set in stone yet.

Second, even though one might argue that grid search is exhaustive and thus finds better solutions than random search, research shows otherwise. Indeed, [97] gives theoretical and empirical evidence that random search is more efficient than grid search, under some assumptions. The paper shows that, over the same domain, selecting hyperparameters randomly leads to models that are at least as good as those found with grid search, within a small fraction of the computation time. Furthermore, the authors argue that by running random search as long as grid search, random search finds better models by searching a larger hypothesis space.

In conclusion, random search gives us the advantage of being able to control the duration of the tuning phase, with no performance penalty compared to grid search.

Hyperparameters

Aside from the different architectures mentioned earlier, we search among a set of four scalar hyperparameters during the tuning stage. Note that these hyperparameters only concern the domain classifier, as the rest of the network is left unchanged compared to the baseline. During the search, all four parameters are sampled uniformly at random from the ranges given in the following:

Domain regularization λ_0 . As described earlier, this parameter directly influences how much λ weighs in the final loss function. We select it in the range $\{2, \dots, 20\}$.

Domain regularization γ . Similarly, this parameter influences the form of the loss function as a function of the training progress. We select it in the range $\{5, \dots, 20\}$.

L2 regularization factor. Weight regularization controls the trade-off between the bias of our domain classifier and its variance. We select it in the interval $[10^{-4}, 10^4]$.

Dropout rate. As a separate way of managing overfitting, we allow between 0 and 100% of activation units to be dropped.

6.2 Results

With all the different aspects of our experimental setup established, we now share the most interesting results. This section represents a chronological account of our experiments, as well as the motivation behind them and their impact on performance.

The evaluation procedure we use is standard and is the same as for our baseline models. For each architecture described, we select the model that performs the best on the validation set. We then retrain the model on the training and validation sets. We then report the final performance on the test set.

At the end of this section, Table 6.1 provides a summary of the experimental results obtained on the target domain. Furthermore, Appendix A discusses results obtained when performing training and evaluation independently on different combinations of the two domains.

6.2.1 Initial Model

For our first experiment, we chose the simplest configuration options among those presented previously. The reason is of course that we were interested in first figuring out how a simple DANN-based model would perform, instead of immediately diving into time-consuming experiments and optimizations.

In this optic, the first model used was built using one domain classifier connected to the three outputs of the feature extractor, as it is the easiest configuration to implement. Regarding the

internal architecture of this domain classifier, we used the two-layer configuration described earlier. Furthermore, we used fixed hyperparameter values of $\lambda_0 = 2$ and $\gamma = 10$, mirroring DANN’s authors, while disabling L2 regularization and dropout. Most importantly, at this point we had not yet come up with the idea of fine-tuning the source-only model. As such, we trained DANN from scratch (following the two-stage approach of Section 4.1.1).

With no big surprise, this first tentative model performs poorly and only obtains 13.9% AP on the target domain. This is worse than the performance of our source-only model (28.67%), as well as that of our subspace alignment model (23.46%).

6.2.2 Tuning the Initial Model

Seeing as though our initial model performs poorly with arbitrarily chosen hyperparameters, an obvious idea is to try and tune a model using an identical neural architecture but different hyperparameters. The assumption is that the architecture may be good enough to obtain good results, and that the main obstacle to that end is the set of scalar hyperparameters.

We therefore tuned the model using the protocol defined in Section 6.1.5. We found an increase of 7.43 percentage points (p.p.) in terms of AP compared to the previous model, leading to a 21.33% AP. This validates our assumption that the architecture itself is viable, and that it can learn better given better-chosen hyperparameters. However, the performance of this model is still 2.13 p.p. less than that of our SA baseline.

6.2.3 Fine-Tuning the Source-Only Model

Seeing as though we saturated the performance of the current model architecture, an obvious idea would have been to research how to complexify the domain classifier in order for it to learn better (under the assumption that it is the bottleneck). Instead, this is when we first had the idea of importing the weights from the source-only model to fine-tune those, rather than training the model from scratch, seeing as it might lead to a quick win.

Doing so indeed yields a large performance improvement and brings the AP to 31.87%. We tentatively argue that the features learned on the source domain initially help the domain classifier discriminate, which in turns leads the feature extractor to learn better domain-invariant features. This is our first truly significant result, as it shows that:

- (i) Domain adaptation using DANN is able to give better results (31.87%) than the lower-bound baseline given by non-adaptive learning on the source domain data (28.67%). Recall from Section 5.4.3 how, in the literature, the performance of domain adaptation technique is expressed in terms of the gap covered between a lower-bound baseline (28.67%) and an upper-bound baseline (44.37%). With this model, we can calculate that the gap is covered at 20.38%.

- (ii) Because our SA model performs worse than the source-only model, the previous point necessarily implies that our DANN model outperforms our SA model. This is meaningful as it signifies that implementing DANN is worth the extra effort compared to SA.

However, one might wonder whether this performance increase is mainly due to the fact this domain adaptation model gets a headstart compared to the source-only model. In other words, the question is whether training the source-only model for longer (without adaptation) could lead to the same performance improvement seen here. To test this hypothesis, we further train the source-only model (i.e., with no adaptation) for 30 more epochs. This only leads to a 0.8-p.p. increase compared to the original source-only model, allowing us to conclude that the performance gain induced by this domain adaptation model is not in caused by the headstart induced.

6.2.4 Boosting the Domain Classifier

We managed to get good first results with the previous model. However, the performance gap covered (20.38%) is still rather small, as we seek to get closer to 100%. To get better results, it is interesting to think about what currently limits the performance of the model.

To do so, we analyzed the behavior of the domain classifier throughout a training session. We found that, in its current architecture, the domain classifier mainly predicts one class (the source domain class $d_i = 0$). However, it is difficult to characterize how a “good” domain classifier should behave in general. Recall that its goal is to discriminate among domains, but at the same time the feature extractor tries to confuse the domain classifier by producing domain-invariant features. Therefore, is it better for the domain classifier to be rather good or rather bad at its task? Interestingly, not even DANN’s authors care to weigh in on this question, and instead of getting carried away with mere speculation, we leave this question open.

Nevertheless, we can emit a hypothesis. As described in the previous paragraph, the domain classifier has trouble discriminating among the domains, and its accuracy is thus close to 50%. However, this behavior only starts to manifest after a few epochs, meaning that at the beginning of the training session, it has a higher accuracy. We therefore posit that, were the domain classifier able to sustain a higher accuracy for longer, the feature extractor may be able to produce better features.

To test this hypothesis, it is necessary to build a more accurate domain classifier. To this end, we first replace the existing two-layer component with a three-layer network, having the same number of units per layer (see Section 6.1.2). We find a small but significant improvement, with the resulting model obtaining an AP of 32.47% (compared to 31.87% previously). Complexifying the domain classifier further (while being careful not to saturate our GPU’s memory) using a five-layer domain classifier with 512-unit layers yields a 33.74% AP. With both architectures, we observe that the domain classifier sustains a high accuracy for longer than with the previous architecture, thus validating our hypothesis.

6.2.5 Multiple Domain Classifiers

As we covered before, YOLOv3 makes use of three box prediction heads at the end of the network. The rationale behind such a setup is that Darknet-53 outputs feature maps at three different scales, with each head specializing in a different scale and producing the bounding boxes for the objects that it detected at that scale. This detection task is of course different from the classification done by our domain classifier. Yet we reuse the same architecture on the basis that each domain classifier can be considered as voting for a given scale. Note that we initially share the weights of each domain classifier in an attempt to lower the memory footprint.

In practice, this approach gives the best results so far by a large margin, delivering an AP of 38.29% (+4.55 p.p. compared to the previous model). To explain the reasons behind this improvement, we conjecture that each of the three domain classifiers is indeed able to specialize in a given scale, whereas a single domain classifier is unable to discriminate when the three feature maps are concatenated with each other. This could be explained by the complexity of the domain classifier: all domain classifiers have the same complexity (i.e., 5×512 units), and in the three-classifier architecture, each classifier handles approximately three times smaller feature maps than in the one-classifier architecture.

In a second three-classifier experiment, we make each domain classifier independent from each other (i.e., they do not share weights). The assumption is that the features learned by the feature extractor may be different enough that they need independent networks for accurate classification. In practice, we have found that such a configuration does not yield improved performance. As such, we may conclude that the domain classifiers are not able to specialize for a specific feature map scale. Rather, the features learned seem to be similar across scales, and a domain classifier indeed learns from training with additional data.

Method	Fine-tuned	Hypertuned	Layers	Classifiers	AP	Gap
Source only	—	—	—	—	28.67%	—
SA	—	Yes	—	—	23.46%	—33.18%
DANN	No	No	2	1	13.90%	—94.08%
DANN	No	Yes	2	1	21.33%	—46.75%
DANN	Yes	Yes	2	1	31.87%	20.38%
DANN	Yes	Yes	3	1	32.47%	24.2%
DANN	Yes	Yes	5	1	33.74%	32.29%
DANN	Yes	Yes	5	3	38.29%	61.27%
Target only	—	—	—	—	44.37%	—

Table 6.1: A summary of the experimental results we obtained on the target domain in this thesis. The source-only and target-only models refers to the non-adaptive baseline describe in Section 4.1. SA refers to the subspace alignment baseline detailed in Section 4.2. The DANN models refer to the different configurations detailed in Sections 6.2.1–6.2.5, in their respective order.

6.2.6 Other Experiments

On top of the experiments above, we had several other ideas when experimenting with DANN. Unfortunately, those ideas did not end up giving satisfying results. We briefly describe them in this section.

Moving the gradient reversal layer. Notice that in all the different architectures that we experimented with, the gradient reversal layer (and thus the subsequent domain classifier network) is placed at the exit of YOLOv3’s neck. This is a design choice that was guided by the decision made by the original authors to place the box predictor at that position, and nothing forgoes inserting the gradient reversal layer (GRL) at a different position in the network. As such, we considered feeding the feature map output by the Darknet-53 backbone directly into the GRL. This architecture is such that only one feature map is passed to the GRL, and thus only one domain classifier is needed (see Figure 6.3). This choice is based on the fact that Darknet is indeed a feature extractor, and as such it might produce features that are more interesting for a simple classifier, whereas YOLOv3’s neck is more fitted to a detection task. While an interesting premise, implementing such architecture did not show an improvement over the five-layer model described in Section 6.2.4.

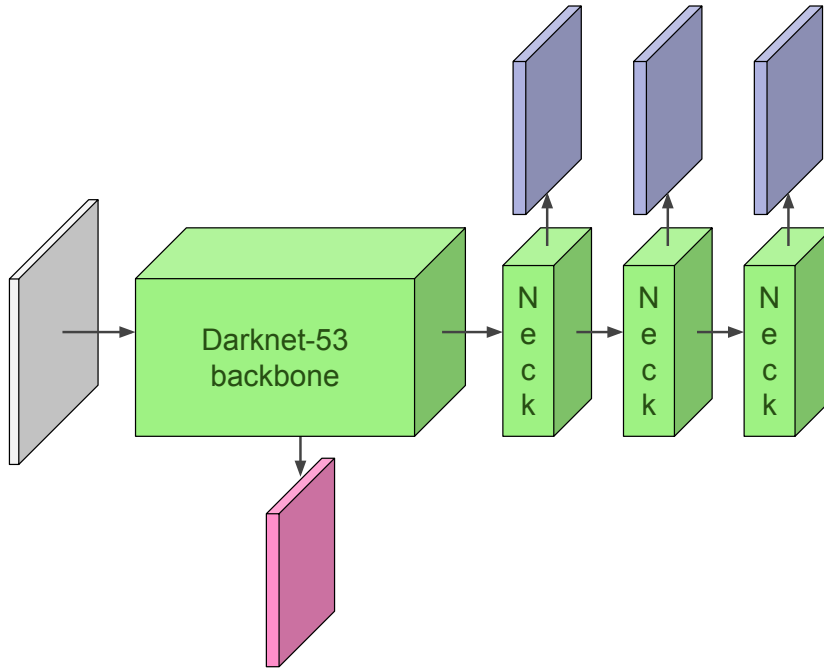


Figure 6.3: Our DANN architecture with a domain classifier inserted directly after Darknet.

Domain-balanced batches. Recall from Section 3.3.2 that one third of our images belong to the source domain, while the rest belong to the target domain. One may thus wish to reduce this imbalance in order to improve the performance of the domain classifier [66].

To this end, we modify the sampling procedure described in Section 6.1.5 such that, in each generated batch, the proportion of source domain data is equal to that of target domain data. Unfortunately, this did not improve the performance of the model.

Domain-weighted loss. Similarly to the previous point, another way to deal with class imbalance is to give more weight to examples from the minority class relative to those of the majority class [98]. In our case, this is equivalent to more strongly penalizing our model for errors made on the target domain. More formally, we define a weighted loss function $\mathcal{L}_d^w$ such that

$$\mathcal{L}_d^w = \frac{1}{N} \sum_{i=1}^N w(d_i) \left(-d_i \log(\hat{d}_i) - (1-d_i) \log(1-\hat{d}_i) \right), \quad (6.2)$$

where $w: \{0,1\} \rightarrow \mathbb{R}$ gives the weight factor for the specified domain. Following [98] and [99], we define w as

$$w(d) = \frac{N}{2 \cdot ((1-d)N_S + dN_T)}, \quad (6.3)$$

giving $w(0) = 0.75$ and $w(1) = 1.5$. We also tried to match the domain distribution of our dataset with $w(0) = 1/3$ and $w(1) = 2/3$. In both cases, the domain classifier did not improve compared to one with unit weights in the loss function. Along with the result obtained in the previous paragraph, this suggests that the domain classifier is not sensitive to the observed domain class imbalance.

In the next chapter, we conclude this thesis and raise some potential future work areas.

7 Conclusion

At the end of this thesis, we have been able to gain a better understanding of the fields of domain adaptation, computer vision, and deep learning, as well as some the challenges and different approaches. In particular, we successfully implemented a deep domain-adaptive approach to solve the task of traffic sign detection for our particular dataset.

As a first step in our journey, we had to deal with data quality issues. These issues stemmed mainly from the ground-truth data provided by our human annotators and the way that they were processed. The most pressing problem arose because we employed multiple annotators without accounting for the resulting duplicated annotations, thus skewing the dataset. We solved this problem by proposing a novel algorithm (based on non-maximum suppression) capable of merging competing annotations for an image.

We then focused on answering our first research question regarding how to split our dataset between a source domain and a target domain. We evaluated metrics such as image luminance and image sharpness. However, methods for calculating these metrics and their thresholds seemed too arbitrary and did not yield satisfying results in terms of their discriminative power. Solar altitude was also investigated as a more objective metric, since this can accurately measured and precise twilight thresholds exist in the literature. This was however overshadowed by the discovery that our image dataset in fact came from two different cameras, each exhibiting distinct image quality profiles. We therefore used this criterion as the basis for the domain split of our dataset.

Following this, we trained baseline, non-adaptive YOLOv3 models that would be used to benchmark the performance of domain-adaptive approaches. In particular, we trained YOLOv3 separately on the source domain and on the target domain, with the assumption that domain adaptation should close the gap between source-only model (lower bound) and the target-only model (upper bound). We found that, on the target domain, the source-only model obtained 28.67% AP, while the target-only model obtained 44.37% AP.

Our first interesting result came from the implementation of a domain adaptation algorithm based on subspace alignment using PCA. Recall, that one of our research questions asked whether shallow adaptation approaches could work in our traffic sign detection problem. We chose the SA algorithm because it was easy to implement and had shown good results in classification tasks. However, evaluating this algorithm on our dataset resulted in only 23.46% AP on the target domain, which is worse than our lower-bound model. This result shows that subspace alignment is ill-suited for our problem, as using a non-adaptive model trained solely on the source domain yields better performance. We conjecture that the features learned by an object detector are too specific to be effectively adapted by such a simple

method as subspace alignment.

Next, we discussed the implementation of DANN, an approach to deep domain adaptation that embeds domain the process of deep feature learning such that the final learning decisions are made based on features that are both discriminative for the main learning task, and invariant to domain shift. DANN showed better results than SA in previous traffic sign classification work, but it was not applied to the task of detection by the original authors.

Without surprise, we found that obtaining good performance with DANN required careful tuning of the hyperparameters and the network architecture. In particular, a major change in the training procedure compared to the original paper consisted in training not from scratch, but from the source-only baseline model. This change boosted the performance of DANN over the lower bound to reach 31.87% AP. This was significant as it means that domain adaptation is indeed viable for our problem.

We were further able to increase the performance of our model by complexifying its domain classifier. This modification stemmed from the observation that the domain classifier had trouble making truthful predictions during training, thus we hypothesized that a more complex classifier may be able to better discriminate among domains. This led to our model obtaining 33.74% AP on the target domain.

In a final experiment, we used three domain classifiers instead of a single one. This emulates YOLOv3’s approach to detection, where features are learned at three different scales, each having a different prediction head. This change proved significantly effective and was able to bring performance up to 38.29% AP. This implies that domain adaptation is in our case capable of covering 61.28% of the performance gap formed by the source-only model and the target-only model. We further tried different side experiments such as moving the location of the gradient reversal layer, and domain-balancing the batches or the loss function, to no avail.

7.1 Future Work

Here, we propose some ideas for future research areas that we think could be interesting to investigate:

Convolutional domain classifier. In our current model, we only use fully-connected layers in the domain classification component of the network. Such layers are easy to design and integrate into an existing network. However, as we emulated YOLOv3 in utilizing three domain classifiers instead of a single one, it makes sense to investigate the use of convolutional layers to learn better features. This is further motivated by research in the field of image classification [31, 32, 33, 34] and our assumption that a more accurate domain classifier leads to better adaptation.

Semi-supervised adaptation. In this thesis, we worked with the assumption that the class labels for the target domains were hidden during training. This is without a doubt an interesting case and it gave us compelling evidence in favor of the value of domain adaptation. From a more practical point of view however, now that we have indeed acquired annotations for a significant amount of target images, it would be interesting to train a DANN model using this annotated data combined with more *unannotated* data, such that the algorithm learns in a semi-supervised way. The assumption is that such a model would perform even better than our current one, since we would be simultaneously revealing the ground truths for the current data, as well as adding more unannotated data.

Multi-class detection. So far, we have only looked at the general problem of traffic sign detection, that is, identifying each object as a traffic sign, or another object. However, to build a proper traffic sign inventory as described in the introduction of this thesis, it is necessary to label each traffic sign with its particular type. Applying our current model to multi-class traffic sign detection is rather trivial and could hypothetically reveal interesting insights, such as maybe the ability of domain adaptation to better predict certain types of traffic signs compared to non-adaptive methods.

A Contingency Table Discussion

In this thesis, we focused on evaluating the benefits of domain adaptation with regard only to how it is able to improve the performance of a model on the harder domain to predict, the *target* domain. This is relevant, as the target domain is generally the one for which data collection and annotation is hard, on top of having characteristics that make the data harder to process for a learning algorithm. Furthermore, in the worst case that a domain adaptation impairs the performance on the source domain compared to a baseline source-only model, this latter model can always be used as a fallback when processing data from the source domain.

However, it is interesting for the sake of completeness to discuss the interactions between the different models that we described and the subsets of data at our disposal. In this appendix, we provide a succinct discussion which reveals that, according to the assumptions we make, our models are coherent with each other, and no big surprise emerges.

In the following, we analyze Table A.1 first on a row basis, and we then look at it from a column-wise perspective.

Training	Evaluation		
	S	T	S+T
S	52.39%	28.67%	43.38%
T	45.76%	44.37%	45.08%
S+T	57.20%	44.64%	52.38%
DA	58.92%	38.29%	50.11%

Table A.1: A contingency table comparing different models evaluated on multiple test sets. In the leftmost column, S and T refers to non-adaptive models trained respectively on the source and target domains only (i.e., the baselines described in Chapter 3). S+T refers to the same model trained on both domains. DA refers to the best DANN model trained on labeled source data and unlabeled target data (see Chapter 6). Similarly, the subsequent columns give the AP for each model respectively evaluated on the source domain, the target domain, and both.

Let us first make clear that the results obtained in the S+T row, where we see that a model trained on the entire dataset performs better on the source domain than on the target domain, were already partly established in Chapter 3. Indeed, this is the basis of our work and shows that our domain split criterion in fact discriminates between two domains of different

predictive complexity. What we did not show, however, is that the same model *evaluated* on the entire dataset performs (i) worse than on source domain and (ii) better than on the target domain ($44.64\% < 52.38\% < 57.20\%$). Without surprise, (i) is explained by the fact that S+T has more images that are further complicated compared to S. Similarly, we can explain (ii) by noting that S+T includes more easier images compared to T.

This understanding makes our analysis of the first two rows easy. Indeed, we can conclude that the source-only model performs better on the source domain than on the target domain ($52.39\% > 28.67\%$), because the former evaluation set shares the same distribution as its training set and contains easier images. Evaluation on the entire dataset gives an AP in the middle (43.38%), for the reasons given before.

Regarding the target-only model, it follows the trend and performs better on the source domain than on the target ($45.76\% > 44.37\%$), with full-set evaluation giving performance in between (45.08%). However, one might ask why it does not perform best on T, which shares the same domain. We contend that this is due to the source domain containing images that are easier to predict, as shown by rows S and S+T, such that the target-only model’s specialization power for its own domain is not able to overcome this. In other words, even though the target-only model specializes for its own domain (target), the source images are much easier to predict, and it performs better on those. This can also explain the smaller difference between its performance on the source and target domains ($45.76\% - 44.37\%$), compared to the source-only model: for the target-only model, the performance on T should tend to saturate (as it specialized on T), even though it is slightly higher on S (as the images are easier), whereas the target-only model would not saturate its performance on T (as it specialized on S).

The last row is consistent with our previous analysis and shows that, even with a domain-adaptive model, performance is better on the source domain than on the target domain, with performance on S+T falling in between ($38.29\% < 50.11\% < 58.92\%$).

Let us now analyze the table in a column-wise fashion.

Looking at the performance of the models on the source domain, we observe that the source-only model performs better than the target-only model ($52.39\% > 45.76\%$), as expected. We also note that the S+T model performs even better (57.20%), indicating that the combination of both domains leads to learning better features. Note that this is also true when evaluating on T and S+T. Another result is that DA leads to better performance on S than any non-adaptive model (58.92%). This is interesting because, on the target domain, DA only improves on the source-only model. We conclude that, when evaluated on the source domain, it is more beneficial for a model to perform unsupervised domain adaptation, rather than to reveal the ground-truth labels for the target domain.

When evaluated on the target domain, the source-only model trained on images that were easier to process and as such performs worse than the target-only model ($28.67\% < 44.37\%$). The S+T model performs only slightly better than the target-only model (44.64%). This comforts us in our assumption that the target-only model’s performance saturates with the target domain. If it did not, adding more images from the other domain ($T = 44.37\% \rightarrow$

$S + T = 44.64\%$) would lead to a greater increase in performance, as is the case when evaluated on S ($S = 52.39\% \rightarrow S + T = 57.20\%$). As discussed in Chapter 6, DA is able to improve the performance on the target domain compared to the source-only model.

Finally, the last column, regarding evaluation on $S + T$, shows similar results. We note that DA is not able to outperform a model trained on the entire dataset ($50.11\% < 52.38\%$). This is expected as we used DANN to perform unsupervised DA.

Bibliography

- [1] Stamatia Dasiopoulou, Vasileios Mezaris, Ioannis Kompatsiaris, V-K Papastathis, and Michael G Strintzis. Knowledge-assisted semantic video object detection. *IEEE Transactions on Circuits and Systems for Video Technology*, 15(10):1210–1224, 2005. [1, 12]
- [2] Mei Wang and Weihong Deng. Deep visual domain adaptation: A survey. *Neurocomputing*, 312:135–153, 2018. [1, 2, 19]
- [3] Wanyi Li, Fuyu Li, Yongkang Luo, and Peng Wang. Deep domain adaptive object detection: a survey. *arXiv preprint arXiv:2002.06797*, 2020. [1, 2, 16, 47]
- [4] Judy Hoffman, Eric Tzeng, Taesung Park, Jun-Yan Zhu, Phillip Isola, Kate Saenko, Alexei A Efros, and Trevor Darrell. CyCADA: Cycle-consistent adversarial domain adaptation. *arXiv preprint arXiv:1711.03213*, 2017. [2]
- [5] Licheng Jiao, Fan Zhang, Fang Liu, Shuyuan Yang, Lingling Li, Zhixi Feng, and Rong Qu. A survey of deep learning-based object detection. *IEEE Access*, 7:128837–128868, 2019. [3, 13, 16]
- [6] Zhengxia Zou, Zhenwei Shi, Yuhong Guo, and Jieping Ye. Object detection in 20 years: A survey. *arXiv preprint arXiv:1905.05055*, 2019. [3, 13, 16]
- [7] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. <http://www.deeplearningbook.org>. [3, 6]
- [8] Yu Cheng, Mo Yu, Xiaoxiao Guo, and Bowen Zhou. Few-shot learning with meta metric learners. *arXiv preprint arXiv:1901.09890*, 2019. [3]
- [9] Joseph Redmon and Ali Farhadi. YOLOv3: An incremental improvement, 2018. [3, 13, 17, 42]
- [10] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. YOLOv4: Optimal speed and accuracy of object detection. *arXiv preprint arXiv:2004.10934*, 2020. [3]
- [11] EURA NOVA. <https://euranova.eu>, 2020. [Online; accessed 13-June-2020]. [4]
- [12] EURA NOVA. EURA NOVA Research. <https://research.euranova.eu>, 2020. [Online; accessed 13-June-2020]. [4]
- [13] EURA NOVA. digazu, end-to-end big data solution. <https://www.digazu.com>, 2020. [Online; accessed 13-June-2020]. [4]

- [14] Apache Software Foundation. Hadoop. <https://hadoop.apache.org>, 2019. [Online; accessed 13-June-2020]. [4]
- [15] Apache Software Foundation. Flink. <https://flink.apache.org/>, 2020. [Online; accessed 13-June-2020]. [4]
- [16] Apache Software Foundation. Kafka. <https://kafka.apache.org/>, 2020. [Online; accessed 13-June-2020]. [4]
- [17] Fernando Pérez and Brian E. Granger. IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, May 2007. [4]
- [18] Google. Kubeflow. <https://www.kubeflow.org/>, 2020. [Online; accessed 13-June-2020]. [4]
- [19] Anas Al Bassit, Sabri Skhiri, and Hejer Ammar. LEAD: A formal specification for event processing. In *Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems*, pages 91–102, 2019. [4]
- [20] Katsiaryna Krasnashchok and Aymen Cherif. Coherence regularization for neural topic models. In *International Symposium on Neural Networks*, pages 426–433. Springer, 2019. [4]
- [21] Muaz Twaty, Amine Ghrab, and Sabri Skhiri. GraphOpt: a framework for automatic parameters tuning of graph processing frameworks. In *2019 IEEE International Conference on Big Data*, pages 3744–3753. IEEE, 2019. [5]
- [22] David Poole, Alan Mackworth, and Randy Goebel. Computational intelligence. 1998. [6]
- [23] Douglas B Lenat. When will machines learn? In *Knowledge Acquisition: Selected Research and Commentary*, pages 9–11. Springer, 1989. [6]
- [24] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997. [6, 27]
- [25] Kevin Jarrett, Koray Kavukcuoglu, Marc’Aurelio Ranzato, and Yann LeCun. What is the best multi-stage architecture for object recognition? In *2009 IEEE 12th International Conference on Computer Vision*, pages 2146–2153. IEEE, 2009. [9]
- [26] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning*, pages 807–814, 2010. [9]
- [27] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the 14th International Conference on Artificial Intelligence And Statistics*, pages 315–323, 2011. [9]
- [28] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986. [10]
- [29] Claude Lemaréchal. Cauchy and the gradient method. *Doc Math Extra*, 251:254, 2012.

[10]

- [30] Yann LeCun, Patrick Haffner, Léon Bottou, and Yoshua Bengio. Object recognition with gradient-based learning. In *Shape, Contour and Grouping in Computer Vision*, pages 319–345. Springer, 1999. [11]
- [31] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012. [11, 70]
- [32] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014. [11, 70]
- [33] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015. [11, 70]
- [34] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016. [11, 70]
- [35] Yi-Tong Zhou and Rama Chellappa. Computation of optical flow using a neural network. In *IEEE International Conference on Neural Networks*, volume 1998, pages 71–78, 1988. [12]
- [36] Li Liu, Wanli Ouyang, Xiaogang Wang, Paul Fieguth, Jie Chen, Xinwang Liu, and Matti Pietikäinen. Deep learning for generic object detection: A survey. *International Journal of Computer Vision*, 128(2):261–318, 2020. [13]
- [37] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask R-CNN. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2961–2969, 2017. [13]
- [38] Andrej Karpathy and Li Fei-Fei. Deep visual-semantic alignments for generating image descriptions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3128–3137, 2015. [13]
- [39] Kai Kang, Hongsheng Li, Junjie Yan, Xingyu Zeng, Bin Yang, Tong Xiao, Cong Zhang, Zhe Wang, Ruohui Wang, Xiaogang Wang, et al. T-CNN: Tubelets with convolutional neural networks for object detection from videos. *IEEE Transactions on Circuits and Systems for Video Technology*, 28(10):2896–2907, 2017. [13]
- [40] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You Only Look Once: Unified, real-time object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 779–788, 2016. [13, 47]
- [41] Joseph Redmon and Ali Farhadi. YOLO9000: better, faster, stronger. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7263–7271, 2017. [13, 14, 47]

- [42] <https://www.deeplearning.ai>, 2020. [Online; accessed 13-June-2020]. [14]
- [43] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: Towards real-time object detection with region proposal networks. In *Advances in Neural Information Processing Systems*, pages 91–99, 2015. [15, 17, 42, 46]
- [44] Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. The Pascal visual object classes (VOC) challenge. *International Journal of Computer Vision*, 88(2):303–338, 2010. [16]
- [45] Jonathan Hui. mAP (mean average precision) for object detection. https://medium.com/@jonathan_hui/map-mean-average-precision-for-object-detection-45c121a31173, 2018. [Online; accessed 13-June-2020]. [17]
- [46] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft COCO: Common objects in context. In *European Conference on Computer Vision*, pages 740–755. Springer, 2014. [16, 17]
- [47] Yudong Liu, Yongtao Wang, Siwei Wang, TingTing Liang, Qijie Zhao, Zhi Tang, and Haibin Ling. CBNet: A novel composite backbone network architecture for object detection. *arXiv preprint arXiv:1909.03625*, 2019. [17]
- [48] Hidetoshi Shimodaira. Improving predictive inference under covariate shift by weighting the log-likelihood function. *Journal of Statistical Planning and Inference*, 90(2):227–244, 2000. [18, 51]
- [49] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, 2009. [18, 19]
- [50] Karl Weiss, Taghi M Khoshgoftaar, and DingDing Wang. A survey of transfer learning. *Journal of Big Data*, 3(1):9, 2016. [18, 19]
- [51] Wenyan Dai, Qiang Yang, Gui-Rong Xue, and Yong Yu. Self-taught clustering. In *Proceedings of the 25th International Conference on Machine Learning*, pages 200–207, 2008. [19]
- [52] Zheng Wang, Yangqiu Song, and Changshui Zhang. Transferred dimensionality reduction. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 550–565. Springer, 2008. [19]
- [53] Dan Ventura and Sean Warnick. A theoretical foundation for inductive transfer. *Brigham Young University, College of Physical and Mathematical Sciences*, 2007. [19]
- [54] Pirmin Lemberger and Ivan Panico. A primer on domain adaptation. *arXiv preprint arXiv:2001.09994*, 2020. [19, 20, 21]
- [55] Basura Fernando, Amaury Habrard, Marc Sebban, and Tinne Tuytelaars. Unsupervised visual domain adaptation using subspace alignment. In *Proceedings of the IEEE*

- International Conference on Computer Vision*, pages 2960–2967, 2013. [21, 22, 23]
- [56] Sinno Jialin Pan, James T Kwok, Qiang Yang, et al. Transfer learning via dimensionality reduction. In *AAAI*, volume 8, pages 677–682, 2008. [22]
- [57] Boqing Gong, Yuan Shi, Fei Sha, and Kristen Grauman. Geodesic flow kernel for unsupervised domain adaptation. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 2066–2073. IEEE, 2012. [22]
- [58] OpenStreetMap Wiki. Road signs in Belgium. https://wiki.openstreetmap.org/w/index.php?title=Road_signs_in_Belgium&oldid=1952214, 2020. [Online; accessed 16-March-2020]. [26, 28, 29]
- [59] Antoine Buetti-Dinh, Vanni Galli, Sören Bellenberg, Olga Ilie, Malte Herold, Stephan Christel, Mariia Boretska, Igor V Pivkin, Paul Wilmes, Wolfgang Sand, et al. Deep neural networks outperform human expert’s capacity in characterizing bioleaching bacterial biofilm composition. *Biotechnology Reports*, 22:e00321, 2019. [26]
- [60] Frank E Saal, Ronald G Downey, and Mary A Lahey. Rating the ratings: Assessing the psychometric quality of rating data. *Psychological Bulletin*, 88(2):413, 1980. [26]
- [61] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960. [27]
- [62] Joseph L Fleiss. Measuring nominal scale agreement among many raters. *Psychological Bulletin*, 76(5):378, 1971. [27]
- [63] John J Bartko. The intraclass correlation coefficient as a measure of reliability. *Psychological Reports*, 19(1):3–11, 1966. [27]
- [64] Fredrik Lundh, Alex Clark and Contributors. Pillow. <https://pillow.readthedocs.io>, 2020. [Online; accessed 21-March-2020]. [31]
- [65] International Telecommunication Union. Studio encoding parameters of digital television for standard 4:3 and wide-screen 16:9 aspect ratios. 2011. [31]
- [66] Nathalie Japkowicz and Shaju Stephen. The class imbalance problem: A systematic study. *Intelligent Data Analysis*, 6(5):429–449, 2002. [31, 67]
- [67] Shengyang Dai and Ying Wu. Motion from blur. In *2008 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8. IEEE, 2008. [32]
- [68] Said Pertuz, Domenec Puig, and Miguel Angel Garcia. Analysis of focus measure operators for shape-from-focus. *Pattern Recognition*, 46(5):1415–1432, 2013. [32, 34]
- [69] José Luis Pech-Pacheco, Gabriel Cristóbal, Jesús Chamorro-Martínez, and Joaquín Fernández-Valdivia. Diatom autofocusing in brightfield microscopy: a comparative study. In *Proceedings 15th International Conference on Pattern Recognition. ICPR-2000*, volume 3, pages 314–317. IEEE, 2000. [32]
- [70] G. Bradski. The OpenCV Library. *Dr. Dobb’s Journal of Software Tools*, 2000. [32]

- [71] Nathaniel Bowditch. *The American Practical Navigator*. National Imagery and Mapping Agency, 1802. [35]
- [72] Brandon Stafford. pysolar. <https://doi.org/10.5281/zenodo.1461066>, 2018. [Online; accessed 13-June-2020]. [35]
- [73] 14 C.F.R. § 1.1. <https://www.law.cornell.edu/cfr/text/14/1.1>. [Online; accessed 13-June-2020]. [36]
- [74] 14 C.F.R. § 121.323. <https://www.law.cornell.edu/cfr/text/14/121.323>. [Online; accessed 13-June-2020]. [36]
- [75] E. Alpaydin. *Introduction to Machine Learning*. Adaptive Computation and Machine Learning. MIT Press, 2010. [40]
- [76] NVIDIA. NVIDIA Tesla V100 GPU architecture. <https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf>, 2017. [Online; accessed 13-June-2020]. [43]
- [77] Stergios Christodoulidis, Marios Anthimopoulos, Lukas Ebner, Andreas Christe, and Stavroula Mougiakakou. Multisource transfer learning with convolutional neural networks for lung pattern analysis. *IEEE Journal of Biomedical and Health Informatics*, 21(1):76–84, 2016. [43]
- [78] Cheng Lei, Benlin Hu, Dong Wang, Shu Zhang, and Zhenyu Chen. A preliminary study on data augmentation of deep learning for image classification. In *Proceedings of the 11th Asia-Pacific Symposium on Internetware*, pages 1–6, 2019. [43]
- [79] Alexander B. Jung, Kentaro Wada, Jon Crall, Satoshi Tanaka, Jake Graving, Christoph Reinders, Sarthak Yadav, Joy Banerjee, Gábor Vecsei, Adam Kraft, Zheng Rui, Jirka Borovec, Christian Vallentin, Semen Zhydenko, Kilian Pfeiffer, Ben Cook, Ismael Fernández, François-Michel De Rainville, Chi-Hung Weng, Abner Ayala-Acevedo, Raphael Meudec, Matias Laporte, et al. imgaug. <https://github.com/aleju/imgaug>, 2020. Online; accessed 01-Feb-2020. [44]
- [80] Leslie N Smith and Nicholay Topin. Super-convergence: Very fast training of neural networks using large learning rates. In *Artificial Intelligence and Machine Learning for Multi-Domain Operations Applications*, volume 11006, page 1100612. International Society for Optics and Photonics, 2019. [45]
- [81] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4700–4708, 2017. [45]
- [82] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: Training ImageNet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017. [45]
- [83] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. [45]

- [84] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011. [45]
- [85] Rich Caruana, Steve Lawrence, and C Lee Giles. Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping. In *Advances in Neural Information Processing Systems*, pages 402–408, 2001. [45]
- [86] Yaroslav Ganin and Victor Lempitsky. Unsupervised domain adaptation by backpropagation. *arXiv preprint arXiv:1409.7495*, 2014. [47, 49, 50, 54, 60, 61]
- [87] Siqi Yang, Lin Wu, Arnold Wiliem, and Brian C Lovell. Unsupervised domain adaptive object detection using forward-backward cyclic adaptation. *arXiv preprint arXiv:2002.00575*, 2020. [47]
- [88] Mehran Khodabandeh, Arash Vahdat, Mani Ranjbar, and William G Macready. A robust learning approach to domain adaptive object detection. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 480–490, 2019. [47]
- [89] Taekyung Kim, Minki Jeong, Seunghyeon Kim, Seokeon Choi, and Changick Kim. Diversify and match: A domain adaptive representation learning paradigm for object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 12456–12465, 2019. [47]
- [90] Mandikal Vikram. A YOLO-based approach for traffic sign detection. Technical report, Department of Information Technology, NITK Surathkal, 2018. [47]
- [91] Chung Yu Wang and R Cheng-Yue. Traffic sign detection using You Only Look Once framework. *Stanford, Tech. Rep.*, 2016. [47]
- [92] Yaroslav Ganin, Evgeniya Ustinova, Hana Ajakan, Pascal Germain, Hugo Larochelle, François Laviolette, Mario Marchand, and Victor Lempitsky. Domain-adversarial training of neural networks. *The Journal of Machine Learning Research*, 17(1):2096–2030, 2016. [49, 50, 54]
- [93] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015. [49]
- [94] Johannes Stallkamp, Marc Schlipsing, Jan Salmen, and Christian Igel. The German Traffic Sign Recognition Benchmark: A multi-class classification competition. In *IEEE International Joint Conference on Neural Networks*, pages 1453–1460, 2011. [54]
- [95] Dan CireşAn, Ueli Meier, Jonathan Masci, and Jürgen Schmidhuber. Multi-column deep neural network for traffic sign classification. *Neural Networks*, 32:333–338, 2012. [55]
- [96] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. Reading digits in natural images with unsupervised feature learning. 2011. [56, 57]

Bibliography

- [97] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012. [62]
- [98] Gary King and Langche Zeng. Logistic regression in rare events data. *Political Analysis*, 9(2):137–163, 2001. [68]
- [99] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011. [68]

UNIVERSITÉ CATHOLIQUE DE LOUVAIN

École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl