ALMA MATER STUDIORUM
UNIVERSITY OF BOLOGNA

SCHOOL OF ENGINEERING AND ARCHITECTURE

MASTER'S DEGREE IN COMPUTER ENGINEERING

DOMAIN ADAPTATION AND FUSION FOR DOMAIN-SPECIFIC TRAFFIC SIGN DETECTION

Supervisors:

Prof. Michela Milano

M.Sc. Joshua Niemeijer

Author:

Eric Philip Siebenrock

Co-Supervisors:

M.Sc. Gurucharan Srinivas

Dr.-Ing. Andreas Leich

III Session

Academic Year 2019/2020

ABSTRACT

Machine Learning models are becoming useful and high-accurate in a vast area of contexts, their use can be valuable especially in road safety applications. The quality and overall characteristics of the data used to train such models plays a key role to the efficacy and accuracy of Deep Learning algorithms. However, when the applicative domain is different from the training domain, the performance will inevitably degrade, causing the algorithm to be ineffective at runtime. This thesis aims to investigate the methods and processes of data modeling and domain adaptation that are necessary to avoid a performance loss in these instances, in particular for the case of traffic sign detection in a specific domain.

This work shows that using a dataset composed of different data domains as training data can lead to an improvement of a deep learning model performance in a diverse and specific domain, and that the domain adaptation can also be used to successfully increase the accuracy of the fusion of two domains in a single dataset. In this particular case, the project aims to create a model capable of accurately recognizing traffic signs in the very specific domain of the North Rhine-Westphalia region (Germany), which is devoid of any ground truth.

The work is first focused on adapting, through modeling, both the datasets and the hyperparameters of a deep learning model. Afterwards, a technique of domain adaptation through self-supervision is designed and experimented in both directions with respect to the training datasets, in order to obtain a good accuracy for the fusion of their domains through cross pseudo-labeling. Since in many cases no ground truth can be used to evaluate the model performance change in such a particular case, a set of pseudo-metrics has also been designed, both to measure the efficacy of the domain fusion and the object detection accuracy on the NRW target domain. It is shown that the our domain adaptation technique yields a better generalization on the target domain while keeping almost the same accuracy on the source domain;

The application of semi-supervised learning through pseudo-labeling is also experimented, however the non-perfection of the predicted annotations inevitably harm the performances of the CNN model when used as training data; this shows that processes of semi-supervised learning are still ineffective in complex cases like this.

SOMMARIO

I modelli di Machine Learning stanno diventando utili e estremamente precisi in una area sempre più vasta di contesti, il loro utilizzo può essere estremamente prezioso soprattutto nelle applicazioni riguardanti la sicurezza stradale. La qualità e le caratteristiche generali dei dati utilizzati per addestrare tali modelli giocano un ruolo chiave per l'efficacia e l'accuratezza degli algoritmi di Deep Learning. Tuttavia, quando il dominio applicativo è diverso dal dominio di addestramento, le prestazioni di tali modelli peggiorano inevitabilmente, rendendoli inefficaci in fase di esecuzione. Questa tesi si propone di indagare i metodi ei processi di modellazione dei dati e adattamento del dominio che sono necessari per evitare una perdita di prestazioni in questi casi, in particolare per il caso di rilevamento di segnali stradali in domini specifici.

Questo lavoro mostra che l'utilizzo di un set di dati composto da diversi domini come dataset di addestramento può portare a un miglioramento delle prestazioni di un modello di rete neurale in un dominio diverso e specifico, e che l'adattamento del dominio può anche essere utilizzato per aumentare l'accuratezza della fusione di due domini in un unico set di dati.

In questo caso particolare, il progetto mira a creare un modello in grado di riconoscere con precisione la segnaletica stradale nel dominio estremamente specifico della regione del Nord Reno-Westfalia (Germania), che è privo di qualsiasi annotazione (o ground truth).

Il lavoro si concentra innanzitutto sull'adeguamento reciproco, attraverso la modellazione, dei set di dati e degli "iperparametri" di addestramento del modello di deep learning. Successivamente, viene progettata e sperimentata una tecnica di adattamento dei domini tramite auto supervisione in entrambe le direzioni rispetto ai dataset disponibili per l'addestramento, al fine di ottenere una buona accuratezza della fusione dei loro domini tramite pseudo-annotamento reciproco. Poiché in molti casi non vi è alcuna "ground truth" da poter essere utilizzata per valutare i miglioramenti delle prestazioni del modello in casi così particolari, è stata progettata anche una serie di pseudo-metriche, sia per misurare l'efficacia della fusione del dominio che l'accuratezza del rilevamento di oggetti sul dominio di NRW. Si mostra che la nostra tecnica di adattamento del dominio migliora la generalizzazione nel dominio target mantenendo quasi le stesse prestazioni nel dominio sorgente.

Viene sperimentata anche l'applicazione dell'apprendimento semi-supervisionato tramite pseudo-annotazioni, tuttavia la non perfezione delle annotazioni predette danneggia inevitabilmente le prestazioni del modello CNN quando vengono utilizzate come dati di addestramento; questo dimostra che i processi di apprendimento semi-supervisionato sono ancora inefficaci in casi complessi come questo.

CONTENTS

1. INTRODUCTION

With the advent of neural networks, and especially Convolutional Neural Networks (CNNs), object detection accuracy had a considerable increase, and so its applications as well. One of the most important applicative cases is the road environment and scene analysis, with the objective of intelligent driving assistance and, in future, autonomous driving.

The research project of this thesis was developed during an internship at the German Aerospace Center (DLR) and it is part of one of its projects, named “KI4Safety”.

The latter plans to efficiently classify a road junction with suitable measures which analyze the relevant features of such areas, with the objective of understanding what are the factors that mostly influence the happening of road accidents. The approach consists in performing an evaluation through *MaKaU measures*, which estimate the safety features of an intersection. AI algorithms are applied to discern the relevant features described in the measures and to build a database that is necessary to subsequently execute a statistical evaluation. In particular, the images provided by a camera mounted on the car are used as input to perform semantic segmentation of the street scene and traffic sign detection, while aerial images are used to classify the road junction. Before inserting the data in the database, the information provided by these algorithms is combined with the number of accidents that have occurred at the corresponding location, so to distinguish which are the most probable features that improve or worsen the safety of a particular intersection.

The system will then be evaluated and optimized in close cooperation with police authorities, accident commissions and planners; in the specific case, the system will be tested on the region of North Rhine-Westphalia (NRW).

In this context, the objective of the work of this thesis is to develop an accurate and reliable deep learning based object detection algorithm capable of analyzing with precision the image data relating to areas frequently subject to accidents, so to provide useful data analysis that can be exploited in the context of the *KI4Safety* project. In particular, the developed system focuses on the analysis and recognition of the traffic signs in images of the local road authority, taken in sites where car accidents have occurred.

In cases like these, the technology behind the implementation of such models are precisely Convolutional Neural Networks (CNNs).

For this purpose, there are two main datasets at disposal to train such a deep learning model: the Mapillary traffic sign dataset and the Slovenian DFG traffic sign dataset, also called data domains. Many classes of the two source datasets are in common, albeit with different names, and both also contain specific classes that are not present in the other one. First, the datasets must be modified and a CNN model must be designed, along with its hyperparameters, so that both are suited to each other. Furthermore, in this case, a CNN model is considered valuable if it is capable of recognizing the classes of all training domains, including the specific ones. To achieve such a result, the datasets have to be merged together; the technique adopted in this particular case to fuse the datasets plans to first make a deep learning model trained on one dataset generate pseudo-labels in the other one, and performing such pseudo-labeling in both directions. This process would eventually lead to having both datasets containing also the new classes specific to the other one, making it easier to fuse them in a single dataset. Techniques for evaluating the pseudo-annotations generated are also applied, through the definition of pseudo-metrics.

However, since the two datasets describe different data domains, the performance of the CNN model is likely to drop when generating the pseudo-labels on the other dataset. For this reason, strategies of domain adaptation are investigated in the context of this project, which is the main work of the thesis. There are many techniques that aim to achieve such adaptation for the model; our case adopts a quite simple, customizable but effective approach.

Such technique can then be applied both to improve the accuracy in the pseudo-labeling process and the accuracy on NRW, which is a different and unlabeled target domain.

2. BACKGROUND AND STATE OF THE ART

2.1 TensorFlow 2

The framework used for the design and development is TensorFlow 2, which is an end-to-end open source platform for machine learning.

It uses Python to provide a convenient front-end API for building applications with the framework, while executing those applications in high-performance C++.

TensorFlow allows to create dataflow graphs structures that describe how data moves through a graph, or a series of processing nodes. Each node in the graph represents a mathematical operation, and each connection or edge between nodes is a multidimensional data array, or tensor. Basically, to develop a deep learning program, a set of graph nodes (operations) and tensors (multidimensional data) must be designed. Tensors are usually created by external data (i.e. the datasets images and annotations) and are processed by the graph nodes to build and train, in this case, a deep neural network

2.2 Convolutional Neural Networks

A convolutional neural network (CNN) is a type of deep neural network that applies a convolution operation at least in one layer. CNNs are able to take an image as input, assign importance (learnable weights and biases) to various features/objects in the image and be able to differentiate one from the other.

2.2.1 Architecture of a CNN

A CNN is able to successfully capture the spatial dependencies in an image through the application of relevant filters, such dependencies would be eliminated by flattening the image and using a fully connected layer. The architecture performs a better fitting to the image dataset due to the reduction in the number of parameters involved and reusability of weights: the layers are not fully connected but are linked through the weights of the filter that “slides” over the image; as such, the filter is shared among all the spatial dimension of the input, and the only weights of the layer are the ones of the filter. Having less parameters (that is property called “sparse connectivity”) makes it harder for the network to overfit; sharing the filter weights instead of having one weight for each connection and performing a dense matrix

multiplication heavily improves the network performances and uses less memory. Such parameters sharing also makes the network invariant to translation but not invariant to scale change, which must be managed externally, as it is investigated in this case, as described later.
A filter has three dimensions: width, height and depth (which must be equal to the number of input channels), and its elements represent the layer weights that have to be optimized.
CNNs have the ability to learn filters to apply to the image, that are optimized to detect (i.e. be activated by) the features of the object to detect or classify. Previously, such filters had to be defined by hand instead.
The receptive field is another important aspect of a CNN, which is simply the number of input elements (pixels) that become the input of a node in the network (and consequently, affect its output); the receptive field depends on the filter size and stride when sliding on the input image, as such, it can vary from layer to layer.
Basically, the role of a CNN is to reduce the images into a feature representation which is easier to process, without losing features which are critical for getting a good prediction. With this architecture, initial layers typically learn low-level features from the image (e.g. edges, corners etc.), whereas deep layers learn high-level features that can make the network understand the scene structure.
Another typical layer of a CNN is the pooling layer, which is useful for denoising and extracting dominant (the most important) features that are rotation and position invariant, thus maintaining the process of effectively training the model. The pooling simply returns the maximum or average value of the portion of the input covered by the filter, depending on which type of pooling is being performed. The pooling layer is typically used also to reduce the input dimensions to a fixed size.
So a CNN layer is generally composed of at least a convolutional layer and a pooling layer, but the number of layers can vary.
After extracting a feature map, the network should learn a non-linear combination of the high-level features represented by the output of the convolutional layer; to do this the features are flattened and fed to a fully connected layer that is responsible for performing classification.

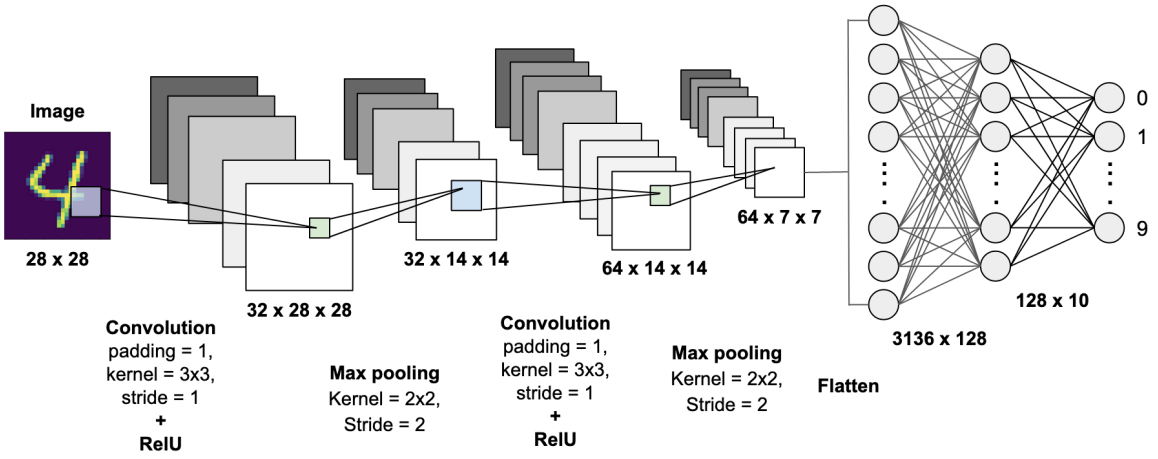


Figure 1 - Typical architecture of a convolutional neural network

2.2.2 Training of a Neural Network

The prerequisite to train a neural network with supervised training is to have a high amount of data already annotated with ground-truth information, usually the annotation process is performed manually by humans, therefore it is a time-consuming process. However, there are many datasets already endowed with ground-truth.

2.2.2.1 Backpropagation

In a very simple scheme, the process of supervised training is composed of four steps. First, a sample of data is given in input to the network, which goes through the entire neural network until it reaches the output layer where it makes the network produce an output, namely a prediction; this phase is called forward pass.

Then, the difference between the prediction made by the network and the groundtruth of the input is calculated, this value is referred to as the output loss. For each network parameter (weight or bias), the gradient of the output loss is calculated with respect to it; finally, such gradients are used to update the value of each corresponding parameter of the network. By adjusting the values of its weights, the network should “learn” to make more and more accurate predictions, so as to make the loss become negligible. This overall process is called backpropagation.

Other than having an output loss extremely low, another desirable property is to have a model that is able to generalize, which means that it is able to make good predictions on data that was not used during its training process.

2.2.2.2 Fine-Tuning Vs Training from scratch

Generally speaking, there are two approaches to train a neural network: training it from scratch or fine-tuning it. The latter is an approach of the so-called Transfer Learning.
The main difference between them is in the initialization of the network parameters (i.e. the weights).

From scratch training

When training from scratch, the model´s parameters are initialized either randomly or using a specific method, but in both cases they are intialized "blindly" with respect to the ultimate objective (i.e. the detection of objects of a specific domain).
Random initialization is generally not recommended because it can lead to a very slow training or to vanishing/exploding gradients.
To understand how that problem could happen, a simple example can help: presuming that we have a network with 250 nodes in each layer and the weights are randomly initialized with a mean of zero and a standard deviation of one. If we focus on the input received from a single node in the first hidden layer, it will be a sum of 250 terms where each term is the value of the input node multiplied by the weight. In the (very theoretical) case where all the values of input nodes are equal to one, the input to the hidden node would be equal to the sum of the weights. This sum will be normally distributed around 0 but its variance will be the sum of the variances of each weight, hence 250, leading to a standard deviation of 15.81.
The resulting value will likely be much greater than one, and passing such a large value to the activation function will saturate it. During training, when SGD updates the weights in attempts to influence the activation output, it will only make very small changes in the value of this activation output, barely even incrementally moving it in the right direction. This causes the training to become stuck or very slow.
However, even with non-random initialization (e.g. Xavier initialization) the network would still need to be trained from scratch, and for such a complex model it would take a lot of time.
Furthermore, when dealing with a small dataset, training complex models (with a lot of parameters) from scratch would negatively affect the model's ability to generalize, or it might be not large enough to let the model reach an acceptable performance.
However, generally training a network from scratch generally leads to better results (mAPs) but is obviously much more time and resource consuming.

Fine-tuning

Therefore, more often in practice, it is better to fine-tune existing networks that are trained on a large dataset like the ImageNet (1.2M labeled images) by continuing training it (i.e. running back-propagation) on the smaller dataset we have; provided that the fine-tuning dataset is not drastically different in context to the original dataset, the pre-trained model will already have learned features that are relevant to our own classification problem.

In this case, the weights are initialized by restoring the provided checkpoint of the model.

The common practice is to truncate the last layer (softmax layer) of the pre-trained network and replace it with our new softmax layer that is relevant to our own problem, since the output layer dimension depends on the number of possible classes, then back-propagation is run to fine-tune the pre-trained weights.

It is also a common practice to freeze the weights of the first few layers of the pre-trained network. This is because the first few layers capture universal features like curves and edges that are also relevant to our new problem. We want to keep those weights intact. Instead, we will get the network to focus on learning dataset-specific features in the subsequent layers.

By restoring weights from the pre-trained model's checkpoint the training phase time is considerably reduced.

Of course also fine-tuning could lead to overfitting, especially if the dataset is relatively small, therefore in both cases the training process must be monitored using cross-validation.

In conclusion, it is needed to evaluate the trade-off between the resources and time consumption with the accuracy desired to choose the best approach.

2.3 Object Detection with Convolutional Neural Networks

The general process of object detection is composed of localization of object instances and their classification. Objects, classes and their amount strictly depends on the specific problem and the training dataset.

When using convolutional neural networks, the object detection pipeline is composed of two steps: *feature extraction* and *detection+classification*.

2.3.1 The Backbone Feature Extractor

In this section an overview of the commonly used CNN architectures for feature extraction and for classification will be discussed. Some of those architectures will be deployed and also compared for the specific implementation case.

2.3.1.1 The Backbone Network: VGGNet and ResNet

This section provides a brief overview of two of the most important backbone networks used as feature extractor in the object detection context. These backbone networks act as feature extractor in the object detection pipeline.

Feature extraction involves extracting a higher level of information from raw pixel values that can capture the distinction among the categories involved. This feature extraction is done in an unsupervised manner wherein the classes of the image have nothing to do with information extracted from pixels. The hidden layers are typically activated through a Rectified Linear Unit (ReLU) function, which is $f(x) = max\,(0, x)$. The activation function simply determines the output given the input.

After the feature is extracted, a classification module is trained with the images and their associated labels.

There are multiple versions of the **VGG network** which differ only in the number of layer, however the VGG16 is the most widely used so we will focus on that. The VGG16 network architecture is shown in *figure 2*.

The objective of VGGNet was mainly to reduce the number of parameters in the convolutional layers in order to speed up training time.

The important point to note here is that all the conv kernels are of size 3x3 and maxpool kernels are of size 2x2 with a stride of two.

The idea behind having fixed size kernels is that all the variable size convolutional kernels (used in other networks such as Alexnet) can be replicated by making use of multiple 3x3 kernels as building blocks.

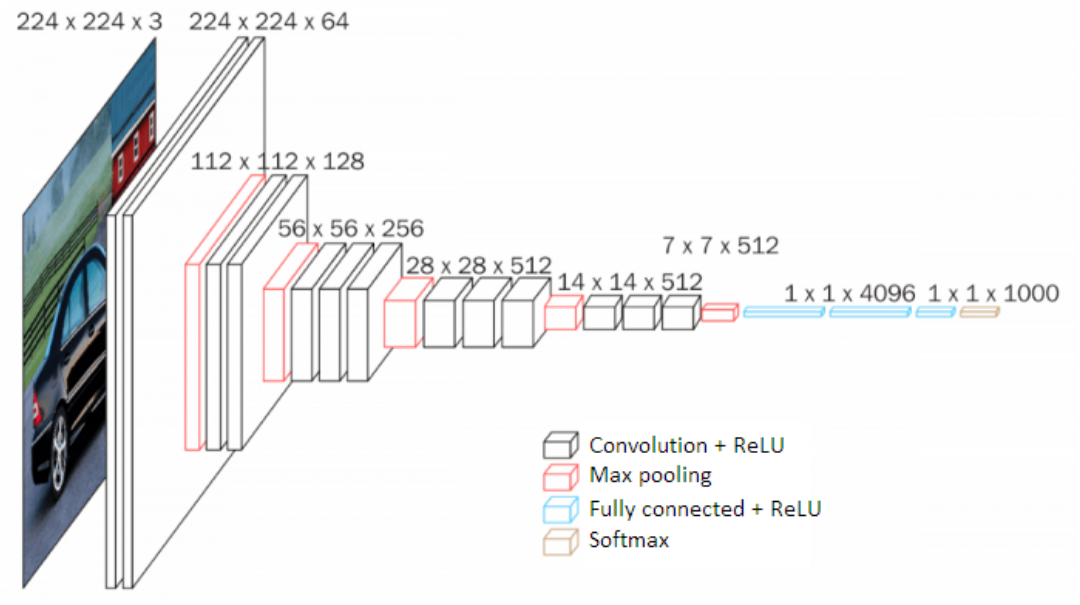


Figure 2 - basic architecture of VGG16 network

VGG16 - Structural Details													
#	Input Image			output			Layer	Stride	Kernel		in	out	Param
1	224	224	3	224	224	64	conv3-64	1	3	3	3	64	1792
2	224	224	64	224	224	64	conv3064	1	3	3	64	64	36928
	224	224	64	112	112	64	maxpool	2	2	2	64	64	0
3	112	112	64	112	112	128	conv3-128	1	3	3	64	128	73856
4	112	112	128	112	112	128	conv3-128	1	3	3	128	128	147584
	112	112	128	56	56	128	maxpool	2	2	2	128	128	65664
5	56	56	128	56	56	256	conv3-256	1	3	3	128	256	295168
6	56	56	256	56	56	256	conv3-256	1	3	3	256	256	590080
7	56	56	256	56	56	256	conv3-256	1	3	3	256	256	590080
	56	56	256	28	28	256	maxpool	2	2	2	256	256	0
8	28	28	256	28	28	512	conv3-512	1	3	3	256	512	1180160
9	28	28	512	28	28	512	conv3-512	1	3	3	512	512	2359808
10	28	28	512	28	28	512	conv3-512	1	3	3	512	512	2359808
	28	28	512	14	14	512	maxpool	2	2	2	512	512	0
11	14	14	512	14	14	512	conv3-512	1	3	3	512	512	2359808
12	14	14	512	14	14	512	conv3-512	1	3	3	512	512	2359808
13	14	14	512	14	14	512	conv3-512	1	3	3	512	512	2359808
	14	14	512	7	7	512	maxpool	2	2	2	512	512	0
14	1	1	25088	1	1	4096	fc		1	1	25088	4096	102764544
15	1	1	4096	1	1	4096	fc		1	1	4096	4096	16781312
16	1	1	4096	1	1	1000	fc		1	1	4096	1000	4097000
Total													**138,423,208**

Table 1 - layers' details of VGG16 network

The replication is in terms of the receptive field covered by the kernels. By replacing convolution of bigger kernels with multiple convolution with a 3x3 kernel reduces (sometimes drastically) the number of parameters to be learned, which turns into faster learning and higher robustness to over-fitting.

Let us consider the following example: we have an input layer of size 5x5x1, implementing a conv layer with a kernel size of 5x5 and stride one will result in an output feature map of 1x1. The same output feature map can be obtained by implementing two 3x3 conv layers with a stride of 1.

By looking at the number of variables needed to be trained, it can be seen that for a 5x5 convolutional layer filter, the number of variables is 25. On the other hand, two convolutional layers of kernel size 3x3 have a total of 3x3x2=18 variables (a reduction of 28%).

In addition, multiple non-linear layers increases the depth of the network which enables it to learn more complex and representative features.

Residual Networks (ResNet) arised by the problem of vanishing gradients when training deeper and deeper neural networks. In fact it would be intuitive to think that adding more layers would increase the accuracy (or at least have the same one) of a network but experiments have shown that the accuracy decreases by adding more layers to the network (overfitting is also another side-effect); this is caused by the vanishing gradients problem: as the neural network becomes deeper and deeper, the derivative when back-propagating to the initial layers becomes almost insignificant in value, this causes the weights in the earlier layer to be almost as frozen. The second problem with training the deeper networks is, performing the optimization on huge parameter space and therefore naively adding the layers leading to higher training error.

Let us see first the intuition behind solving the vanishing gradient problem using skip connections; skip connections allow information to skip layers, so, in the forward pass, information from layer x can directly be fed into layer x+t (i.e. the activations of layer x are added to the activations of layer x+t), for $t \geq 2$, and, during the forward pass, the gradients can also flow unchanged from layer x+t to layer x.

The vanishing gradient problem occurs when the elements of the gradient (the partial derivatives with respect to the parameters of the neural network) become exponentially small, so that the update of the parameters with the gradient becomes almost insignificant and, consequently, the NN learns very slowly or not at all. Given that these partial derivatives are computed with the chain rule, this can easily occur, because you keep on multiplying small numbers. The deeper the NN, the more likely this problem can occur. By allowing information to skip layers, layer x+t receives information from both layer $x+t-1$ and layer x (unchanged, i.e. you do not perform multiplications). For example, to compute the activation

of layer x+t−1 you perform the usual linear combination followed by the application of the non-linear activation function (e.g. ReLU). In this linear combination, you perform multiplications between numbers that could already be quite small, so the results of these multiplications are even smaller numbers. If the activation of layer x+t are even smaller than the activations of layer x+t−1, the addition of the information from layer x will make these activations bigger, thus, to some extent, they will prevent these activations from becoming exponentially small. A similar thing can be said for the back-propagation of the gradient, therefore, skip connections can mitigate the vanishing gradient problem when training deeper NNs.

Residual Networks addresses this network by introducing two types of 'shortcut connections': Identity shortcut and Projection shortcut.

There are multiple variants of ResNets architectures, also here the difference is the number of layers, but the most commonly used are ResNet50 and ResNet101. Basically, deeper ResNets have better accuracy but require more training time and memory resources.

Since the vanishing gradient problem was taken care of, CNN started to get deeper and deeper.

For simplicity, *table 2* presents the 18-layer ResNet structure.

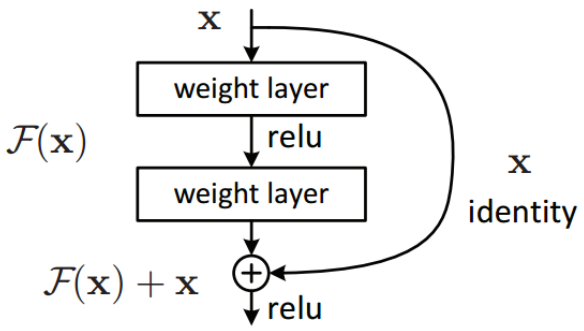
Resnet18 has around 11 million trainable parameters, it consists of convolutional layers with filters of size 3x3 (like VGGNet). Only two pooling layers are used throughout the network: one at the beginning and other at the end of the network. Identity connections are between every two CONV layers.

It can be seen from *table 2* that across the network the spatial dimensions are either kept the same or halved, and the depth is either kept the same or doubled and the product of Width and Depth after each conv layer remains the same i.e. 3584.

ResNet18 - Structural Details														
#	Input Image			output			Layer	Stride	Pad	Kernel		in	out	Param
1	227	227	3	112	112	64	conv1	2	1	7	7	3	64	9472
	112	112	64	56	56	64	maxpool	2	0.5	3	3	64	64	0
2	56	56	64	56	56	64	conv2-1	1	1	3	3	64	64	36928
3	56	56	64	56	56	64	conv2-2	1	1	3	3	64	64	36928
4	56	56	64	56	56	64	conv2-3	1	1	3	3	64	64	36928
5	56	56	64	56	56	64	conv2-4	1	1	3	3	64	64	36928
6	56	56	64	28	28	128	conv3-1	2	0.5	3	3	64	128	73856
7	28	28	128	28	28	128	conv3-2	1	1	3	3	128	128	147584
8	28	28	128	28	28	128	conv3-3	1	1	3	3	128	128	147584
9	28	28	128	28	28	128	conv3-4	1	1	3	3	128	128	147584
10	28	28	128	14	14	256	conv4-1	2	0.5	3	3	128	256	295168
11	14	14	256	14	14	256	conv4-2	1	1	3	3	256	256	590080
12	14	14	256	14	14	256	conv4-3	1	1	3	3	256	256	590080
13	14	14	256	14	14	256	conv4-4	1	1	3	3	256	256	590080
14	14	14	256	7	7	512	conv5-1	2	0.5	3	3	256	512	1180160
15	7	7	512	7	7	512	conv5-2	1	1	3	3	512	512	2359808
16	7	7	512	7	7	512	conv5-3	1	1	3	3	512	512	2359808
17	7	7	512	7	7	512	conv5-4	1	1	3	3	512	512	2359808
	7	7	512	1	1	512	avg pool	7	0	7	7	512	512	0
18	1	1	512	1	1	1000	fc					512	1000	513000
Total														11,511,784

Table 2 - layers' details of ResNet18 network

The basic building block of ResNet is a Residual block that is repeated throughout the network, as schematized below.



Instead of learning the mapping x →F(x), the network learns the mapping x → F(x)+G(x). When the dimension of the input x and output F(x) is the same, the function G(x) is an identity function (G(x) = x) is an identity function and the shortcut connection is called *Identity connection*.

For the case when the dimensions of F(x) differ from x (due to a stride higher than 1 in the convolutional layers in between), the *Projection connection* is implemented and the function G(x) changes the dimensions of the input x to that of the output F(x).

In this case, two kinds of mapping were considered in the original paper:

- ***Non-trainable Mapping (Padding)****:* The input x is simply padded with zeros to make the dimension match to that of F(x)
- ***Trainable Mapping (Conv Layer)****:* 1x1 Conv layer is used to map x to G(x). 1x1 convolutional layers are used to half the spatial dimension and double the depth by using stride length of two and multiple of such filters respectively. The number of 1x1 convolutional layers is equal to the depth of F(x).

As shown in *figure 3*, ResNet (and in general deeper networks, thanks to residual connections) achieves better accuracy than VGGNet and GoogLeNet while being computationally more efficient than VGGNet. The accuracy improvement can be imputed to the greater depth of ResNet (which is possible thanks to residual connection), hence it has more capacity to actually learn better and more sophisticated features.
Furthermore, residual connections "smooths" the trend of the loss function with respect to the network parameters (see *figure 3*), which allows the updates of backpropagation to become more effective.

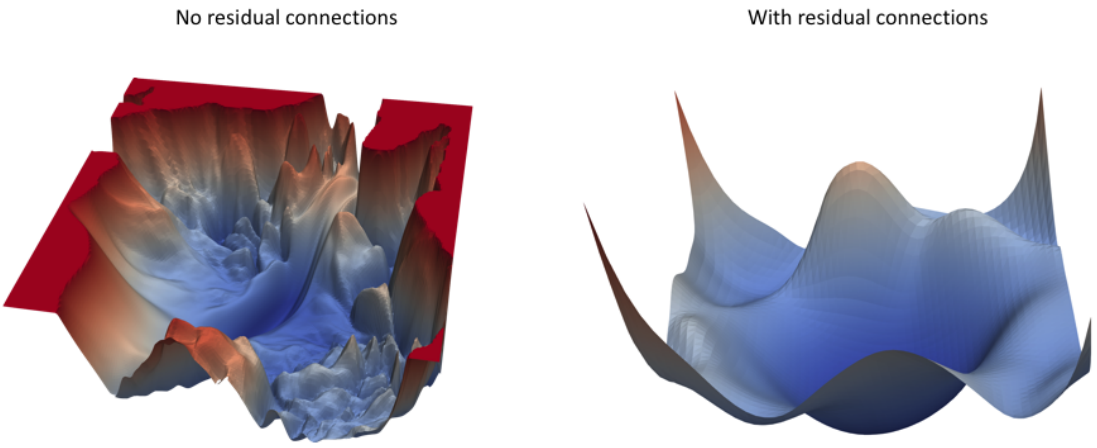


Figure 3 - Left: loss in function of the network weights without the presence of residual connections between layers - Right: loss in function of the network weights with residual connections between layers.

The computational gap derives mainly from the fact that ResNets performs more resizings to the image since the earlier layers. For these reasons a ResNet backbone was chosen for the implementations discussed later.

Comparison					
Network	Year	Salient Feature	top5 accuracy	Parameters	FLOP
AlexNet	2012	Deeper	84.70%	62M	1.5B
VGGNet	2014	Fixed-size kernels	92.30%	138M	19.6B
Inception	2014	Wider - Parallel kernels	93.30%	6.4M	2B
ResNet-152	2015	Shortcut connections	95.51%	60.3M	11B

Table 3 - Four of the most common CNNs backbones sorted with respect to their top-5 accuracy on the Imagenet dataset

2.3.2 Detection and Classification Phase

The second phase of the object detection consists in using the features maps extracted by the backbone network to detect the position of the object within the image and classify it. Again, the second stage is composed of neural networks, however of a simpler structure with respect to the backbone. The detection phase is generally divided into two approaches: two stage detection or single stage detection.

2.3.2.1 Object Detection with Spatial Attention Mechanisms: Two Stage Detection

Different layers of the CNN contain spatial features of objects of different dimensions.
Low layers of the CNN mainly contain edge and corner features. As the layers deepen, they contain higher-dimensional features of objects, such as features of objects parts or even features of the whole objects. As a result, it is important to focus the computational resources on the most informative positions with respect to spatial dimensions. This is exactly the purpose of the Region Proposal Network (RPN) stage in the Faster R-CNN architecture, which will be now further analyzed.

2.3.2.1.1 Faster R-CNN

The Faster R-CNN object detection system is composed of two modules. The first module is a deep fully convolutional network that proposes regions, and the second module is the Fast R-CNN detector whose inputs are the proposed regions.
Using the recently popular terminology of neural networks with ‘attention’ mechanisms, the (first) Region Proposal Network module tells the (second) Fast R-CNN module where to “focus its attention” in the generated feature map. *Figure 4* shows the overall architecture of the system.

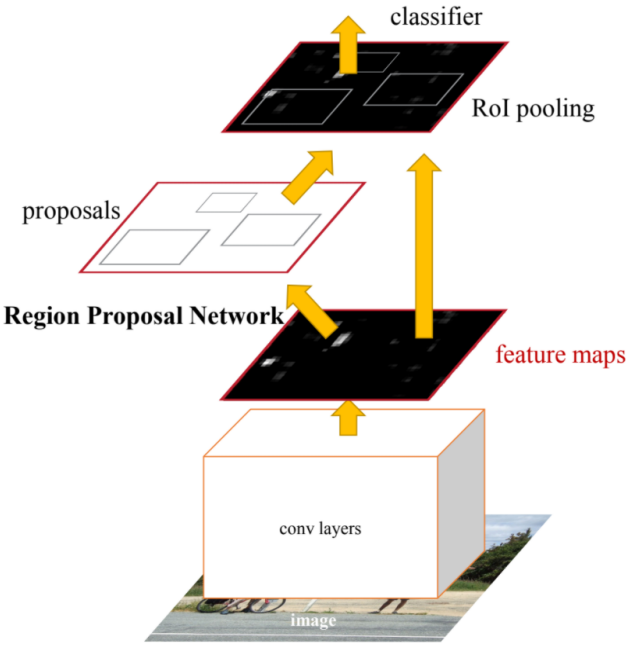


Figure 4 - Basic architecture of the Faster R-CNN network

Region Proposal Network (RPN)

A Region Proposal Network (RPN) takes an image (of any size) as input and outputs a set of rectangular object proposals, each of them with an objectness score. This process is modeled with a fully convolutional network (a network that has only convolutional layers, and no fully connected layers, networks of this kind try to learn representations and make decisions based on local spatial input). Because the ultimate objective is to share computation with a Fast R-CNN object detection network, it is assumed that both networks share a common set of convolutional layers.

To generate region proposals, a small network slides over the convolutional feature map generated by the last shared convolutional layer. This small network takes as input an n × n spatial window of the convolutional feature map (by default n=3). Each sliding window is mapped to a lower-dimensional feature which is then fed into two sibling fully-connected layers: a box-coordinate-regression layer and a box-classification layer; the aim of the first is to refine the bounding box coordinates, whereas the second gives an objectness score.

Since this small network operates as a sliding-window, the fully-connected layers are shared across all spatial locations of the feature map. This architecture is naturally implemented with

a 3×3 convolutional layer followed by two sibling 1 × 1 convolutional layers (for regression and classification, respectively).

At each feature map location, the network simultaneously predicts multiple region proposals, where the number of maximum possible proposals for each location is denoted as k. So the regression layer has 4k outputs encoding the coordinates of k boxes, and the cls layer outputs 2k scores that estimate probability of containing an object or not for each proposal.

The k proposals are parameterized relative to k reference boxes, that are called **anchors**.

An anchor is centered at the sliding window, and is associated with a scale and aspect ratio, consequently $k = num\ of\ scales \times num\ of\ aspect\ ra$tios and for a convolutional feature map of a size W×H, there are W×H×k anchors in total. For each point in the output feature map, the network has to learn whether an object is present in the input image at its corresponding location and estimate its size: two adjacent points in the feature map correspond to two point distant r pixels in the input image (r is called subsampling ratio and its another hyperparameter of the network); therefore the final anchors reference the original image.

These scales and aspect ratios are very important hyperparameters of this architecture to allow the model to be able to detect objects in a wide range of scales and aspect ratios, since the implemented method classifies and regresses bounding boxes with reference to these anchor boxes of multiple scales and aspect ratios.

The performed regression can be thought of as bounding-box regression from an anchor box to a nearby ground-truth box.

In this formulation, the features used for regression are of the same spatial size (3 × 3) on the feature maps. To account for varying sizes, a set of k bounding-box regressors are learned. Each regressor is responsible for one scale and one aspect ratio, and the k regressors do not share weights. As such, it is still possible to predict boxes of various sizes albeit the features are of a fixed size/scale, thanks to the design of anchors.

This algorithm also allows predictions that are larger than the underlying receptive field; the extent of an object can be roughly inferred even if only the middle of the object is visible.

Figure 5 shows the anchors generated in a single position of the sliding window on the feature map.

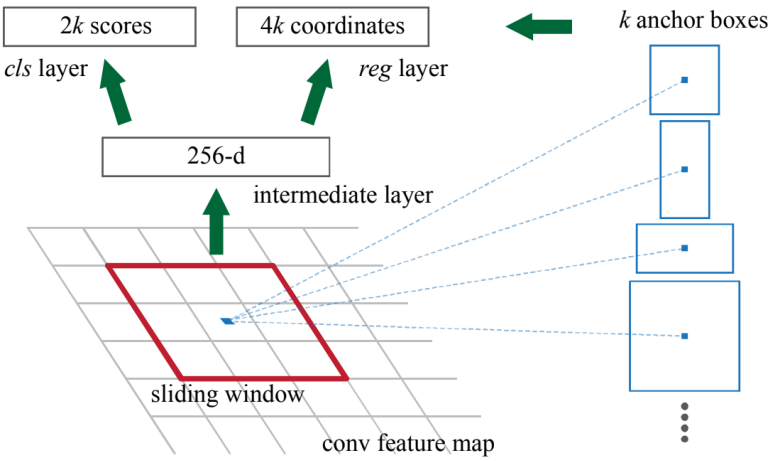


Figure 5 - Anchors generated by the RPN on the feature map produced by the feature extractor

This approach allows the network to have the translation-invariant property, both in terms of the anchors and the functions that compute proposals relative to the anchors.

The computational efficiency is also improved with respect to other methods, because this method only relies on images and feature maps of a single scale, and uses filters (sliding windows on the feature map) of a single size (3x3).

Training

For training the RPN, a binary class label (of being an object or not) is assigned to each anchor. A positive label is then assigned to an anchor if it has an IoU overlap higher than 0.7 with a ground-truth box.

A negative label is assigned to an anchor if its IoU ratio is lower than 0.3 for all ground-truth boxes (anchors that are neither positive nor negative do not contribute to the training objective).

$$L(\{p_i\},\{t_i\}) = \frac{1}{N_{cls}}\sum_i L_{cls}(p_i, p_i^*) + \lambda\frac{1}{N_{reg}}\sum_i p_i^* L_{reg}(t_i, t_i^*)$$

Where i is the index of an anchor in a mini-batch and p_i is the predicted probability of anchor i being an object (output of the classification layer). The ground-truth label p_i* is 1 if the anchor is positive, and is 0 if the anchor is negative. t_i is a vector representing the 4

parameterized coordinates of the predicted bounding box (output of the regression layer), and t_i* is that of the ground-truth box associated with a positive anchor.

The classification loss Lcls is the binary cross-entropy loss over two classes for the RPN (object vs. not object) and over all classes for the detector. For the regression loss, $L_{reg}(t_i, t_i^*) = R(t_i - t_i^*)$ is used where R is the robust loss function used in Fast R-CNN. The regression loss is used only for positive anchors ($p_i^* = 1$). N_{cls} and N_{reg} are normalization parameters: mini-batch size and number of anchors respectively. λ is a balancing parameter to equally weight the two losses.

For training, 256 anchors are randomly sampled in an image to compute the loss function of a mini-batch, where the sampled positive and negative anchors have a ratio of up to 1:1 (using all the anchors would bias towards negative anchor samples).

During training, the two networks need to share convolutional layers, rather than learning two separate networks.

4-step alternating training

The *paper*[5] proposes a 4-step training algorithm to learn shared features via alternating optimization:

- ***First step***: the RPN is fine-tuned independently for the region proposal task using by default back-propagation with stochastic gradient descent (the backbone CNN is initialized with an ImageNet-pre-trained model and fine-tuned)
- ***Second step***: the Fast R-CNN detection network is fine-tuned independently for classification task using the proposals generated by the RPN (also initialized by the ImageNet-pre-trained model)
- ***Third step***: the detector network's weights are used to initialize the RPN, which is then fine-tuned again. This time the shared convolutional layers are freezed and only the layers unique to RPN are fine-tuned, in order to make the networks share the convolutional layers. At the end of this step the final RPN is obtained.
- ***Fourth step***: keeping the shared convolutional layers freezed, the layers of Fast R-CNN are again fine-tuned using the RPN.

At the end of this process, both networks share the same convolutional layers and form a unified network.

During training, the anchor boxes that cross image boundaries need to be handled with care:If the boundary-crossing anchors are not ignored in training they introduce a large and difficult to correct error term in the objective function, and training does not converge.

The benefits of sharing convolutional layers between is both in terms of computational efficiency (by sharing layer it is obvious that less operations performed) but also in terms of accuracy (see *table 4* for a simple comparison using VGG16 as backbone and VOC 2007 trainval as training data); It is observed that this is because in the third step when the detector-tuned features are used to fine-tune the RPN, the proposal quality is improved.

Method	# Proposals	mAP(%)
RPN+VGG, unshared	300	68.5
RPN+VGG, shared	300	69.9

Table 4 - effect of sharing convolutional layers between the RPN and Fast R-CNN detection network. This is obtained by simply stopping the 4-step training process after completing the second step

Backbone

Some experiments computed on the COCO (Common Objects in Context) validation set showed that an accuracy improvement is achieved when replacing the VGG-16 backbone network with the 101-layer ResNet (ResNet-101), the Faster R-CNN system increases the mAP from 41.5%/21.2% to 48.4%/27.2% for the RPN and detector network respectively. This results confirms the superiority of ResNet, due to the aspects also discussed previously. Consequently, experiments described later will use the Faster R-CNN architecture with ResNet101 as the backbone feature extractor, for its good trade-off between accuracy and complexity.

Fast R-CNN as a Detector for Faster R-CNN

The Fast R-CNN detector also consists of a CNN backbone (shared), a ROI pooling layer and fully connected layers followed by two sibling branches for classification and bounding box regression. As stated before, besides test time efficiency, another key reason using an RPN as a proposal generator is beneficial is the advantages of weight sharing between the RPN backbone and the Fast R-CNN detector backbone.

The input image is first passed through the backbone CNN to get the feature map, successively, the bounding box proposals from the RPN are used to pool features from the backbone feature map. This is done by the ROI pooling layer, which essentially works by taking the region corresponding to a proposal from the backbone feature map; Dividing this region into a fixed number of sub-windows and then performing max-pooling over these sub-windows to give a fixed size output (for this reason the Faster R-CNN can handle input images of different resolutions). The RoI pooling outputs go through two fully connected layers, the features are then fed into the sibling classification and regression branches (which are different from those of the RPN). The classification layer has C units for each of the classes in the detection task whose features are passed through a softmax layer to get the classification scores for each class. The regression layer coefficients are used to improve the predicted bounding boxes, and for each class there is a specific regression layer. That is, all the classes have individual regressors with 4 parameters each corresponding to C*4 output units in the regression layer.

The overall final architecture (Region Proposal Network + Fast R-CNN network) is schematized in *figure 6.*

2.3.2.1.2 Feature Pyramid Networks (FPNs)

Recognizing objects at vastly different scales is a fundamental challenge in computer vision. Feature pyramids built upon image pyramids (featured image pyramids) form the basis of a standard solution (see *figure 7*). These pyramids are scale-invariant since an object's scale change is offset by shifting its level in the pyramid.

This architecture enables a model to detect objects across a large range of scales by scanning the model over both positions and pyramid levels.

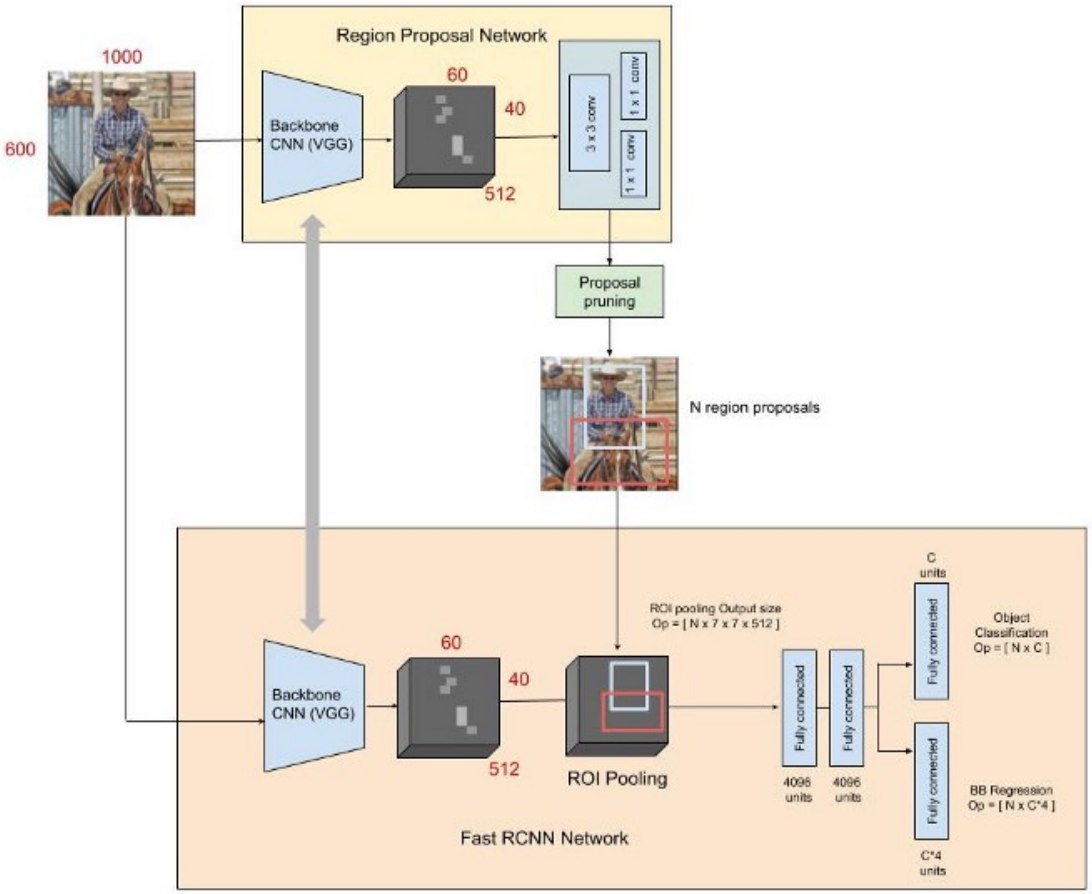


Figure 6 - Faster R-CNN overall architecture

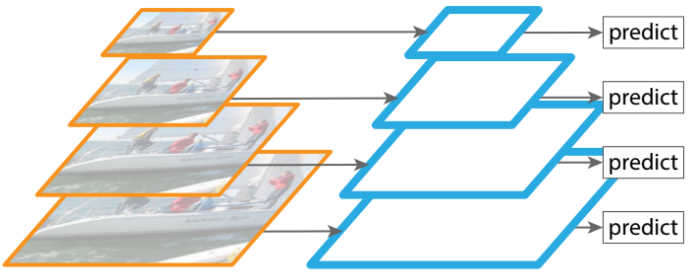


Figure 7 - Feature pyramids of a single image. Each level of the pyramid corresponds to a different image scale

The principal advantage of featurizing each level of an image pyramid is that it produces a multi-scale feature representation in which all levels are semantically strong, including the

high-resolution levels (which usually contain very general features). However, this approach has high limitations due to the low speed performance and huge memory usage.

Recent and more accurate detection methods like Fast R-CNN and Faster R-CNN advocate using features computed from a single scale, because it offers a good trade-off between accuracy and speed. Multi-scale detection, however, still performs better, *especially for small objects, which is the main purpose of the FPN architecture*.

This approach takes a single-scale image of an arbitrary size as input, and outputs proportionally sized feature maps at multiple levels, in a fully convolutional fashion. This process is independent of the backbone convolutional architectures, in this case ResNet is considered as the backbone network.

All levels of the pyramid use shared classifiers/regressors as in a traditional featurized image pyramid, for this reason the feature dimension needs to be fixed.

Bottom-up pathway

The bottom-up pathway is the feedforward computation of the backbone CNN, which computes a feature hierarchy consisting of feature maps at several scales with a scaling step of 2 (layers keeping the feature map size are considered of being in the same pyramid level, but the feature map of the last layer is used).

The outputs of these last residual blocks are denoted as{C2, C3, C4, C5} for conv2, conv3, conv4, and conv5 outputs, and note that they have strides of {4, 8, 16, 32} pixels with respect to the input image.

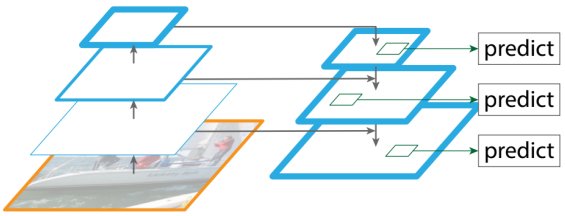


Figure 8 - Feature pyramid generated by a single image scale. On the left there is the bottom-up pathway, on the right the top-down pathway with lateral connections on the same level coming from the first one. note: predictions are made independently at each level

Top-down pathway and lateral connections

The top-down pathway tries to recreate higher resolution features by upsampling spatially coarser, but semantically stronger, feature maps from higher pyramid levels. These features are then enhanced with features from the bottom-up pathway via lateral connections. Each lateral connection merges feature maps of the same spatial size from the bottom-up pathway and the top-down pathway. The bottom-up feature map is of lower-level semantics, but its activations are more accurately localized as it was subsampled fewer times, so lateral connections are used to refine the localization accuracy in the top-down pathway.

Figure 9 shows the building block that constructs the top-down feature maps.

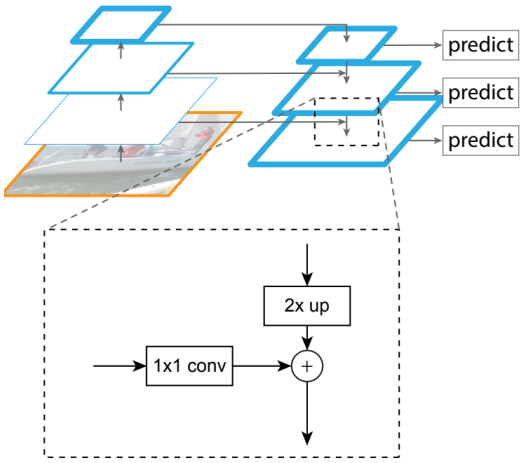


Figure 9 - A building block illustrating the lateral connection and the top-down pathway, merged by addition.

To start the iteration, a simple 1×1 convolutional layer is attached to C5 to produce the coarsest resolution map of the top-down pathway.

The coarser-resolution feature map is upsampled the spatial resolution by a factor of 2 (using nearest neighbor upsampling).

The upsampled map is then merged with the corresponding bottom-up map (which undergoes a 1×1 convolutional layer to reduce channel dimensions) by element-wise addition. Finally, a 3×3 convolution is appended to each merged map to generate the final feature map, which is to reduce the aliasing effect of upsampling.

This process is iterated until the finest resolution map is generated.

This final set of feature maps is called {P2, P3, P4, P5}, corresponding to {C2, C3, C4, C5} that are respectively of the same spatial sizes.

2.3.2.1.3 Faster R-CNN with FPN

As discussed previously, when it is necessary to detect small-scale objects, feature pyramid network architecture is almost indispensable due to its robustness to scale variance.

Feature Pyramid Network (FPN) for Region Proposal Networks (RPN)

The RPN is adapted to the FPN architecture by replacing the single-scale feature map with our FPN. A head with the same design (3×3 convolution and two sibling 1×1 convolutions) is applied to each level on the feature pyramid. Since the head slides densely over all locations in all pyramid levels, it is not necessary to have multi-scale anchors on a specific level. Instead, a single anchor of a different scale is assigned to each level. By default, the anchors are defined so to have areas of {32, 64, 128, 256, 512} pixels on {P2, P3, P4, P5, P6} respectively, so that the low resolution feature maps in the higher pyramid level is used to detect large objects, and the detecting scale lowers by going down in the pyramid to the high resolution feature maps. At each level the anchors have different aspect ratios (three aspect ratios are defined by default) leading to having 15 anchors over distributed in the pyramid.

Scales of ground-truth boxes are not explicitly used to assign them to the levels of the pyramid; instead, ground-truth boxes are associated with anchors, which have been assigned to pyramid levels.

It is worth pointing out that the parameters of the heads are shared across all feature pyramid levels, and the good performance of sharing parameters indicates that all levels of the pyramid share similar semantic levels.

Feature Pyramid Network (FPN) for Fast R-CNN Detector

As seen before, Fast R-CNN is a region-based object detector in which Region-of-Interest (RoI) pooling is used to extract features. Fast R-CNN is most commonly performed on a single-scale feature map. To use it with our FPN, RoIs of different scales must be assigned to the pyramid levels.

Formally, a RoI of width w and height h (on the input image to the network) is assigned to the level Pk of our feature pyramid by the following formula:

$$k = \lfloor k_0 + \log_2(\sqrt{wh}/224) \rfloor$$

Here 224 is the canonical ImageNet/COCO pre-training size. Intuitively, this equation means that if the RoI's scale becomes smaller (e.g. 1/2 of 224), it should be mapped into a finer-resolution level (e.g. k = 3). Predictor heads are then attached to all RoIs of all levels (all the heads share parameters, regardless of their levels). Specifically, a RoI pooling is applied to extract 7×7 features, and two hidden 1024-d fully-connected layers (each followed by ReLU) are attached before the final classification and bounding box regression layers.

Comparison between Faster R-CNN with and without FPN

The input image is resized such that its shorter side has 800 pixels. Synchronized SGD is used to train the model on 8 GPUs. Each mini-batch involves 2 images per GPU and 512 RoIs per image. A weight decay of 0.0001 and a momentum of 0.9 are used. The learning rate is 0.02 for the first 60k mini-batches and 0.002 for the next 20k. 2000 RoIs are used per image for training and 1000 for testing. The Fast R-CNN with FPN was trained for about 10 hours on the COCO dataset.

Table 5 shows a simple comparison of the effect of the FPN integration in the Faster R-CNN architecture. The metrics compared in the table are described in *section 2.3.3* and *section 3.3.7.2*.

Faster R-CNN	**Feature**	**Head**	**AP@0.5**	**AP**	**AP_S**	**AP_M**	**AP_L**
(a)Standard baseline	C_4	*2fc*	53.1	31.6	13.2	35.6	47.1
(b)Baseline with FPN	P_K	*2fc*	56.9	33.9	17.8	37.7	45.8

Table 5 - The first row presents the results based on the default implementation of Faster R-CNN, the last row shows based on the custom implementation with FPN integration

As can be seen the accuracy is considerably improved, especially on small scales.

Running time

The FPN-based Faster R-CNN system has inference time of 0.148 seconds per image on a single NVIDIA M40 GPU for ResNet-50, and 0.172 seconds for ResNet-101.6 As a comparison, the single-scale ResNet-50 baseline in *Table 5*(a) runs at 0.32 seconds.

The new method (*Table 5* (b)) introduces small extra cost by the extra layers in the FPN, but has a lighter weight head. Overall the FPN system is faster than the ResNet-based Faster R-CNN counterpart, which reinforces the supremacy of the FPN-integrated version.

2.3.2.2 Single Shot Object Detection: Single Stage Detection

As the name already suggests, the process of single shot detectors has only one stage to detect objects; as such, they are generally faster than two-stage detectors, albeit less accurate. Here a brief description of the Single Shot Detector (SSD) is provided, since it was used in the experiments as well.

2.3.2.2.1 Single Shot Multibox Detector (SSD)

The SSD architecture has higher inference speed performance than the Faster R-CNN, since it does not have the region proposal network, but only a single detection stage. The speed improvement is paid in accuracy; to recover the drop in accuracy, the SSD applies a few improvements including multi-scale features and default boxes. These aspects allow the SSD architecture to achieve good accuracy even with low resolution input images, which in turn further improves speed.

The SSD is shown in *figure 10* and is composed of two simple stages: extract the feature maps from the input and apply convolution filters to detect objects.

The backbone feature extractor is again generally VGG16 or ResNet, which generates multiple feature maps.

For each cell in the feature maps, the network generates a fixed number of predictions (by default four or six depending on the layer) using default box shapes.

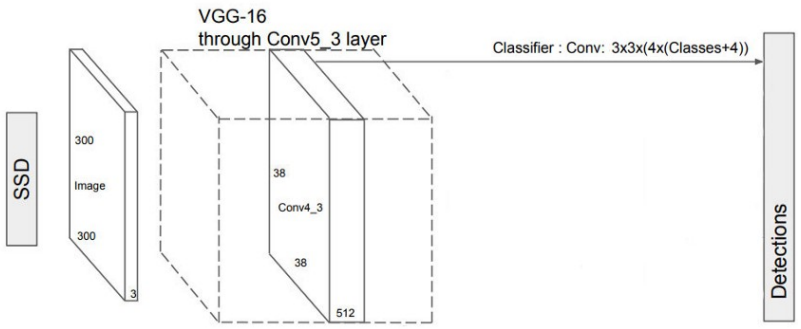


Figure 10 - Basic architecture of the Single Shot Detector network

Thereafter, for each location a bounding box regression and a classification score for each possible class is generated (an additional class is inserted as "no object"), regardless of the feature map depth. *Figure 11* shows a scheme of the detection process.

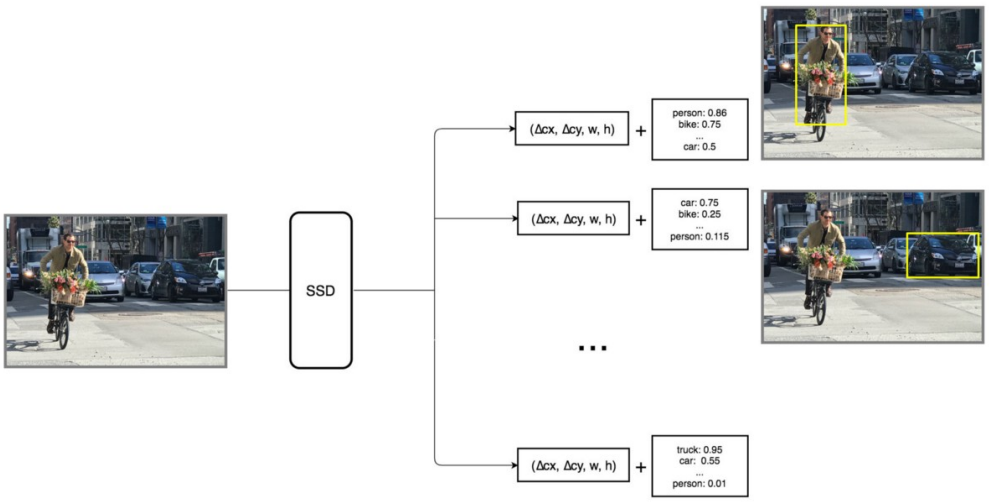


Figure 11 - Detection process based on the generation of default bounding boxes at each location of the feature map

Making multiple predictions containing boundary boxes and confidence scores is called *multibox*, hence the name of the network.

So, SSD computes both the location and class scores using small convolution filters: after extracting the feature maps, SSD applies 3 × 3 convolution filters for each cell to make predictions. Each filter outputs a prediction of N channels, where $N = number\ of\ classes\ +\ four\ bounding\ box\ coordinates$, as shown in *figure 11*.

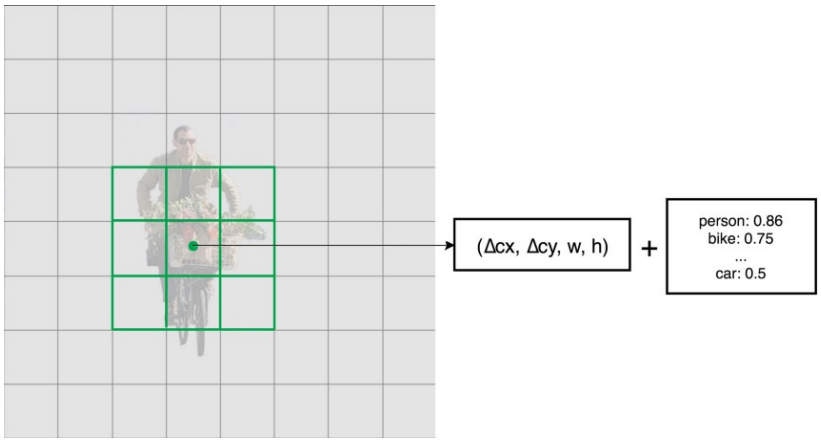


Figure 12 - Predictions generated at each location of the feature map

Δcx and *Δcy* are the offset with respect to the default bounding box of width *w* and height *h*.

The SSD generates many feature maps of different resolution to detect objects of different scales, in practice lower resolution maps are used to detect larger scale objects and vice versa (see *figure 13*). Due to this aspect of its architecture, SSD models have typically a lower performance on the detection of small-scale objects.

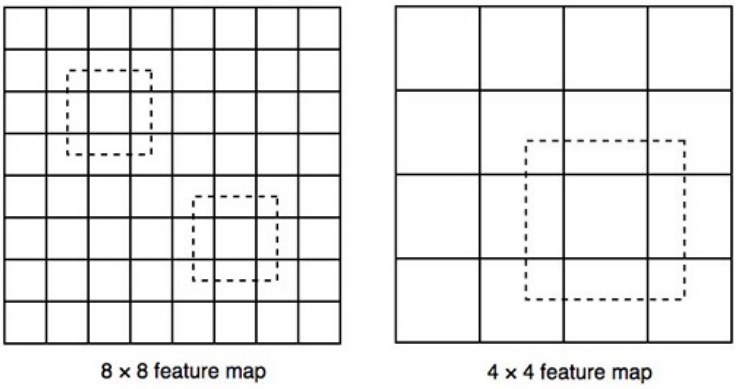


Figure 13 - Default bounding boxes on two feature maps at different resolutions. Lower resolution maps are used to detect larger scale objects and vice versa

The overall architecture of the SSD network is shown in *figure 14*.

In the default implementation, 8732 predictions are generated, using six layers. The first layer for object detection conv4_3 has a spatial dimension of 38 × 38, a quite large reduction from the input image.

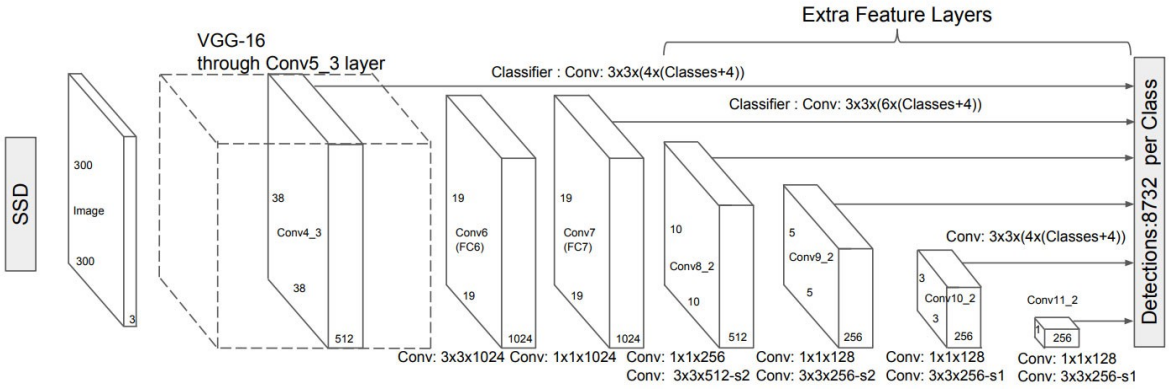


Figure 14 - Single Shot Detector network overall architecture

In addition, since the very first layers have typically weak semantic features, they are not used for detection, but ideally those feature maps would be useful for detecting low scale objects since they have a higher resolution; for these reasons, SSD usually performs worse on small objects compared with other detection methods. It was nevertheless taken into consideration

with the integration of the FPN network, also to have a useful comparison with the Faster R-CNN.

The boundary boxes are generated using default boxes shapes, which have to be chosen manually, ideally by clustering the ground truth boundary boxes and taking the centroid of each cluster as the default bounding box, so as to have custom bounding boxes adapted for the specific problem.

For each default box, a prediction is generated: class scores and bounding box regression, where the latter are relative to the default box coordinates. Note that each layer has its own set of default boxes so to have the possibility to customize the object detection at different resolutions.

During training, predictions are classified as positive or negative matches, of which only positive ones are used to compute the localization cost. Precisely, if the default boundary box has an IoU higher than 0.5 with a ground truth box it is classified as a positive match. Afterwards, the predicted boundary boxes corresponding to the positive matches are used to calculate the localization loss. This matching strategy encourages each prediction to predict shapes closer to the corresponding default box.

Localization Loss

The localization loss between the predicted box *l* and the groundtruth box *g* is defined as the smooth L1 loss:

$$L_{loc}(x, l, g) = \sum_{i \in Pos}^{N} \sum_{m \in \{cx, cy, w, h\}} x_{ij}^{k} \text{smooth}_{\text{L1}}(l_i^m - \hat{g}_j^m)$$

$$\hat{g}_j^{cx} = (g_j^{cx} - d_i^{cx})/d_i^w \qquad \hat{g}_j^{cy} = (g_j^{cy} - d_i^{cy})/d_i^h$$

$$\hat{g}_j^w = \log\left(\frac{g_j^w}{d_i^w}\right) \qquad \hat{g}_j^h = \log\left(\frac{g_j^h}{d_i^h}\right)$$

$$x_{ij}^p = \begin{cases} 1 & \text{if } IoU > 0.5 \text{ between default box } i \text{ and ground true box } j \text{ on class } p \\ 0 & \text{otherwise} \end{cases}$$

with *cx* and *cy* as the offset to the default bounding box *d* of width *w* and height *h*.

Classification Loss

For every positive match prediction, the loss regarding the confidence score of the corresponding class is computed. For negative match prediction, the loss regarding the confidence score of the class "0" is calculated, where "0" is the "no object" class.

The loss is computed as the *softmax cross-entropy* loss over all the *c* classes:

$$L_{conf}(x,c) = -\sum_{i \in Pos}^{N} x_{ij}^{p} log(\hat{c}_i^p) - \sum_{i \in Neg} log(\hat{c}_i^0) \quad \textbf{where} \quad \hat{c}_i^p = \frac{\exp(c_i^p)}{\sum_p \exp(c_i^p)}$$

Where *N* is the number of matched default boxes.

Eventually, the total loss is computed as:

$$L(x,c,l,g) = \frac{1}{N}(L_{conf}(x,c) + \alpha L_{loc}(x,l,g))$$

Where α is the weight of the localization loss.

Of course the number of positive matches will be much lower than the negative ones, and this imbalance is going to disturb the training process. Therefore, the negative matches are sorted by their confidence loss, and only few of the top negatives are taken, making sure that the ratio between the negatives and positives doesn't exceed 3:1; doing so leads to faster and more stable training.

Such prediction techniques lead to having many negative (with no object) and duplicate predictions.

At inference time, negative predictions are removed by filtering out those with confidence scores lower than 0.01 (of course not considering class "0").

To remove duplicate predictions, SSD implements a non-maxima suppression; predictions are sorted by their confidence scores, for each class the one with highest confidence score is considered and all the other predicted bounding boxes for the same class that have an IoU higher than 0.45 with it are removed. At the end, only the top 200 predictions are kept for each image.

2.3.3 Object Detection with Convolutional Neural Networks: Evaluation Metrics

To evaluate the performance of an object detection algorithm, two main metrics are used: *Precision* and *Recall,* which are defined by the following formulas.

$$Precision = \frac{True\ positives}{True\ positives + False\ positives} \qquad Recall = \frac{True\ positives}{True\ positives + False\ negatives}$$

The *precision* is an indicator of the accuracy of the detected object: an high precision means that an object that is being detected by the network is indeed a real object, therefore the higher is the precision, the less are the false positives detected by the network.

The *recall* value indicates the percentage of real objects that the network is detecting, therefore an high recall indicates that the network is able to detect all the objects within the image that is given in input to it. Consequently, it is desirable to have a CNN model that has both high precision and recall scores, but in reality an increase in recall translates to a decrease in precision and vice versa.

By setting the threshold of the confidence score at different levels, different pairs of precision and recall values are obtained. With the recall on the x-axis and the precision on the y-axis, we can draw a precision-recall curve, which indicates the association between the two metrics. The precision-recall curve shows the tradeoff between precision and recall for different thresholds. A high area under the curve represents both high recall and high precision, where high precision relates to a low false positive rate, and high recall relates to a low false negative rate. High scores for both show that the classifier is returning accurate results (high precision), as well as returning a majority of all positive results (high recall). This curve helps to select the best threshold to maximize both metrics. In multi-class detectors there is a different precision-recall curve for each class. *Figure 15* shows a typical precision-recall curve.

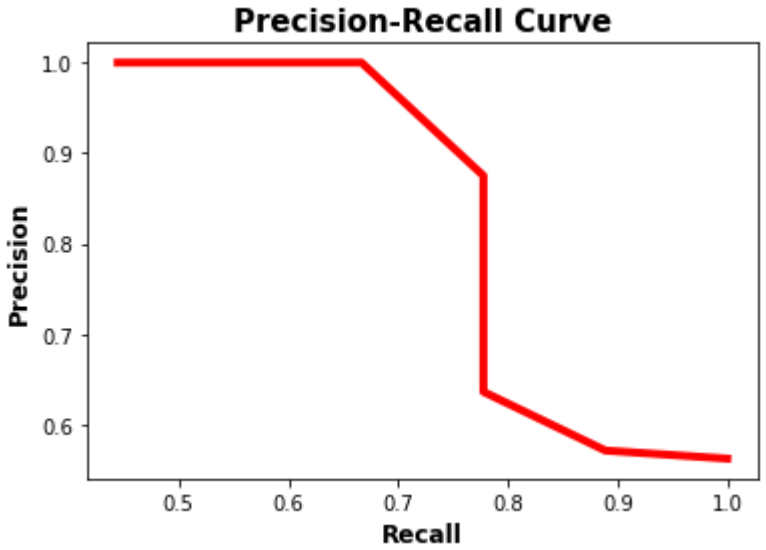


Figure 15 - Typical precision-recall curve

Generally, as the recall increases, the precision decreases. The reason is that when the number of positive samples increases (high recall), the accuracy of classifying each sample correctly decreases (low precision).

The **average precision (AP)** is a way to summarize the precision-recall curve into a single value representing the average of all precisions. It can be seen as the weighted sum of precisions at each threshold where the weight is the increase in recall, and is calculated as follows:

$$AP = \sum_{k=0}^{k=n-1} (recalls(k) - recalls(k+1)) * precisions(k)$$

where n = number of thresholds

Finally, the *accuracy* is the percentage of correctly predicted examples out of all predictions, formalized with the formula:

$$Accuracy = \frac{True\ positives + True\ negatives}{True\ positives + False\ positives + True\ negatives + False\ negatives}$$

In the context of object detection, there exist many families of accuracy metrics, that are all composed of different combinations of the *precision* and *recall* indicators. Those metrics

typically differentiate the scores by object dimensions, classes and number of proposals in the first stage of detection.

2.4 Datasets

2.4.1 Mapillary Traffic Sign Dataset

The main dataset used to train the network is the Mapillary Traffic Sign Dataset. This dataset was chosen due to its characteristic of including examples with an extremely large range of properties, which stimulates the network to learn features that makes it able to properly generalize on unseen data.

The datasets includes 52'453 images, that are divided into train, validation and test set, with 36'589, 5'320 and 10'544 images and 180'287 and 26'101 annotations respectively (there are no annotations in the test set).

The dataset taxonomy comprises a total of 314 classes, whose distribution is depicted in *figure 16*.

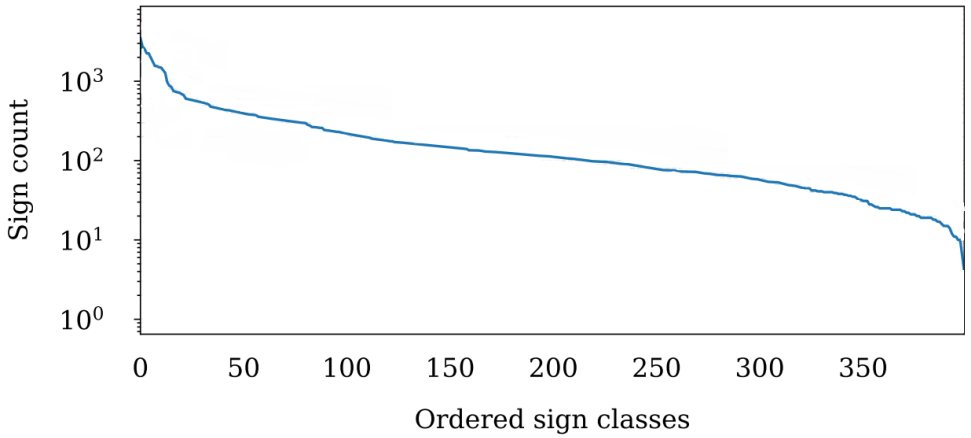


Figure 16 - Traffic signs classes distribution within the Mapillary dataset

Each annotation is also endowed of informative properties that can be used to improve the quality of the training examples (e.g. by filtering examples with bad properties): *occluded* if the sign is partly occluded; *ambiguous* if the sign is not classifiable at all (e.g. too small, bad quality, heavy occlusion etc.); *dummy* if it looks like a sign but isn't (e.g. car stickers, reflections, etc.); *out-of-frame* if the sign is cut off by the image border; *included* if the sign is part of another bigger sign; and exterior if the sign includes other signs.

To encourage the generalization ability of the model, the images are taken from different part of the world, since traffic signs' appearances change across countries. *Figure 17* shows the distribution of the images among countries.

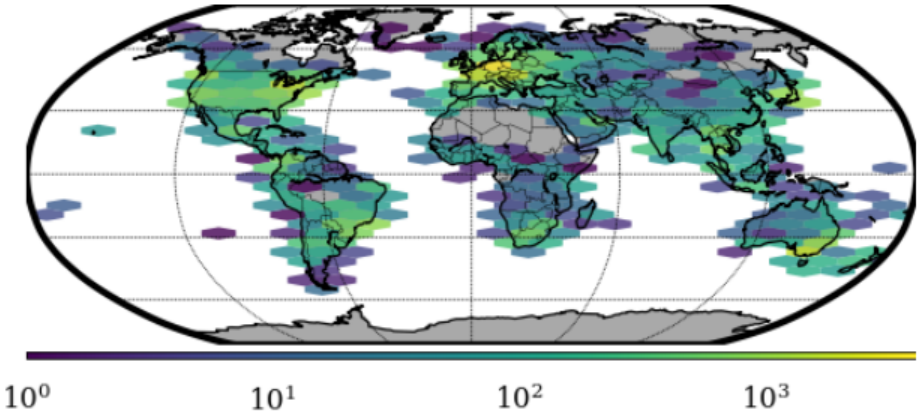


Figure 17 - Amount of images of the Mapillary dataset coming from each country

A detailed analysis of this dataset was carried out, which will be described in the next chapter. Images within the dataset are taken with different devices, therefore have different resolutions and lens distortions. *Figure 18* shows the distribution of image resolutions in the entire dataset.

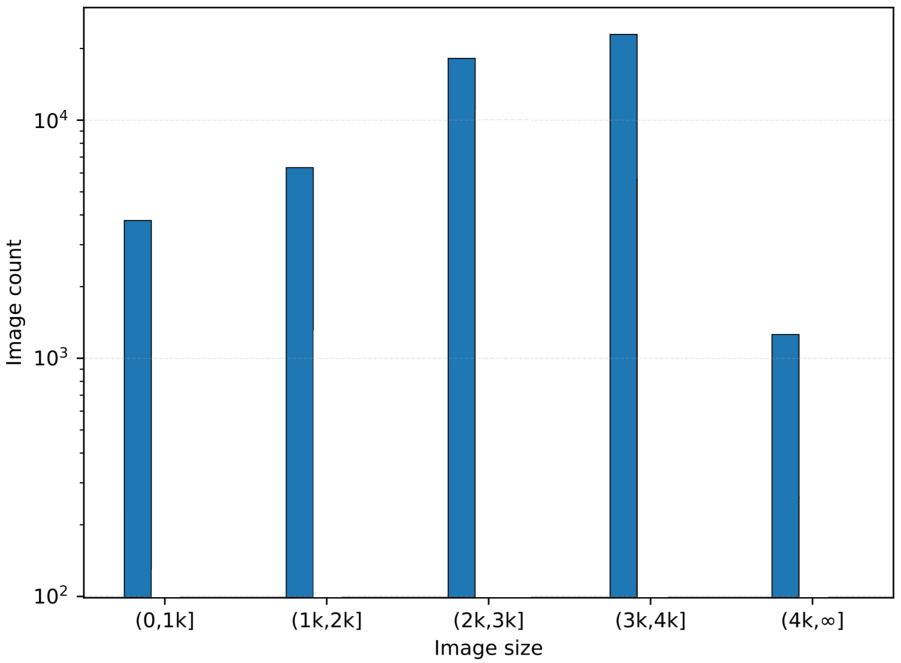


Figure 18 - Distribution of the image resolutions of the Mapillary dataset

As shown in *figure 19*, traffic signs annotated in the dataset cover a wide range of pixel-areas, which is also due to the different resolutions of images.

All these properties make the Mapillary dataset a valuable source of information and features that can be used to train a neural network.

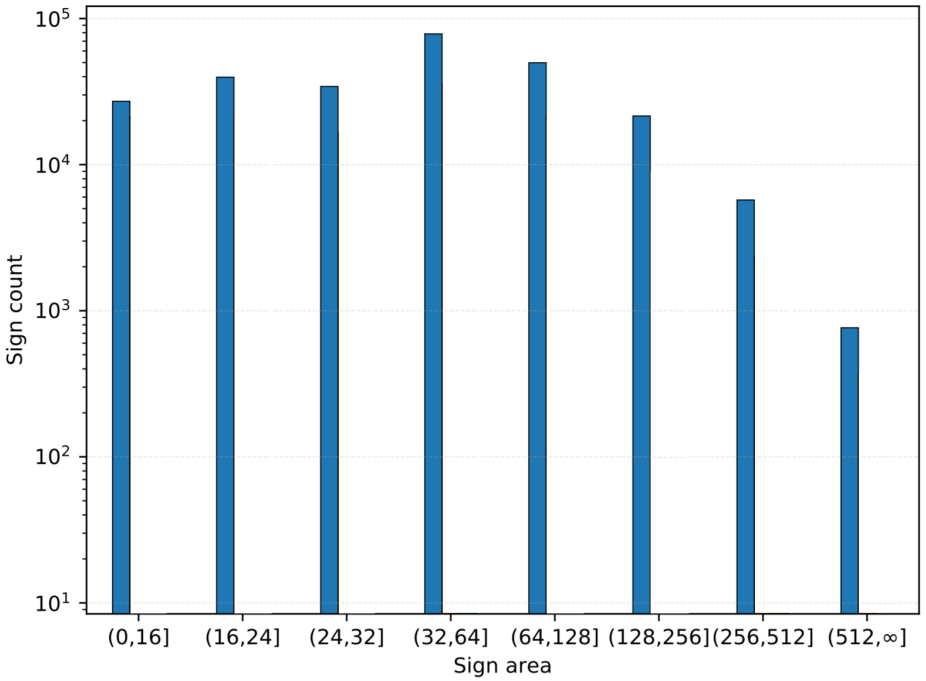


Figure 19 - Distribution of the traffic sign pixel areas in the Mapillary dataset

2.4.2 Slowenian DFG Traffic Sign Dataset (DFGTSD)

To test the efficacy of the domain adaptation processes performed, the Slovenian traffic signs dataset is used (DFGTSD or simply DFG).

Such a dataset does not have all the useful properties that the Mapillary dataset has; for example, images all have the same resolution (1920 x 1080) and are all taken in Slovenia. In addition, the dataset is quite smaller compared to Mapillary, consequently has also less classes, with a total of 200 classes taxonomy.

The dataset includes a total of 16’264 images, of which 5’254 in the training split, 1’703 in the validation split and 9’307 in the test set. The training split has 9’815 annotations, while the validation 4’393.

The class distribution (*figure 20*) is less uniform with respect to the Mapillary dataset.

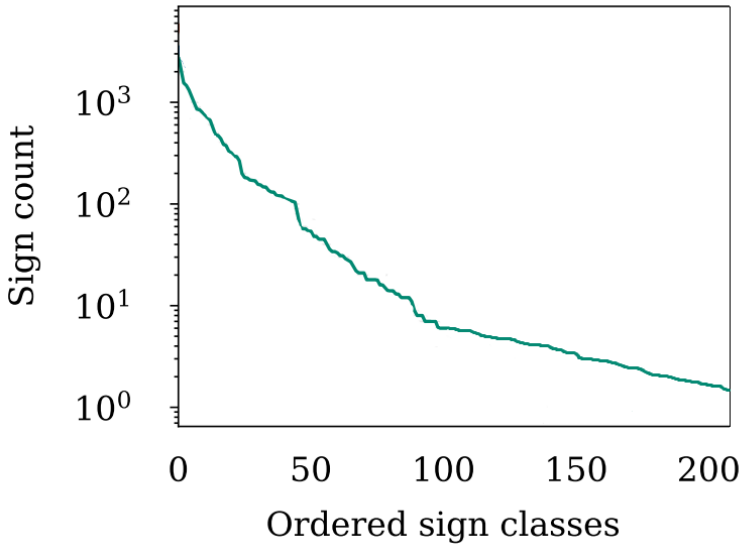


Figure 20 - Traffic signs classes distribution of the Slovenian traffic sign dataset

Signs's area distribution is shown in *figure 21*, it can be seen how the traffic signs are generally larger in this dataset, which is also caused by having images of lower resolutions.

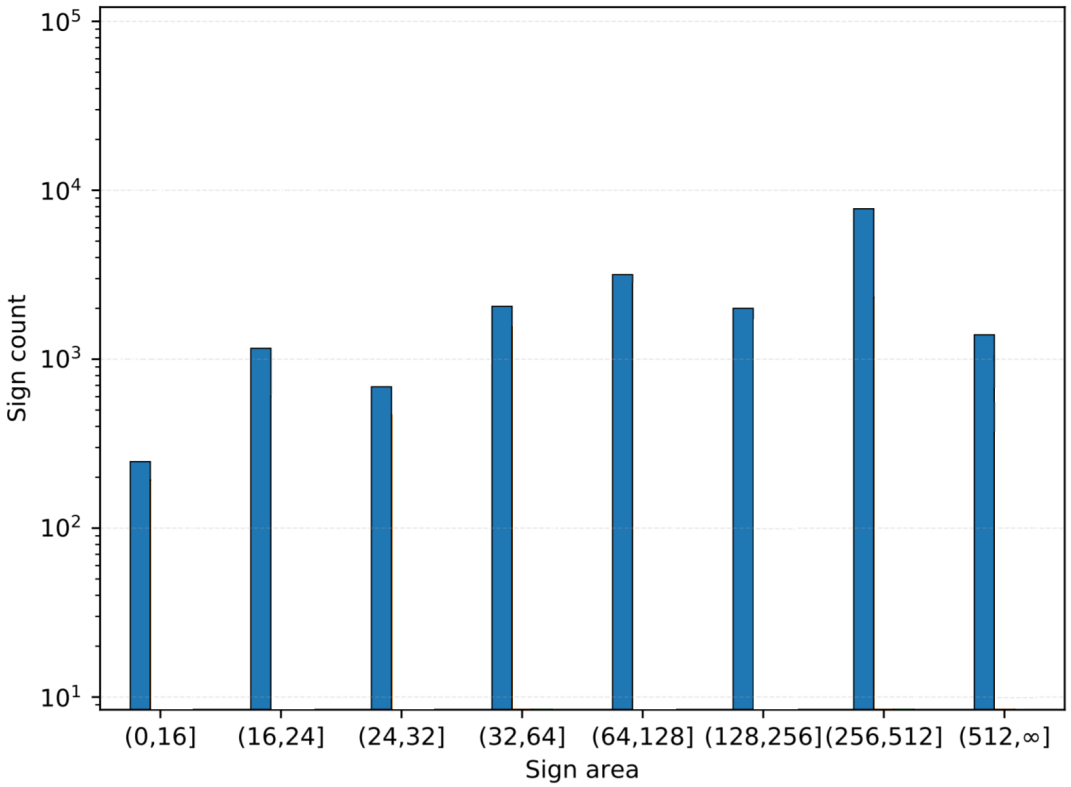


Figure 21 - Distribution of the traffic sign pixel areas in the DFG dataset

2.5 Domain Adaptation in the Context of Object Detection through CNNs

It is very common that the domain at inference time is not the same as the domain at training time. In this particular case, the domain of the input data changes (traffic signs appearances are different among countries) while the task domain (the labels) remains more or less the same (semantic of traffic signs are the same, but each country can also have some specific traffic signs). Domain adaptation is the ability to apply a machine learning model trained in one or more "source domains" to a different (but related) "target domain".

Domain adaptation is a subcategory of transfer learning; in domain adaptation, the source and target domains all have the same feature space (with a few exceptions in this case), but different distributions; in contrast, transfer learning includes cases where the target domain's feature space is different from the source one. So typically the difference is solely in input domain divergence.

Formally, a distribution shift happens when a model is trained on data from one domain distribution (source), but the goal is to make good predictions on some other domain distribution (target) that shares the label space with the source.

In the case of traffic signs, for example, such distribution variation can be caused by slight changes in the appearance of the signs or in their positions within the environment; which is likely to happen among different countries.

Domain adaptation can of course occur in one step (one-step domain adaptation) or through multiple steps, traversing one or more domains in the process (multi-step domain adaptation), in this occasion a one-step domain adaptation is performed.

Depending on the data available from the target domain, domain adaptation can be further classified into supervised (there is labeled data from the target domain, albeit an amount which is too small for training a whole model), semi-supervised (there is both labeled and unlabeled data), and unsupervised (there is not any labeled data from the target domain).

The efficacy of the adaptation heavily depends on the task relatedness between the two domains, where two tasks can be defined to be similar if they use the same features to make a decision (i.e. the features extracted by the backbone network when a CNN is used), which happens in this particular case.

Methods for unsupervised domain adaptation in computer vision can be divided into three general classes:

2.5.1 Divergence-Based Domain Adaptation

Divergence-based domain adaptation works by minimizing some divergence criterion between the source and target data distributions, thus achieving a domain-invariant feature representation (by inducing alignment between the source and the target domains in some feature space).

With such feature representation, the model will be able to perform equally well on both domains. This assumes that such a representation exists which in turn assumes that the tasks are related in some way.

This has been done by optimizing for some measurement of distributional discrepancy. One popular measurement is the maximum mean discrepancy (MMD): the distance between the mean of the two domains in some reproducing kernel Hilbert space, where the kernel is chosen to maximize the distance. The idea is that if two distributions are identical, the average (mean) similarity between samples from each distribution should be equal to the average similarity between mixed samples from both distributions.

However, this approach is formulated as a minimax optimization problem, which is widely known, both in theory and practice, to be very difficult to solve.

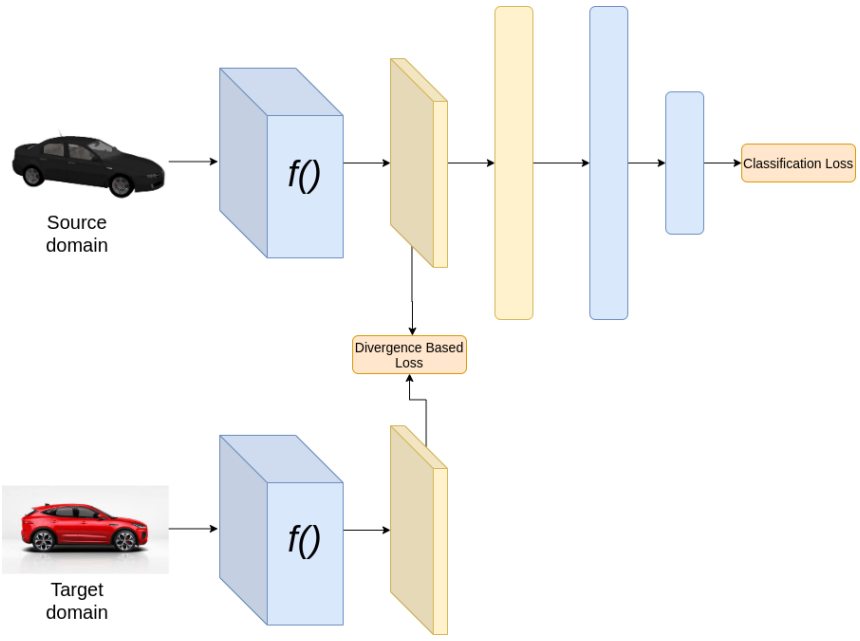


Figure 22 - Scheme of divergence-based domain adaptation training

During training, two losses are getting minimized, the classification loss and divergence based loss. The classification loss is the ordinary training loss, whereas the divergence based loss

assures that the features of the source and target domain become similar in the feature space by updating the weights of the feature extractor.

The divergence loss is based on a mathematical formula which is not specific to the dataset or to the main task

2.5.2 Adversarial-Based Domain Adaptation

This technique was proposed by *Ganin et al.*[20] and tries to achieve domain adaptation by using adversarial training; one approach is to generate synthetic target data which are somehow related to the source domain (e.g. by retaining labels) using Generative Adversarial Networks (GANs). These synthetic data are then used to train the target model. As a simple example, adversarial training means that to align the domain distributions, a domain classifier should be trained and optimized to minimize the domain classification loss, while the parameters of the base feature extractor are getting optimized to maximize this loss, so as to learn a feature space that is domain-invariant.

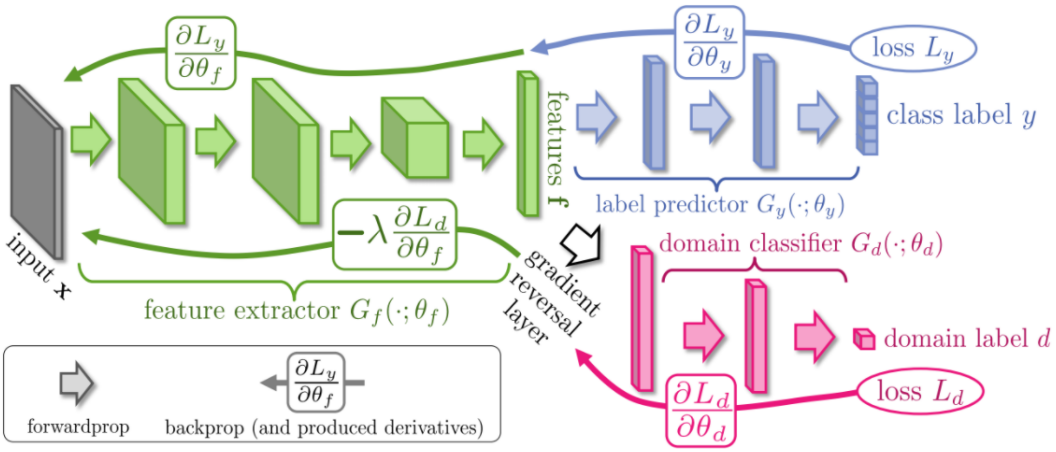


Figure 23 - Scheme of the typical network architecture in adversarial-based domain adaptation

The domain classifier discriminator network learns to extract features which are domain-dependent, hence that are helpful to distinguish between source and target domain samples. The information about those features is passed to the feature extractor network as well throughout the gradient reversal layer. This kind of layer multiplies the gradient to a negative constant so to make the rest of the network (i.e. the backbone) learn to produce more

robust features i.e. features that cannot make the domain of origin be easily distinguished, hence called domain invariant features.

2.5.3 Reconstruction-Based Domain Adaptation

This class of methods, proposed by *Ghifary et al.* [13], trains an encoder-decoder network to modify the samples of the target domain so that they become similar to the samples of the source domain. A discriminator is again trained in parallel with data from both domains to classify the reconstructed samples as real or fake. The discriminator makes sure that during the training process the reconstruction network produces images that are increasingly similar to the source images.

Eventually, a classifier that is trained on the source domain performs the detection on the images of the target domain that are first transformed by the reconstruction network.

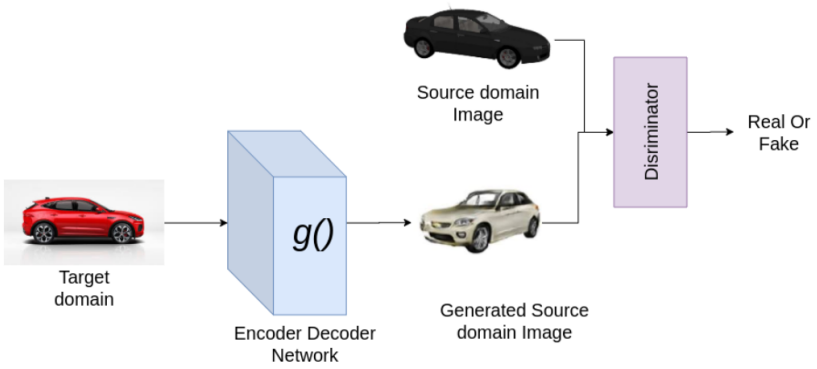


Figure 24 - Training process of the reconstruction and discriminator networks in a reconstruction-based domain adaptation

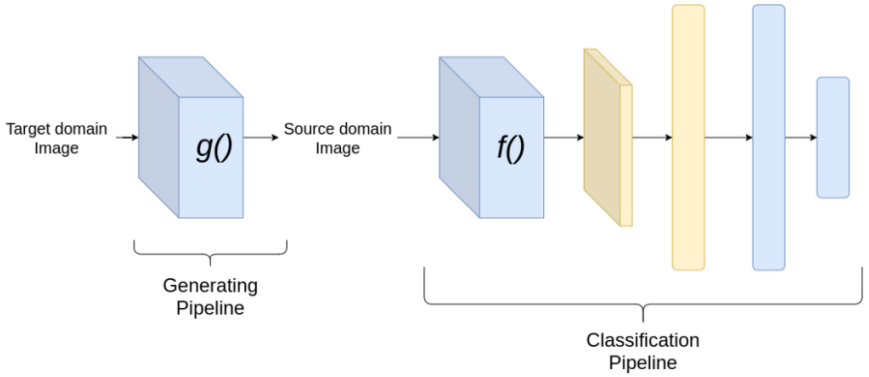


Figure 25 - Inference process of the reconstruction-based domain adaptation

2.5.4 Domain Adaptation with Deep Neural Networks

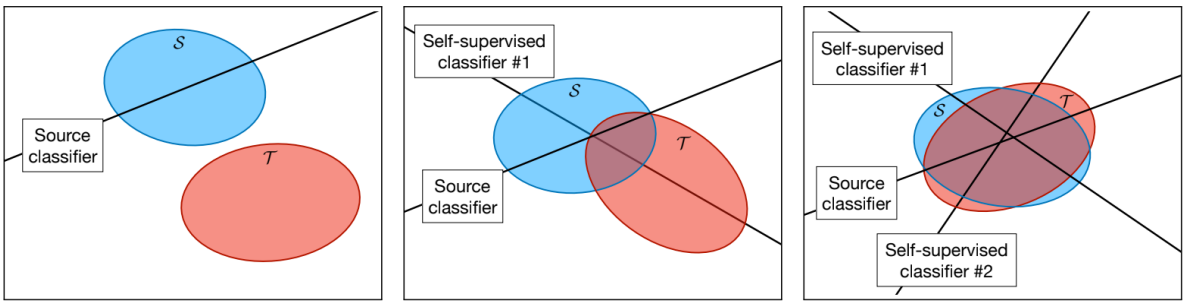
In the context of deep learning, the sophisticated layer-features learned by Deep Neural Networks (DNNs) usually give rise to more transferable representations, generally learning highly transferable features in the lower layers (e.g. edge or corners) while the transferability sharply decreases in higher layers, as the features become more sophisticated. In Deep Domain Adaptation, this property of DNNs is exploited.

2.5.5 Domain Adaptation through Self Supervision

The aim of this technique is again to align the learned representations of the source and target domains while preserving discriminability. This approach is part of the first class and was introduced by *Gidaris et al.*[15] and further developed by *Sun et al.*[16].

The method used in this variant to accomplish the alignment consists in learning to perform auxiliary self-supervised tasks on both domains simultaneously, in addition to the detection task of course. Each self-supervised task creates its own labels directly from the data and brings the learned representations of the two domains closer along the direction relevant to that task in their shared feature space; as such, it has to be possible to automatically generate the ground truth annotations for these tasks.

Figure 26 provides a visual representation of the domain alignment.



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

Figure 26 - Domain alignment in the shared feature space generated by the backbone feature extractor

The key for alignment is to train the model on the *same* (self-supervised) *task in both domains simultaneously.* The distributions align when the self supervised task performs well both on the source and target domain.

Indeed, if there were labels in both domains, the original classification task could have been used for this.

Training this jointly with the main bounding box classification task *on the source domain only* is usually shown to successfully generalize to the unlabeled target domain.
Lacking external supervision (i.e. ground truth), the data itself is used as supervision for these self-supervised auxiliary tasks, which will make the network learn deep feature representations that will hopefully be informative for downstream detection task.
Typically, self-supervision is used as a pre-training step on unlabeled data to initialize a deep learning model, followed by fine-tuning on a labeled training set and evaluating on the corresponding test set.
In this case, the model is trained for the self-supervised tasks in parallel to the main supervised task, favoring a consistent representation that both aligns the two domains and performs well on the main task.

2.6 Model Training on the DLR's Deep Learning Server

All the experiments were carried out on the DLR Deep Learning server, which is endowed with eight Nvidia GeForce 1080 GPUs each with 12GB of memory.
Such GPUs can be used in parallel during training to split the computation of the examples of the training batch and to deploy more complex network architectures that, having more parameters, require more memory; as for example the Faster R-CNN with Feature Pyramid Network.

3. METHODS

This chapter is dedicated to the description of the chosen model's hyperparameters and the modifications carried on the dataset with the aim of achieving a well performing architecture.
Sections *3.1.1.1* and *3.1.2* will provide an explanation of the alterations made on the dataset and the chosen network's parameter respectively. Most of those choices are derived from a series of experiments where many alternatives were analyzed; such experiments will be described in *chapter 4.*

3.1 Data Preprocessing

The process of building or choosing a neural network and training it on a custom dataset should always be preceded by a phase of data preprocessing and data analysis.
Such phases are indispensable in order to be able to understand the most suitable neural network architecture and its corresponding hyperparameters.
Initial data preprocessing generally consists in transforming the dataset into the proper format to be processed by the neural network model. Following, a phase of data analysis is necessary to understand if the information contained in the dataset is appropriate for training the neural network: information should have a high quality (i.e. not ambiguous) and the class distribution should be uniform (i.e. all classes should have similar number of instances among the dataset). It can therefore happen that the process of data analysis gives rise to another phase of data preprocessing, with the aim of optimizing the data for training purposes (e.g. by balancing the data, removing bad quality examples etc.).
The training and model-specific preprocessing needed (i.e. input data normalization, image resizing etc.) will be carried out by the object detection API training pipeline.

3.1.1 Generation of TensorFlow Records and Label Maps

TensorFlow records (TFRecords) are Tensorflow's own binary storage format.
When working with large datasets, using a binary file format for storage of the data can have a significant impact on the performance of the import pipeline and, as a consequence, on the training time of the model. Binary data takes up less space on disk, takes less time to copy and can be read much more efficiently from disk.

Especially for datasets that are too large to be stored fully in memory this is an advantage as only the data that is required at the time (e.g. a batch) is loaded from disk and then processed.
However, pure performance isn't the only advantage of the TFRecord file format: it is also optimized for use with Tensorflow: it makes it easy to combine multiple datasets and integrate the dataset seamlessly with the data import, preprocessing functionality and training functionality provided by the library, included the object detection API.
For using it within the object detection API, the TFRecord must be generated with a specific structure, comprising the image data itself along with its metadata and all the information related to the bounding boxes annotations, such as coordinates and label ids and corresponding names. Since the images will be resized along the preprocessing pipeline, it is necessary that the bounding box coordinates are relative to the image dimensions along the two axis, otherwise a displacement would be introduced on them when resizing the images, which would generate wrong annotations.
Label maps are simple data structures that keep track of the correspondences between label ids and label corresponding class names; therefore those maps only serve for visualization purposes.
It is important to make sure that the label maps between the training and validation datasets are identically generated, otherwise spurious validation losses and accuracy scores would be generated.
It is worth pointing out that the prediction target (i.e. label ids) is automatically encoded in an appropriate format for the structure of the network's output layer (e.g. categorical encoding) and in turn the output layer is generated so that it is suitable to the specific case (i.e. one neuron in the output layer for each possible class).

3.1.1.1 Filtering of Bad Examples

The quality of the dataset instances is an extremely important aspect to achieve good training efficiency for complex neural networks. A dataset containing many bad instances can cause the model not capable of learning the correct features to extract from the images.
This was also known at the time of generation of annotations for the Mapillary dataset; in fact many informative properties regarding each annotated object within the image were added as metadata to the annotations.

Based on that information, an effective filtering of bad instances could be performed quite easily while creating the TensorFlow records for training and validation.
Precisely, object with the three following properties have been filtered out from the dataset:

- Ambiguous: the sign is not classifiable at all; this could be caused by the object being to small or too far from the camera, or in general with impaired visibility
- Dummy: the objects is very similar to a sign but it is not
- Occluded: the sign is (even partly) occluded by another object

Eventually, filtering out objects with these bad properties led to an improvement in both localization and classification performances of the model.

3.1.2 Class distribution Analysis and Alteration

For a correct training of a neural network model, it is essential that the distribution of the classes within the training and evaluation dataset is **uniform**. Having a skewed class distribution would certainly cause the model's predictions to bias towards the most frequent class(es), resulting in poor accuracy on the minority class(es).
This effect manifests itself as a bias in the network towards output values in the ranges for which there were a large number of training patterns. The net result of this is, when test patterns are presented to the system, prediction errors for those patterns with expected results in the well represented region are low while those with expected results outside this region are high.
Since it is an issue very common in this domain of application, many solutions have been proposed.
The most simple approaches are oversampling the less frequent class or performing data augmentation on these last.
Another possibility considers using different learning rates for different classes distribution, lowering it while dealing with classes with a large distribution in order to have an equal influence on the network's parameters coming from all classes. For the latter purpose a weighted cost function was also proposed, which makes possible to lower the influence on the parameters update for the class with higher distribution, and achieve a balanced learning.
Another considered approach is to influence the class distribution within each batch of the training phase, by making sure that the distribution is not skewed, with the purpose to not bias

the model learning. There are many possibilities to obtain this feature, which will not be analyzed.

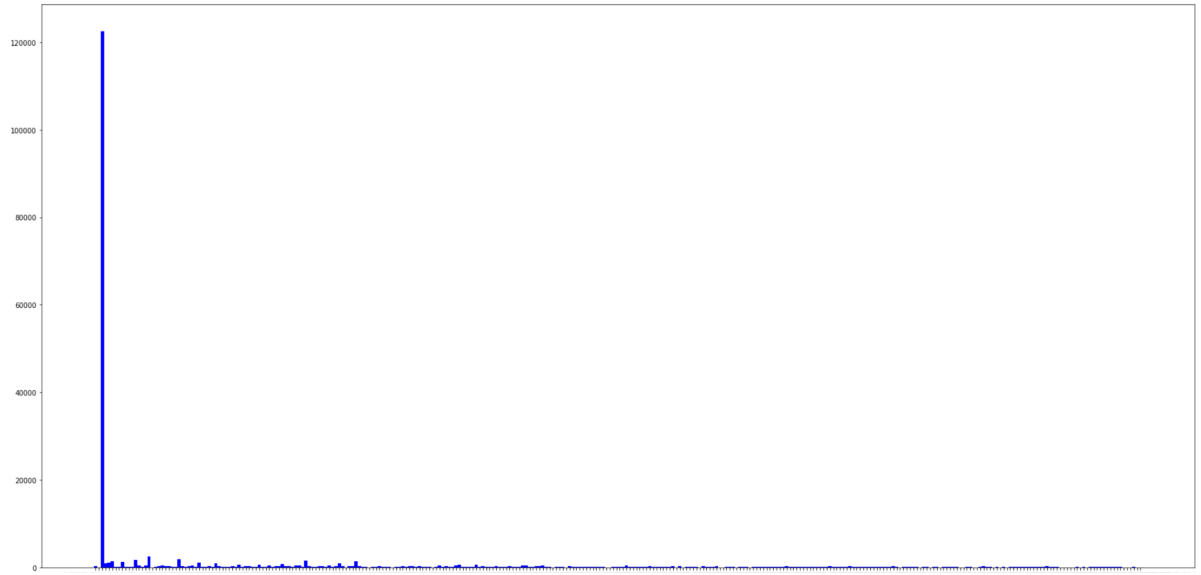


Figure 27 - Histogram of the Mapillary classes, it is evident the higher distribution of the "other-sign" class instance

As can be seen, in the specific case the class 'other-sign' has an exceptionally higher distribution, which is also a general class used when the traffic sign does not represent a specific information. Since the aim is to balance the class distribution and since it is not necessary to detect 'other-sign' instances because of its genetic information, a quite simple approach was applied: the instances of such class have been filtered out from the dataset.
Again, removing this class led to an improvement in the model accuracy in the other ones.

3.2 Model Design and Hyperparameters

Choosing the best configuration for the model's hyperparameters is one of the hardest aspects in this field of research. Given the difficulty, sometimes it is also necessary to proceed by attempts and see how the model's performance changes. This particular phase, in fact, took quite a long time.
Machine learning efficacy depends substantially on two factors: hyperparameters and dataset quality, and they are not independent: they need to be adapted to each other; as such, it is essential to analyze the available data in order to gain knowledge that is essential to

understand which are the optimal (or almost) model's hyperparameters since from the initial configuration.

The model, and often the data as well, need to be changed to gain an optimal compatibility with each other and as it will emerge, this process is not trivial at all. In the specific case, many of the objects scales are excessively small for a standard network setting, therefore either the small objects must be filtered or the network parameters changed to make the model capable of detecting low-scale objects as well.

The subsequent sections will describe the model overall design, which is referred to the final architecture and hyperparameters selected after the completion of the implementation experiments described in *chapter 4*: the Faster R-CNN with ResNet101 as feature extractor and Feature Pyramid Network integration.

3.2.1 Batch Normalization

Batch normalization is applied to layers and it simply normalizes the output of the activation functions.

Every layer in a neural network is learning a representation of the dataset it is trained on as a probabilistic distribution of weights, those distributions are not known before training. Batch normalization is a process to bring numerical stability to these distributions to prevent weights from exploding by normalizing them; it makes sure that the weights within the network don't become imbalanced with extremely high or low values, since the normalization is included in the gradient process.

Varying batch statistics act as a regularization mechanism that can also improve the generalization ability: indeed, using a very large batch size can harm generalization as there is less variation in batch statistics, which decreases regularization.

This can help to minimize overfitting when training for a high number of iterations (which is the case when training from scratch).

Everything just mentioned about the batch normalization process occurs on a per-batch basis, hence the name batch normalization.

Figure 28 shows the steps involved in the batch normalization process.

Step	Expression	Description
1	$z = \frac{x - mean}{std}$	Normalize output x from activation function.
2	$z * g$	Multiply normalized output z by arbitrary parameter g.
3	$(z * g) + b$	Add arbitrary parameter b to resulting product $(z * g)$.

Figure 28 - Steps of the batch normalization process

This calculation sets a new standard deviation and mean for the data; z and b are trainable parameters, meaning that they will be learned and become optimized during the training process.

When fine-tuning on a new dataset, batch statistics are likely to be very different if the fine-tuning examples have different characteristics from the examples in the original training dataset. Therefore, if batch normalization is not frozen, the network will learn new batch normalization parameters (gamma and beta) that are different to what the other network parameters have been optimized for during the previous training. Updating the batch normalization parameters while fine-tuning could improve the performances, but it will very likely require a far higher training time since all the other network parameters need to be re-optimized for them: it would be like training from scratch starting from a good parameter initialization. It must be also taken into account that the fine-tuning dataset could be not large enough to re-learn all the network parameters. Finally, even if the fine-tuning dataset is very similar to the pre-training one, it is not indicated to update the batch normalization parameters, because they would get only a slight change, which however would still require some re-learning of the other parameters.

For these reasons, usually it is better to freeze the batch normalization parameters when a fine-tuning process is used, as was decided in this case.

Conversely, as explained, it is important to train those parameters when training a network from scratch.

3.2.2 Parameters Initialization

As it will be reported later, during experimentations both approaches of training from scratch and fine-tuning where adopted. The latter was employed in the earlier experiments to speed up the training process, and obtain performance results in less time; then a training from scratch was applied with the purpose of earning an accuracy improvement.

To apply a fine-tuning process, in this case the pre-training on the COCO 2017 dataset was exploited.

That has been possible because the pre-training dataset contains images from the domain of interest (i.e. traffic signs). In the specific case of the From the checkpoints all the (pre-trained) parameters of the networks are restored: the feature extractor's weights, the RPN's classification and box regression heads and the Fast R-CNN box classification and box regression heads.

To avoid the problems already mentioned caused by random weights initialization, when training from scratch, parameters are initialized following the rule of "He initialization", that basically tries to make the variance of the outputs of a layer to be equal to the variance of its inputs. Of course, this initialization depends on the layer activation function, which is generally ReLU, for which indeed He initialization is suited.

Since the variance of the input for a given node is determined by the variance of the weights connected to this node from the previous layer, the variance of these weights needs to be shrinked, which will shrink the variance of the weighted sum.

So, it is known that a good distribution for the weights when using ReLU as activation function, should be centered around 0 with a variance of 2/n where n is the number of weights of the layer.

To get these weights to have this variance of 2/n, after randomly generating them with a distribution centered around 0 and a variance of 1, each of them gets multiplied by sqrt(2/n); such a process occurs on a per-layer basis.

3.2.3 Addressing the Small-Scale Objects Problem

Detecting objects of wide scale ranges is certainly a well known problem in the object detection domain that has been tackled for years by now.

As can be seen in *figure 19* the traffic signs pixel areas are quite small, therefore it is not surprising that the problem also affects this specific case. In this occasion, such issue is caused mainly by two factors:

- most of the traffic signs depicted in the images are already small (due to the distance from the camera)
- resizing of images that already have a high resolution causes the small objects of the image to vanish, making the model incapable of detecting them

It is therefore understandable that finding the most suitable and efficient solution for addressing this problem was not trivial and took a long time.
The main parameters that affect this problem are the generated anchors (scales and aspect ratios) and the image size and resizing. In the subsequent sections it will be provided a description of how this problem was tackled, which required also the cropping of images at full resolution as data augmentation.
In addition, the integration of the Feature Pyramid Network architecture to the Faster R-CNN has been necessary to further improve the accuracy on small-scale objects.

3.2.4 Image Resizing and Normalization

Since the images are taken with different devices, the Mapillary dataset contains pictures of various resolutions, most of which are quite large. Because of that, an image resizing must be applied before feeding them to the network. The resize resolution is also one important network hyperparameter, and consequently must be tuned accordingly to the specific problem instance; usually that parameter is chosen among a set of predefined resolutions.
When dealing with images whose original dimensions are rather large, the resizing should not be too substantial, in order to avoid loss of information; in fact the smaller the resizing resolution the harder it would be for the network to detect already small/distant road signs, because they would become more and more small. On the other hand, keeping the resolution too high can cause memory saturation on the training GPUs.
The analysis of the distribution of training images' resolutions was useful to decide what kind of resizing to perform on them: an exaggerated downsizing would make the already small object to almost disappear.

The resolutions distribution can be seen in *figure 18*.

By comparing the evaluation scores (mean average precision) computed on different resizing resolutions, and by analyzing the distribution of training images resolutions, resizing to 1024x1024 turned out to be the optimal solution.

It is worth mentioning that the images' original aspect ratio should also be maintained when resizing in order not to offset the bounding boxes' coordinates (which must be relative to the image dimensions).To achieve this, a zero padding is usually added to the image, instead of stretching one of its dimensions.

Normalizing the input data is important because the larger data points in non-normalized data sets can cause instability in neural networks since the corresponding large inputs can cascade down through the layers in the network, causing imbalanced gradients, which in turn may cause the well-known exploding gradient issue.

For this reason, normalized data should have a mean as close to zero as possible.

To have a better understanding of this, it is worth recalling that the weights update *generally* follows this canonical formula:

$$w_i := w_i - \alpha * \frac{\delta L}{\delta w_i}$$

and that for logistic regression the derivative of the loss function with respect to a weight w_i is (again *generally*):

$$\frac{\delta L}{\delta w_i} = x_i * (a - y)$$

where y is the true class label and x_i is the input feature corresponding to the weight w_i.

For every weight w_i the second term of the partial derivative, (a−y), will be the same. Therefore the differences between the gradients of the different weights solely depend on the inputs x_i: if the inputs are all of the same sign, the gradients will also be of the same sign.

Consequently if a feature of the data is always positive or always negative, it will make the learning process harder for the nodes in the layer that follows: they will have to inefficiently update the weights up and down (because all the weights are constrained to change value in the same direction, positively or negatively) until they reach the optimal value. On the contrary, if the input data undergoes a normalization process so that it has a mean close to

zero, it will be ensured that there are both positive values and negative ones, which will considerably increase the efficiency of the learning phase.

A second positive aspect of normalization is connected to the scale of the inputs. Normalization ensures that the magnitude of the values that a feature assumes are more or less the same.

The larger x_i, the larger the updates and vice versa: the speed of learning is proportional to the magnitude of the inputs.

If the inputs are of different scales, the weights connected to some inputs will be updated much faster than other ones and this generally hurts the learning process.

3.2.5 Feature Extractor Hyperparameters

As mentioned, the backbone feature extractor used is the ResNet 101 with a feature pyramid network, which generates five feature maps (i.e. pyramid levels).

Having images with a high resolution, the feature stride is set to be 16, the higher the stride the bigger the receptive field. A stride of 16 means that two adjacent pixels in the feature maps "summarize" the information between 16 pixels in the original image.

For the aforementioned reasons, when training from scratch, batch normalization's parameters are trained as well, with a momentum for the moving average equal to 0.997, to remove batch noise from the parameters updates.

All the weights of the feature extractor are initialized with a truncated normal distribution with a mean of zero and a standard deviation of 0.03, where values more than two standard deviations from the mean are discarded and re-drawn. Such weights are also used to perform L2-regularization of the network with a weight of 0.0004.

The activation function used within the layers is the rectified linear unit (ReLU), as well as in the RPN and box classifier.

3.2.6 RPN Hyperparameters (First Stage)

The weights are again initialized with a truncated normal distribution with a mean of 0 and a standard deviation of 0.01.

The proposals of the RPN undergo a non-maxima suppression with a threshold of 0.7 for the intersection over union between the anchors and the groundtruth boxes, but no suppression

was applied for the objectness scores, as defined in the paper. The max number of proposals to retain for each image after the post processing phase is set to 300.
It was given more importance to the localization task rather than the objectness loss, by setting respectively a weight of 2 and 1 to their losses.
The RPN box predictor has a depth of 512 and the kernel used within the convolutional layers is set to have size of 3x3.
As classification loss for the box predictor, a softmax *cross-entropy* loss is used.

3.2.6.1 Anchors Generation

In the Faster R-CNN architecture, at each position of the feature map several anchors are generated (in the Faster R-CNN paper, they are nine), which are generated by all possible combinations of the scales and aspect ratios specified as hyperparameters. In the FPN version a different single scale is assigned to each level of the pyramid which is combined with all the aspect ratios to generate the anchors.
Scales and aspect ratios values are among the most important hyperparameters of the Faster R-CNN network and must be chosen wisely in order to detect both small and large objects within the image, without the need of computing different scales for the input images, which is one of the key factor that makes the Faster R-CNN, in fact, faster than its predecessors.
Figure 18 shows that the images contained in the dataset have a variety of different resolutions, but on average it is high.
As already stated, in machine learning, a model's hyperparameters and the training data need to be adapted to achieve the optimal efficacy.
Because of this, ground-truth bounding boxes scales and aspect ratios had to be necessarily analyzed to obtain precious information on the optimal hyperparameters suited to the data, which in this specific case, are the generated anchor boxes scales and aspect ratios when using the Faster R-CNN with FPN.
Figure 19 shows the distribution of ground-truth bounding boxes shapes.
As can be seen, the majority of the traffic signs within the images are distant from the camera and consequently quite small, by resizing also the image to 1024x1024 the resulting objects to detect become extremely small. For this reason, an adjustment of anchors' scales has been necessary, specifically by lowering the scales and adding new ones. As for the aspect ratios of bounding boxes, the ground truth distribution has been analyzed as well in order to achieve a

better consciousness about the aspect ratios which can optimize the network's training process. In *chapter* 4 this analysis will be further detailed.

Given the high original resolutions, an appropriate anchor stride needs to be chosen, which resulted to be 16.

The final configuration of the anchor generator is the following:

```
multiscale_anchor_generator {
min_pyramid_level: 2 #minimum level in the feature pyramid
max_pyramid_level: 6
anchor_scale: 4.0
aspect_ratios: [0.673, 0.955, 0.997, 0.978, 0.941, 1.042, 2.0]
}
```

To further improve the accuracy, the generated anchors which partially lie outside the image window are not removed, but clipped within the image area.

3.2.7 Box Classifier Hyperparameters (Second Stage)

After the first stage, the network generates a crop of size 14 of the feature map for each region of interest of the RPN, which are then fed to the max pooling layer, with both a stride and kernel size of 2.

The second stage box predictor used is the *Mask RCNN*, with both fully connected layers and convolutional layer using ReLU activation function. No dropout or L2-regularization is applied.

The weights of the box predictor are initialized using the variance scaling initialization ("He initialization") with a scale factor of 1. The initializer generates a uniform distribution with a mean of 0, a standard deviation of $\sqrt{\frac{scale\ factor}{average\ number\ of\ input\ and\ output\ units}}$ and bounded to the range [-limit, limit] where $limit = \sqrt{\frac{3 * scale\ factor}{average\ number\ of\ input\ and\ output\ units}}$.

As was for the first stage, the localization loss is weighted more than the classification loss, with a weight of 2 and 1 respectively. This is especially because the localization task is more difficult than the classification one.

For each image, 64 proposals of the RPN are taken by ordering them with respect to the IoU with the ground truth; such regions are cropped from the feature map and the RoI pooling is

performed on them. The classification and box regression losses of all those regions are then combined to compute the total loss of the second stage.
The outcomings of the second stage are the inputs of a post-processing phase, which occurs for every batch; again, the non maxima suppression regards only the IoU between the refined boxes and the ground truth ones, with a threshold of 0.6.
As the post-processing is finished, a maximum of 300 detections are retained for each batch, of which only 100 for each class.
Eventually, the detection scores are converted through a Softmax function, since the problem is a multi-class classification, and softmax function considers all the outputs of the output layer's nodes.

3.3 Training Hyperparameters

Optimization of training hyperparameters is crucial to achieve an efficient training of the model. This section describes the training configuration adopted, again most of which are derived from the experimentations.

3.3.1 Batch Size

The batch size is among the most important hyperparameters; typically a large batch size allows to perform more accurate parameters update since it is generated from multiple images, whereas learning from each single image usually causes to have also bad gradient updates, coming from non-good examples, leading to gradient, and consequently loss, fluctuations. Having a large batch size allows to achieve computational speedups by exploiting GPUs parallelism, but if it is too large it can lead to a model that is unable to generalize. Conversely, using a too small batch size deletes the guarantee that the model will converge to a global optima; in this case the training suffers from sample bias, and the model is likely to overfit the mini-batch distribution, rather than learning the actual distribution of the dataset. Considering those aspects, the actual hardware availability and memory capacity and by performing experimentations described later, a batch size of 8 was chosen for the networks without the FPN integration, otherwise the batch size was halved to 4 due to memory consumption.
In both cases the training was distributed among 4 GPUs.

3.3.2 Training Steps

Another training hyperparameter is the number of steps (one step equals to one gradient update); there are several factors that affect the choice of the latter:

- whether the training is from scratch or it is fine-tuning
- the number of layers within the network
- the dataset size

However, one common and easier practice, which was adopted also in this case, is letting the model train while monitoring the training loss and the loss computed on the validation set at the end of each epoch; when the validation loss begins rising, it means that the model is starting to overfit on the training data. The model state at the previous epoch is generally the one with better training, and can be exported to perform inference on the test data.
The total number of steps is set to 457300 train the model for one hundred epochs, by applying the following simple calculation:

$$\frac{number\ of\ training\ examples}{batch\ size} * num\ epochs\ = number\ of\ steps \Rightarrow \frac{36589}{8} * 100\ =\ 457300\ steps$$

3.3.3 Optimization Algorithm: Momentum Optimizer

The momentum optimizer is an improvement of the SGD optimizer that makes the update navigate along the relevant directions of the loss (i.e. in which it decreases) and softens the oscillations in the irrelevant ones.
It achieves this simply by adding a fraction (typically in a range between 0 and 1) of the direction of the previous step to the current step. This yields amplification of speed in the correct direction and suppresses the oscillation in wrong directions.
This optimizer is used along with adaptive momentum: at the beginning of the learning process a big momentum would only hinder your progress, so in this approach the momentum starts with a low value and once all the gradients are stabilized, the momentum is increased. A typical problem that can occur while using this approach is that the loss function minima is reached with a quite high momentum, which could cause it to miss it or to have oscillations around it.

In the practical implementation the momentum is initialized at 0.4 and is gradually increased to 0.9.

Formally, the update of the network's weights follow the formula:

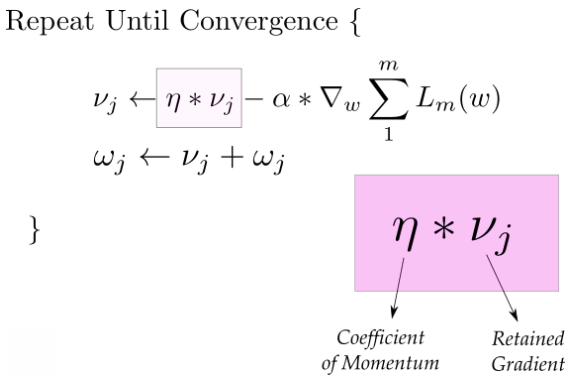


Figure 29 - Formula of the Momentum optimizer's gradient update

Where α is the learning rate.

The previous gradients are also included in subsequent updates, but the weighting of the most recent previous gradients is higher than the one of the older ones.

As an example let's consider the following gradient update:

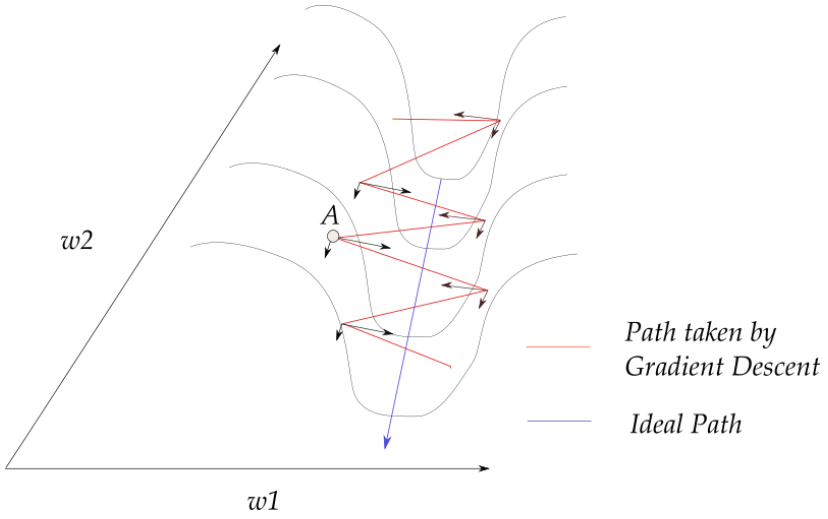


Figure 30 - Visual representation of the weight updates direction along the loss function

Each gradient update has been resolved into components along w1 and w2 directions. If these vectors are individually summed up, their components along the direction w1 get cancelled, while the component along the w2 direction is reinforced.

So by also considering the previous updates, the component along w2 gets reinforced by the previous ones, while the component in w1 is softened. This helps to move more quickly towards the loss minima.

3.3.4 Regularization

Basically, regularization comprises a set of techniques that help to reduce overfitting or reduce variance in the network by making the network structure less complex.
The L2 regularization is the most common type of all regularization techniques, and was also adopted in this case.
During the L2 regularization, the loss function of the neural network is extended by a "regularization term", which is the following:

$$\sum_{j=1}^{n} \left\| w^{[j]} \right\|^2$$

The regularization term is defined as the Euclidean Norm (or L2 norm) of the weight matrices, which is the sum over all squared weight values of a weight matrix. This term is computed by taking into account all the weights of the network.
The regularization term is weighted by a scalar (here called alpha) divided by two and added to the regular loss function. This leads to the following new expression for the loss function:

$$loss + \left(\sum_{j=1}^{n} \left\| w^{[j]} \right\|^2 \right) \frac{\lambda}{2m}$$

Where n is the number of layers, $w^{[j]}$ is the weight matrix for the j^{th} layer, m is the number of inputs and λ is the regularization parameter. λ is another hyperparameter of the model that has to be chosen and tuned; it simply determines how much the model gets "regularized": the larger the value it is set to, the smaller will become the weights of the model, since the SGD tries to minimize the loss as much as possible and the only way to do that for the

regularization term is to lower the weights values. So basically the regularization term penalizes relatively large weights.

To understand the impact of this technique, intuitively, the weights will be set so close to zero, that it would remove or reduce the impact of some of the layers. If that is the case, then it would conceptually simplify the model, making it less complex, which may in turn reduce variance and overfitting.

3.3.5 Learning Rate

The optimal learning rate highly depends on the topology of the loss "landscape" with respect to the network parameters, which is in turn dependent on both the model architecture and the dataset.

Usually the learning rate should start at a high value with the purpose of speeding up the initial convergence to the loss function's global minima and don't get stuck on spurious local minima, and decrease (also in function of the error) as the training proceeds in order not to miss the minima. However, when fine-tuning a network, the weights are already good and they need only a slight change, therefore the learning rate must not have a too high value in order to avoid distorting them.

As mentioned before, the *momentum* is another hyperparameter of the network and it is the term that multiplies the previous gradient update when performing the new network's weights update. Basically it prevents the updates performed on the parameters from undergoing too high variations in direction; his purpose is to prevent non-smooth oscillations in the gradient update, so not to fluctuate around the loss global minima.

Learning rate warm-up

When the training dataset is highly differentiated, the network could suffer from a sort of "early over-fitting". If the data happens to include a cluster of related, strongly-featured observations, the model's initial training can skew badly toward those features, or worse, toward incidental features that aren't truly related to the topic at all.

Learning rate warm-up is a way to reduce the primacy effect of the early training examples. Without it, it may be necessary to run a few more epochs to get the convergence desired, as the model un-trains those early "beliefs".

Learning rate decay

After the learning rate has finished the warmup phase, it begins decaying as the loss function approaches its minima, not to miss it.

The warmup-decay technique is generally useful in fine-tuning too, and given the wide range of features contained in the Mapillary dataset, it was implemented in this case as well. The learning rate decay is the cosine-decay, which gradually decreases the learning rate following the course of the cosine function.

3.3.6 Preprocessing: Data augmentation

The domain of traffic signs is quite complex and instances of it usually have a high variance: different lighting conditions, different angles of the camera/traffic sign, traffic signs of different countries and so on.

Because of this, the network must learn to extract robust and complex features that could be learned only after seeing a lot of different inputs.

Generally, data augmentation is needed also when the network to be trained is complex (i.e. has a lot of parameters) to prevent overfitting.

In order to make the model achieve this behaviour, it has been necessary to define a data augmentation pipeline, paying attention to preserve the semantics of the depicted traffic signs (e.g. no flipping, not too high rotations, not excessive color adjustment etc.) to avoid wrong classifications. Given the particular characteristics of the training dataset (great original resolutions and wide range of ground-truth objects scales), a (random) cropping to the images was required to improve the accuracy of the network on the detection of small objects, which are in turn not so usual in the target domain also due to the lower resolutions of the images (1920x1080). As will be shown in the experiments, the implementation of this augmentation pipeline led to an improvement of the model accuracy, especially on small scales. In that occasion the details of the implemented pipeline will be revealed.

3.3.7 Model Evaluation: Computing Validation Loss

Using the validation dataset to compute the model’s loss is one of the most useful aspects to exploit during the training phase. Such value encloses a lot of information regarding the model's actual expected behaviour on unseen examples, consequently it indicates the model’s

ability to generalize. An anomaly in this loss would indicate that some hyperparameters of the network are not well optimized, hence it can be seen as a diagnostic indicator.

3.3.7.1 Validation-Based Early Stopping Using Cross-Validation

As mentioned previously, the loss computed on the validation dataset is useful also in advanced stages of the training process, since an increase of the validation loss would indicate that the model is starting to become overfitted on the training dataset.
A common practice plans to compute the validation loss every certain number of epochs and stop the training process as soon as the loss begins rising.
This practice has been adopted also in this specific case; since the dataset is quite large (also considering the data augmentation process), the validation loss is computed at each epoch in order to possess updated information at every time of the training process.

3.3.7.2 Score Metrics: COCO Detection Metrics

The set of metrics chosen to compute accuracy is the one used within the COCO detection challenge, because of its large number of variants for mean average precision (mAP) and mean average recall (mAR), these metrics can help to get the in-depth perspective of the model performance.
The COCO detection metrics define several **mAP** detection metrics using different thresholds on the intersection-over-union (IoU) between the predicted and ground truth bounding boxes, including:

- $mAP^{IoU=.50:.55:.60:..95}$: mAP averaged over 10 IoU thresholds (i.e., 0.50, 0.55, 0.60, …, 0.95)
- $mAP^{IoU=.50}$: mAP with the intersection over union threshold of .50
- $mAP^{IoU=.75}$: mAP with the intersection over union threshold of .75

The metrics also include mAPs calculated across different object scales; these mAPs are all averaged over the 10 IoU thresholds (0.50, 0.55, 0.60, …, 0.95):

- mAP^{small}: mAP for small objects that covers area less than 32^2
- mAP^{medium}: mAP for medium scale objects that covers an area between 32^2 and 96^2
- mAP^{large}: mAP for large objects that covers area greater than 96^2

Also **mAR** metrics have different variants:

- $mAR^{max=1}$: mAR with one detection per image
- $mAR^{max=10}$: mAR with ten detections per image
- $mAR^{max=100}$: mAR with one hundred detections per image

Again, mARs is calculated making the same distinctions for object scales as mAPs.
To summarize, the final score is computed by first calculating the average precision (AP) for each class individually across all of the IoU thresholds, then by averaging the APs for all classes to obtain the mAP.
Using the COCO metrics should help to get the in-depth perspective of the model's performance.

3.4 Domain Adaptation through Self Supervision for the Region-Specific North Rhine Westphalia Domain

The general purpose of domain adaptation is to induce alignment of the source and target domains through some transformation. A convolutional neural network maps images to learned representations in some feature space, therefore such alignment is induced by making the distribution shifts small between the source and target domains in this shared feature space. If, in addition, such representations preserve discriminability on the source domain, then a good model can be generated, which still performs good in the source domain and is able to generalize to the target domain under the assumption that the representations of the two domains have the same ground truth.
The peculiarity of the Mapillary dataset is already providing a slight support for this since it includes pictures of traffic signs from various countries taken with different camera devices, however, a process of domain adaptation is still necessary to make the detections of the model not lose accuracy performance on the - different and specific - target domain of NRW. So in this specific case, the Mapillary and DFG datasets are together the source domain, which is very general (since it includes pictures from all over the world and with different camera devices), and the NRW region is the target domain, which is instead specific to the Germany country.

In the context of this project, an supervised domain adaptation through self-supervision is applied between the Mapillary and DFG domains in order to improve the precision of the inferred pseudo-labels used to fuse such domains, so as to improve the fusion accuracy. Afterwards, the self-supervised approach is again applied towards the domain of the NRW region, using the fused dataset as source domain. The motivation behind this design choice is due to the higher simplicity and more customizability of the self-supervised approach; in fact, inserting new self-supervised tasks in the architecture is quite simple, unlike the design of efficacious tasks, as will be discussed.
This section will provide a detailed explanation of the domain adaptation approach experimented.

3.4.1 Unsupervised Domain Adaptation Through Self-Supervised Tasks

Recalling the classes of domain adaptation techniques mentioned in *section 2.5*, the approach adopted in this case is part of the first class, but is based on the domains' alignment through the parallel learning of self-supervised tasks (also called pretext tasks) on both domains, which is simpler than the minimax optimization problem and is shown to yield similar or even better improvements[16]. This technique will be applied on the Mapillary and DFG datasets with the purpose of improving the accuracy and the quantity of generated pseudo-labels, in order to perform a better domain fusion in terms of accuracy. The quality of the datasets fusion would in fact affect the final performance of the CNN model trained on it.

3.4.1.1 Design of Self-Supervised Tasks for Domain Adaptation

The rationale behind such self-supervised tasks is that solving them will force the CNN to learn semantic image features that can be useful for other vision tasks, such the traffic sign detection.
However, the design of effective auxiliary self-supervised tasks is not trivial, since not all of them are suitable for domain adaptation, in order to achieve unsupervised semantic feature learning, it is of crucial importance to properly choose those tasks.
In order to induce alignment between the source and target domains, the labels created by self-supervision should not require capturing information on the aspects where the domains are different (domain-specific information), that is, the aspects of variation that we are trying to eliminate through the adaptation. As a very simple example, if the source domain images

are taken from a country where the direction of travel is switched, the pretext tasks must not consider the position of the traffic signs with respect to the road (i.e. left or right) of that domain (which, by the way, should not be considered anyway) since it's a specific aspect of it that has nothing to do with the traffic signs semantic.

In general, classification tasks that predict structural labels seem better suited for this purpose than reconstruction tasks that predict pixels, since these latter tasks depend strongly on brightness information or other factors of variation in overall appearance that are typically irrelevant to high-level visual concepts.

Therefore, four classification-based self-supervised tasks have been defined, which are the result of an accurate design phase.

Rotation Prediction

An input image is rotated in 90-degree increments (i.e. 0°, 90°, 180°, and 270°); the task for the model is to predict the angle of rotation as a four-way classification problem.

By learning to predict the image orientations, the convolutional neural networks also implicitly learn to localize salient objects in the images, recognize their orientations and object types, and then relate the object orientation with the dominant orientation that each type of object tends to be depicted within the available images. Such implicitly learned knowledge contains semantic information of the target domain images which is expected to improve the cross-domain feature representation power of the network.

In other words, these tasks on target domain images help the network to learn domain invariant feature representation, thus, helps to achieve domain adaptation.

Figure 31 - Example of the rotation task with all four possible rotations

Flip Prediction

An input image is randomly flipped vertically; the task is to predict whether the image is flipped or not. The horizontal flip was not considered since urban scenes' features are typically invariant to them.

Cropping patch prediction

Several implementations have been thought of for this kind of task, which differ also in the task difficulty.

A task too easy to solve has a high probability to be ineffective, just as a too difficult one.

A possibility is to provide the network the original image along with a patch randomly cropped from it, and train the network to predict the location from where this patch was taken as a 4-way regression problem. However, most likely this implementation would not make the network learn the structure of the scenes depicted in the images but just make it learn to compare the two information.

The second imagined possibility planned to copy a random crop from the image and pasting it in a random position on the image area; again this implementation was a 4-way regression problem. This alternative was actually implemented, however the experiments have shown that the task modeled in this manner was way too difficult and as such didn’t bring any improvement to the domain adaptation.

Another implementation of the task has been actually adopted, which is to divide the image in a grid and crop a region from a cell of it. The region is then given to the network as input, which has to predict from which grid cell the region comes from. Hence, the prediction task is easier since it boils down to a classification problem. The number of rows and columns of the grid that divides the image concurs to the difficulty of the task. Because of that a grid of dimension two on both dimensions was chosen, creating four possible regions to crop.

Again, a task of this type implicitly makes the network learn how the typical road scene is structured, making it able to generalize to a different, but similar and related, domain.

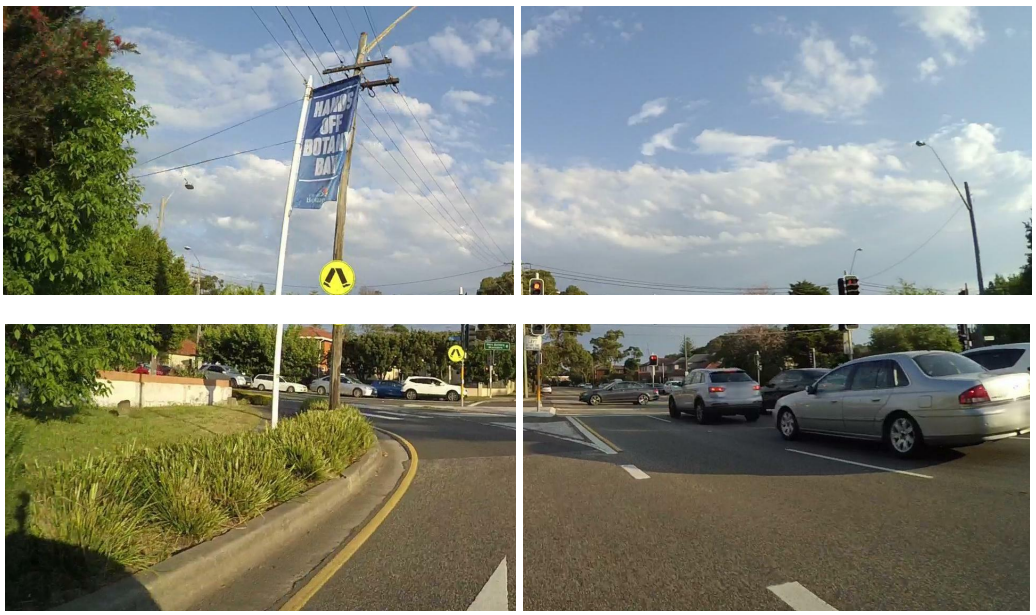


Figure 32 - Example of cropping task with all four possible cropped grid cells

Gaussian Filtering prediction

The image is blurred by applying a Gaussian low-pass filter, with different discrete values of standard deviation (σ). The task for the network is to predict which value of standard deviation was used to filter (so to smoothen or blur) the image. The smoothing effect of the filter is directly proportional to the σ. This is caused by the fact that as σ increases, the weights of closer points get smaller while those of farther points get larger. *This task is particularly suited to increase the model robustness against input corruption, that can be caused by weather conditions, zooming distortion, low-quality lens or image compression.*

Moreover, filtering with different σ can be seen as setting a "scale" of interest to the image so it could even make the network learn features of objects at different scale levels.

The standard deviation was discretized into five values: 0, 3, 6, 9, 12; as such, this task is again a (five-way) classification problem.

Figure 33 - Example of gaussian blurring task. From up left corner to low right corner a filter with σ = 3,6,9,12 respectively is applied.

Edge Detection Classification

This task consists in applying a Sobel edge detector to the input image; two types of filters have been designed, one for the detection of vertical edges, the other for the horizontal edges. The network has to predict which of the two are being detected and enhanced.

The idea behind this task is that it can help the network to learn the structural edges of the scene, and especially of the traffic signs, therefore making it focus on features that are domain-invariant (edges of street environment are quite the same among domains).

*Figure 34 - Example of edge detection classification task. **Left**: image with a sobel vertical edge detector. **Right**: image with a sobel horizontal edge detector*

The tasks have been all intentionally designed as classification problems, with the intent to have a more uniform effect on the network by them.

The core intuition behind using these tasks as the set of transformations is that it is essentially impossible for a CNN model to effectively predict the above classification tasks unless it has first learnt to recognize and detect classes of objects as well as their semantic parts within the image. *Figure 35* schematizes the described approach.

It is important to point out that the main prediction task is performed only on the source domain without any of the alteration needed for the self-supervised tasks (i.e. rotation or flipping etc.).

Once the tasks have been established, a loss function is defined for each of them, so the optimization problem is set up as a combination of these loss functions in addition to the loss of the main prediction task, as is done for standard multi-task learning. At each network step (i.e. weight update) all the tasks are performed.

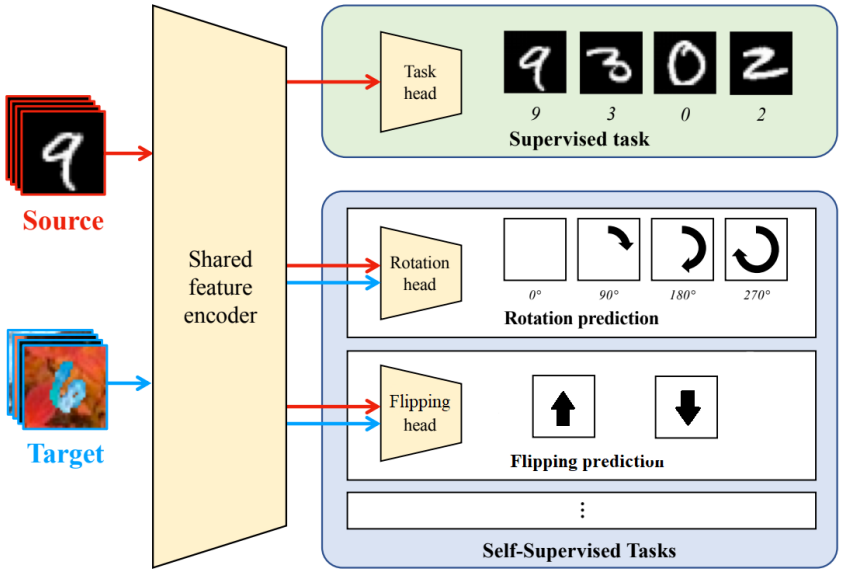


Figure 35 - Scheme of the training process used when applying domain adaptation through self-supervision. At each step, every task is performed, the main task is carried out only on the source domain, while the self-supervised tasks are performed simultaneously on both domains.

3.4.1.2 Network Design and Architecture

A different head (one for each self-supervised task) was added to network architecture, precisely on top of each feature map of the feature pyramid, so as to exploit both high-level features of strong semantic and accurately localized features, since the predictions made on each level are combined together. All the task-specific heads share a common feature extractor (which is the same used also for the main prediction task), that is in this case the backbone Resnet101.

Each loss function corresponds to a single head's output, which produces predictions in the respective label space. The weights of the new heads are added to the set of parameters to optimize during training.

The structure of the heads responsible of predicting the self-supervised tasks should be very simple and, as such, having a low expressivity, in order to make the heads share high-level features (which are domain independent) and force the backbone feature extractor itself to learn and induce the domain alignment in the feature space (by extracting domain invariant features), as shown in *figure 36*. In other words, having simple heads makes the feature extractor do all the work of learning to extract domain independent features from the images.

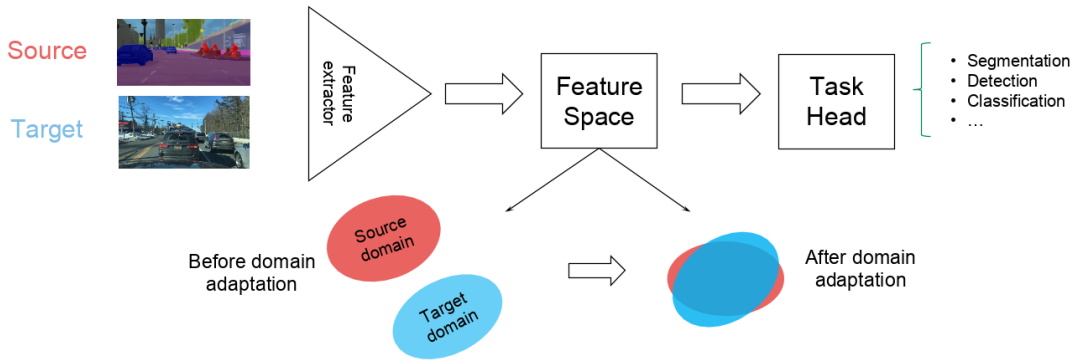


Figure 36 - Scheme of the network's domain alignment in the shared feature space

Denoting $S = \{(x_i, y_i), i = [1...m]\}$as the labeled source data, and $T = \{(x_i), i = [1...n]\}$ as the unlabeled target data.

Each self-supervised task F_k for $k = [1...3]$ modifies the input samples with some transformation fk and creates labels $\hat{y}$.

By denoting $F_k(S) = \{(f_k(x_i), \widehat{y}_i), i = [1..m]\}$ as the self-supervised samples generated from the source samples (discarding the original ground truth labels), and $F_k(T) = \{(f_k(x_i), \widehat{y}_i), i = [1..n]\}$ from the target samples, the loss L_k of each pretext task *k* = [1...4] is made by the following term:

$$L_k(S, T; \varphi, h_k) = \sum_{(f_k(x), \widehat{y}) \in F(S)} L_k(h_k(\varphi(f_k(x))), \widehat{y}) + \sum_{(f_k(x), \widehat{y}) \in F(T)} L_k(h_k(\varphi(f_k(x))), \widehat{y})$$

The optimization problem can be then formalized as in multi-task learning:

$$min_{\varphi, h_k, k=[1...3]} L_0(S, \varphi, h_0) + \sum_{k=1}^{3} L_k(S, T; \varphi, h_k)$$

Where L0 is the loss on the main task:

$$L_0(S, \varphi, h_0) = \sum_{(x, y) \in S} L_0(h_0(\varphi(x)), y)$$

Three variants of the network architecture have been designed and tested:

Architecture 1

The heads predicting pretext tasks take as input directly the feature map generated by the feature extractor. In this case such heads produce a single input for each image.

The network scheme with the additional heads is shown in *figure 37*.

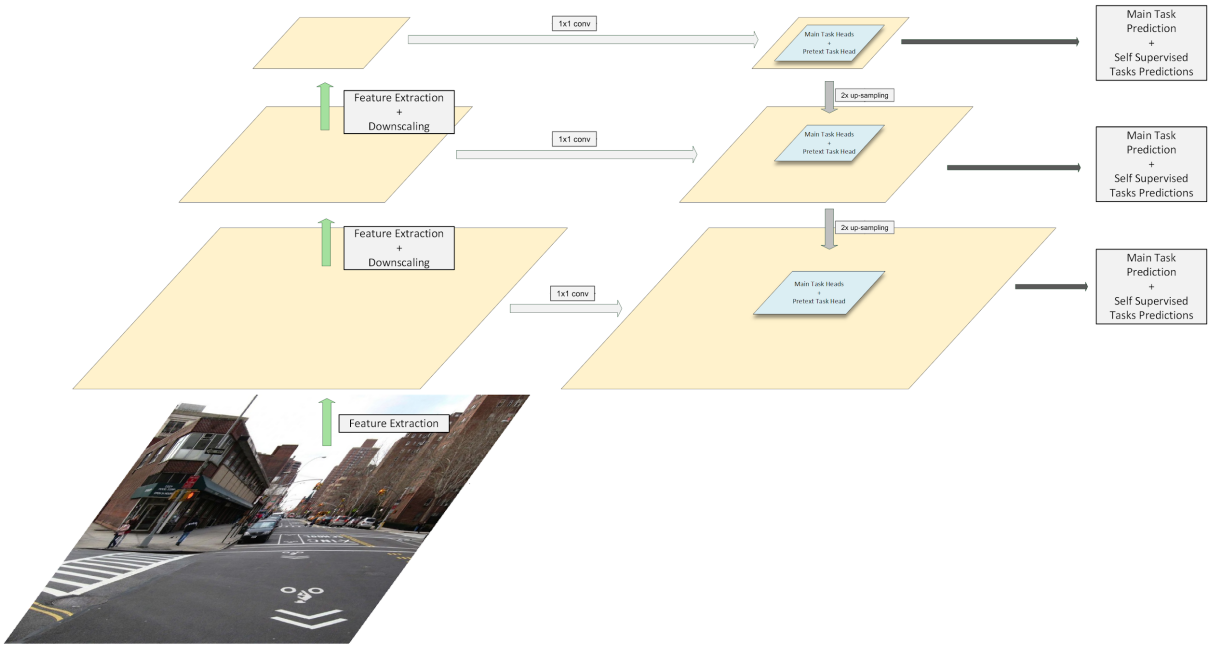


Figure 37 - Scheme of the first network architecture. The self-supervised head of each task is placed on top of each feature map of the feature pyramid network.

Architecture 2

Pretext-tasks heads are inserted in the second stage and operate in parallel to the box classification and regression of the Box Classifier. Their inputs are the cropped and pooled regions of the feature map, proposed by the RPN. For each image, the heads produce a prediction for each region proposed by the RPN, which are, by configuration, 64. As such, the final loss is the average loss calculated by considering each feature map patch.

In addition to those of the feature extractor, this architecture allows to refine the weights of the RPN (first stage) and the Box Classifier (second stage) as well. In this case, additional tasks specific to detection could also be added, for example the edge classification task is well suited for this case.

The network scheme is shown in *figure 38.*

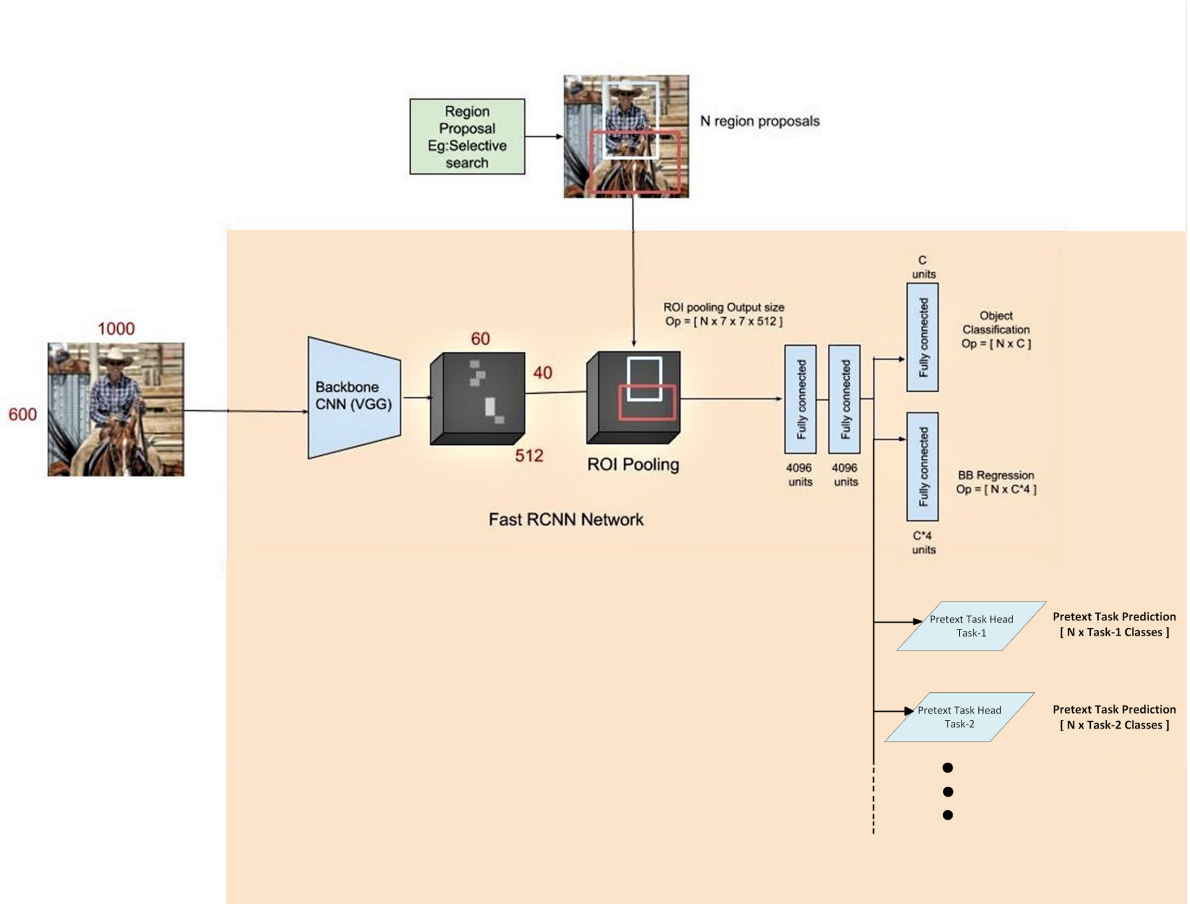


Figure 38 - Scheme of the second network architecture. The heads corresponding to the self-supervised tasks are placed in the second stage and predict in parallel to the Box Classifier.

Architecture 3

This architecture is the result of merging the two previous ones, consequently, it has pretext-task heads both on top of the "raw" feature map produced by the backbone network and in the second stage. The network is shown in *figure 39*.

Again, two architectures for the heads predicting the self-supervised tasks were tested, one with a convolutional layer and the other, even simpler, with just a global average pooling layer.

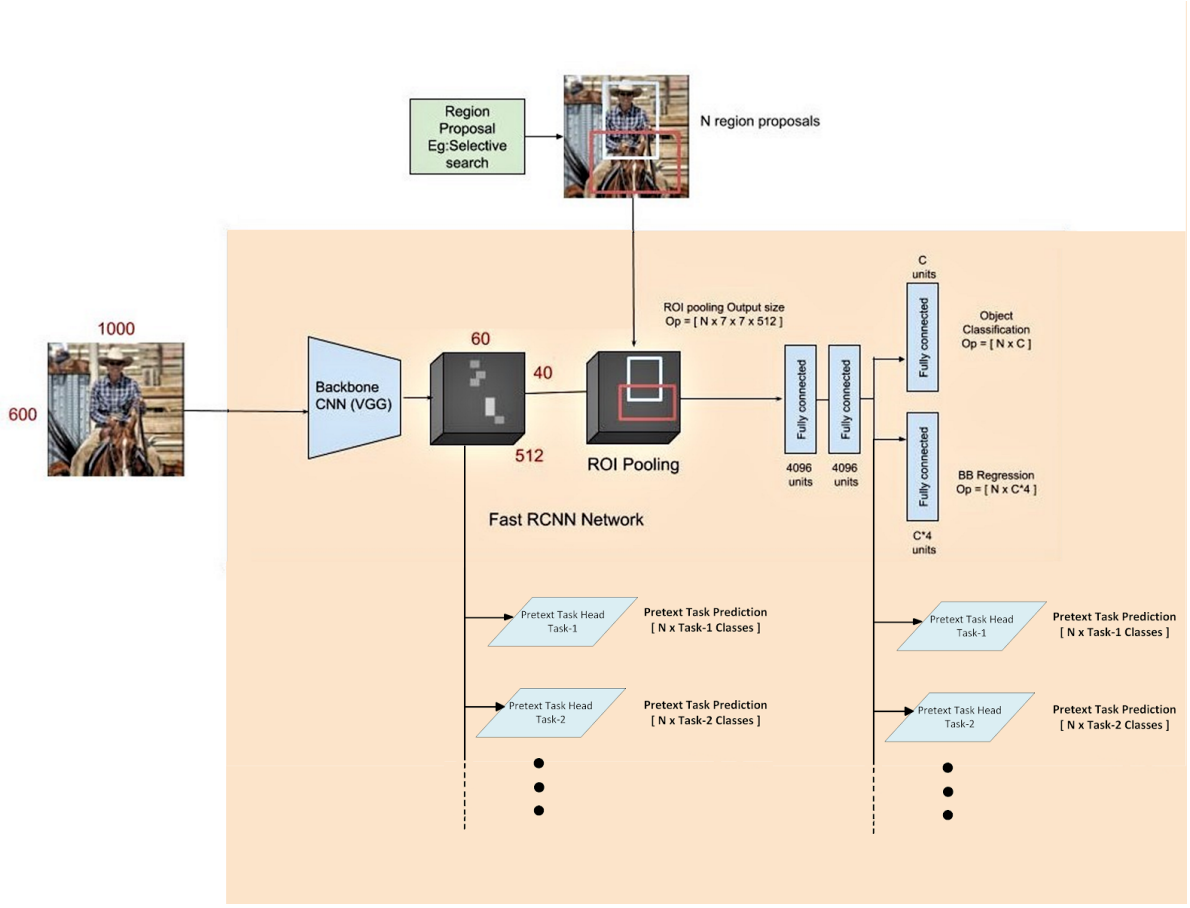


Figure 39 - Scheme of the third network architecture. The heads predicting the self-supervised tasks are placed both "on top" of the feature map and in the second stage.

As can be seen from the schemes below, the designed heads' structure are deliberately kept very simple, for the reasons aforementioned. The first structure (*figure 40*) resulted to be too sophisticated due to the convolutional layer, as such it did not let the feature extractor learn to align the domains. Therefore, a quite simpler head has been designed(*figure 41*). For each channel of the feature map, a single value (the average) is taken.

However, as shown in *figure 42*, the experiments proved that the second version led to better results, which is most likely due to its greater simplicity; for this reason, only the second one will be used for the experiments.

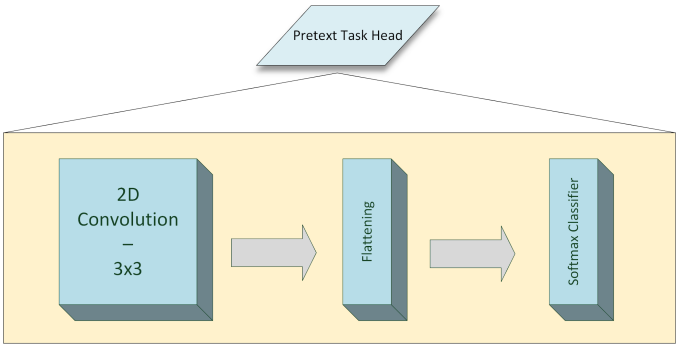


Figure 40 - First architecture of the heads predicting the self-supervised tasks.

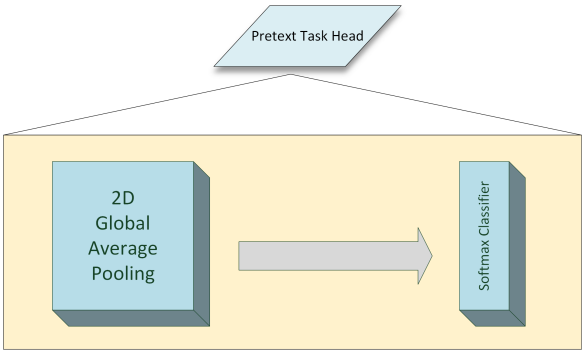


Figure 41 - Second architecture of the heads predicting the self-supervised tasks

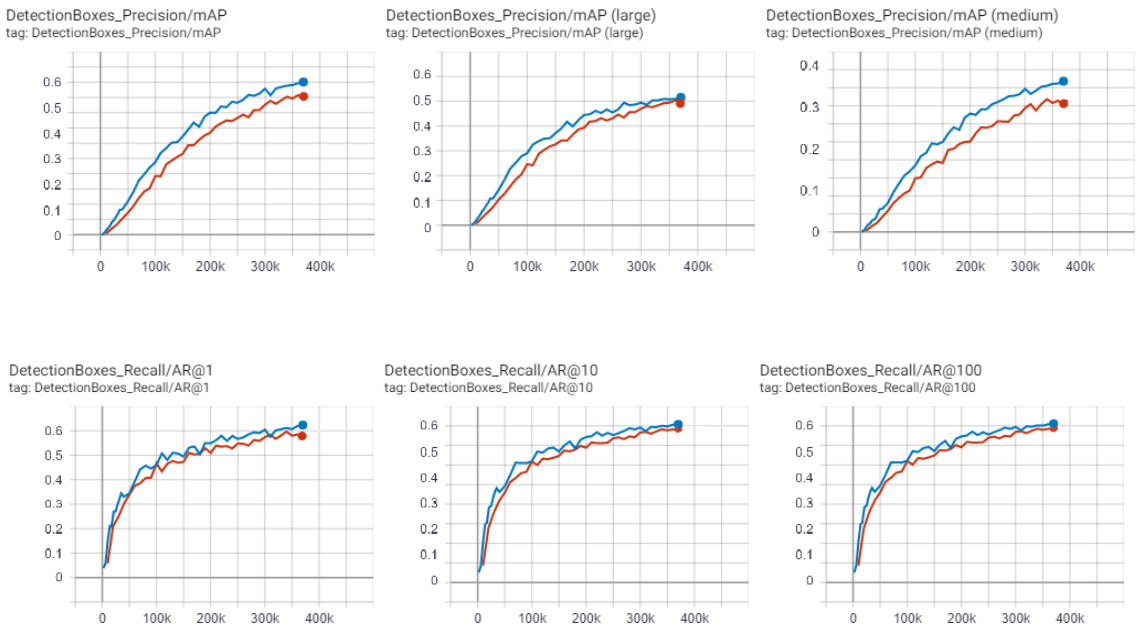


Figure 42 - Accuracy comparisons of the two head architectures in the first network architecture when performing domain adaptation on the DFG domain. The red line represents the first head architecture, while the blue the second one. Similar results were obtained using the other architectures

Self-Supervised Tasks Loss

Since each task was designed as multi-class classification problem, the output of the convolutional/pooling layer is fed to a fully connected layer which is passes the input in the *Softmax* activation function; the output of the last layer is then used to compute the loss on the task using the *cross-entropy loss*:

$$cross - entropy\ Loss = -\sum_{i}^{C} t_i log(s_i)$$

Where t_i and s_i are the groundtruth and the CNN score respectively for each class i of the corresponding task. The combination of the *Softmax* activation function and *cross-entropy* loss is usually also called *categorical cross-entropy loss*, which is typically used for multi-class classification tasks.

3.4.1.3 Domain Adaptation Fine-Tuning

As part of the implementation, a custom training script has been developed to apply the described domain adaptation approach by fine-tuning the network.

Two "virtual" datasets are created, one for the main prediction task and the other for the pretext tasks, the former loads the images and annotations of the source domain, while the latter reads the images of both domains, without any ground truth.

The first dataset has one processing function, which simply loads the image, makes the network preprocess it (performing resizing, normalization etc.) and returns it along with its ground truth annotations. The other virtual dataset, has a processing function for every individual self-supervised task: each function loads an image, modifies it with respect to the corresponding task to be predicted (i.e. rotation, flipping etc.) and generates the label according to the alteration applied. The dataset virtual objects load a batch of examples at a time (which depends on the defined batch size), and the data is repeated infinitely to create epochs; for performance reasons, the subsequent batch of examples is loaded while the model is processing the previous one. Since the datasets are extremely large, cache has been disabled to avoid memory exhaustion.

When performing the tasks, it would be intuitive to perform a joint step over all the tasks and calculate the related losses together; however, the implementation would then have to store the gradients with respect to all losses together.

Instead, for memory efficiency, the implementation loops over all the tasks. For each self-supervised task it samples a batch of combined source and target data, structurally modifies the images by the corresponding function for the task (i.e. by rotating, flipping etc. the image), creates new labels according to the modification, and obtains a loss to perform the backpropagation on the predicting head and the feature extractor.

Finally, a batch of original source images (the same used to perform the pretext tasks) and labels are sampled for a gradient step on the main prediction task, therefore for each task a gradient update is performed. Performing a single gradient step was also tested, but the accuracy difference was negligible, so the memory saving had a good trade-off.

To make sure the model doesn't bias towards one of the two domains when aligning the the domains in the feature space, the examples' domains in each mini-batch are balanced, which means that a batch contains the same number of examples of both domains, and the same samples of the source domain are used to perform both the detection task and the pretext ones;

The model parameters and hyperparameters are almost the same used for the training from scratch. Since it is a fine-tuning process, the learning rate is set to 0.001 and 0.004 for the warmup and base value respectively. The momentum optimizer with cosine-decay learning rate was used in this implementation as well.

A checkpoint of the model state is saved every 10'000 steps, which is also used for the validation purposes described in the next section.

Every 1'000 steps, tensorboard logs are generated for the main task and self-supervised tasks losses; of course each loss is used to calculate the gradients of the weights only when the related task is being predicted.

For each layer of the network, an histogram of the weights values is created, which are used as a further check the correct functioning of the training process.

Order of the Tasks

The optimal order of the tasks in terms of adaptation is not trivial to determine. However, the order between pretext tasks alone is unlikely to have a high influence on the process; the most significant aspect is the order between the pretext tasks and the main one. As a first intuition,

it is probable that performing the pretext tasks first would yield a better generalization effect on the target domain, since the network would need to learn to perform good detections after the weights have been modified along the directions of the self-supervised tasks. Both cases will be experimented and compared in *section 4.2.1*.

Self-Supervised Losses Influence

To further improve the domain alignment while preserving a good performance on the main task, the losses generated by the self-supervised tasks are weighted differently, so as to not make the pretext tasks subdue the main one. Experiments with different losses weightings techniques have shown that a good approach is to weigh those losses equally, with the weights set so that their sum is equal to one, which is the weighting of the main task (i.e. 100%). Such weighting is simply obtained by setting the weight of each pretext task loss to $\frac{1}{number\ of\ pretext\ tasks}$.

This ensures that all pretext tasks together do not dominate the main task in the backpropagation, so to maintain weights that are good for the detection task and only slightly adjusted by the domain-adapting tasks.

Model Convergence in Domain Adaptation

Another problem is that the model convergence while performing domain adaptation is usually slower and more subject to fluctuations than with a normal training, especially on the self-supervised tasks. To help the convergence of the network, the learning rate decay has been slowed down; more precisely, the decay time was doubled, so that the learning rate reaches zero after one million steps.

Another way to help solving this kind of issue is to generate equally distributed batch samples during training, with respect to the pretext tasks. In this specific case, it means to ensure that every batch of samples contains each of the four rotations, both flippings, a crop taken from every grid cell, an image for each defined gaussian filter and both edges detections. Of course this also depends on the batch size.

For comparative purposes, this domain adaptation process has been applied by fine-tuning a model pre-trained on the COCO 2017 dataset (the same used for fine-tuning the networks during the experimentation phase).

Pretext Tasks Input Data

The described approach consists in using the whole images from both domains simultaneously to perform the pretext tasks, in order to induce an alignment in the feature space.

To force the network to focus on the feature actually deriving from the traffic signs, a variant of the technique has also been tested, where crops of the traffic signs are used to perform the pretext tasks, instead of the entire images. To achieve this, the images of the source domain have been cropped using the pixel coordinates of the ground truth labels; as for the target domain, when dealing with a dataset endowed with ground truth again its coordinates are used. Therefore, this case is suitable for supervised cases, where the datasets contain labels. A possible use in an unsupervised case would require to generate pseudo-labels on the unlabeled domain and use such labels to crop the image.

The cropped regions have been slightly enlarged to include a part of the background in it, which should hopefully help the network better learn to distinguish it from the actual traffic signs.

Domain Adaptation on Domain-Specific Classes

The objective of this phase is to adapt the model from the Mapillary domain to the DFG domain and vice versa in order to gain an accuracy improvement with respect to the pseudo-labels generated in the target domain. Therefore, if the approach is proven to yield a good generalization on the target domain, an additional modification of the algorithm consists in filtering the classes of the source domain, so to use only those specific to such domain while performing the domain adaptation. Eventually, the images of the source domain used for the main task and for the pretext tasks are only those containing at least one class specific to such domain.

In order to apply this strategy, another dataset has to be created, containing only annotations of classes specific to the source domain.

With this alteration applied, the model would learn to successfully generalize on the target domain and at the same time become specialized in the source domain-specific classes that are going to be injected into the target domain in the successive step.

S*ection 4.2.1* provides the outcomes of the experiments' of the domain adaptation variants described in this section.

3.4.1.4 Validation Based on Mean Feature-Space Distance and Main Task Loss

In the unsupervised domain adaptation there is no target label available and therefore no target validation set, so typical strategies for hyper-parameter tuning and early stopping that require a validation set cannot be applied. Because of this, a simple heuristic function has been designed and implemented, which combines a measurement of the distributional discrepancy between the two domains in the feature space and the loss on the main detection task. For the discrepancy measurement, the heuristic uses the distance between the mean of the source and target samples in the learned representation space, produced by the backbone feature extractor (the ResNet101).

Formally, such distance is given by the following formula:

$$D(S', T'; \phi) = \left\| \frac{1}{m} \sum_{x \in S'} \phi(x) - \frac{1}{n} \sum_{x \in T'} \phi(x) \right\|_2$$

Where S' is the source validation dataset and T' the target validation dataset, both without labels. The heuristic consists in a simple sum of the feature-space distance and the main task loss, so to give importance both to the domain alignment and the performance on the detection task.

Such computation occurs at each epoch and the one selected for early-stopping is simply the epoch with lower heuristic value.

3.4.2 Pseudo Labeling Process Design

Since the objective of domain adaptation is in most cases to adapt a model from a generic class distribution to a specific one, it can easily happen that the target domain has some specific classes that are not present in the source (the classes of the two domains are semi-overlapped). In this case, both the Mapillary and DFG domains have their own specific classes, however it is necessary that the model learns them as well, which is not possible by only applying the self-supervised approach.

For this reason, an additional approach to the self-supervised training has been designed; such an approach also aims to explore and show the possible improvements produced by

integrating the unsupervised domain adaptation through self-supervision with a training on a dataset that includes more (different) domains.

The process used to merge the domains is mutual cross-domain data augmentation, which consists in making a model (trained on a source domain) generate pseudo-labels in the other target domain; the intuition is that *such pseudo-labels are injecting classes specific to the source domain in the target domain.*

Both the task of generating good quality pseudo-labels and evaluating them are anything but trivial, also because the two considered domains are quite different.

Because of that, to improve the accuracy of the pseudo-labels, the model must be first adapted to the target domain using self-supervision approach described before in order to avoid a decay of the model performance due to the domain distribution shift, then it can be used to perform inference on the data of such domain to extend its ground truth.

To achieve an accurate result on this, the domain adaptation must be applied in both directions on two complementary models, by switching the source and target domain in the adaptation process. At the end, two distinct datasets will be generated, which are the original datasets extended with pseudo-labels on the instances of classes specific to the other domain.

The steps of this approach can be summarized as follows.

Given two domain D1 and D2, and two models M1 and M2:

1. Train a M1 on D1 and perform unsupervised domain adaptation on D2
2. Perform inference on the unlabeled data of D2 using M1 to generate additional data for it, called D2.1
3. Train M2 on D2 and perform unsupervised domain adaptation on D1
4. Perform inference on the unlabeled data of D1 using M2 to generate additional data for D1, called D1.2
5. Merge D1.2 and D2.1 in a single dataset and train a model M3 on it

As already mentioned, this approach has been applied using the Mapillary and DFG datasets; The hyperparameters of the model trained on the DFG dataset have been slightly changed with respect to the aforementioned ones, in order to be suitable to its different characteristics.

The idea behind this approach is that training a model on the resulting fused dataset could lead to even better accuracy on the real target domain of NRW, compared to using only one

dataset as source domain. In fact, by fusing the datasets it is possible to train a model that is capable of recognizing a bigger variety of sign classes, coming from two different domains, and that are relevant to the final objective; In addition, while training, the model should learn a higher variety of features since it learns them from more domains. Therefore, using this fused dataset for the main task when performing self-supervised domain adaptation on the NRW domain should further improve the accuracy on such domain with respect to using only a single dataset.

3.4.2.1 Dataset Extension and Fusion for Multi-Domain Training

The process of mutual extension of the datasets is not as simple as it might seem; when extending a dataset, only the specific classes of the source domain are injected in the inferred domain and keep the original name. To have a good trade-off during the pseudo-labeling among false positives and false negatives, the inferred labels are filtered by the confidence score given by the model with a threshold that slightly favors the false positives (i.e. the recall), since false negatives are usually more harmful for the learning process; such threshold is determined by taking the optimal value of the precision-recall curve that averages the score of each class.

In addition, the pseudo-labels of a common class are compared with the actual ground truth: for each pseudo-label the IoU with all the original labels is computed and those that have an IoU higher than 0.7 with any of them are considered overlapped with the ground truth and filtered.

Since objects of class *other-sign* will be filtered from the Mapillary dataset also during the merging process, such objects are not considered in the IoU filtering, with the attempt to replace most of them with pseudo-labels having a more informative class.

Eventually, the two datasets obtained at the end of this process have to be merged together, to do this, first of all a naming convention for the classes in common to the two domains must be chosen, by converting such classes' names to the ones of the selected domain. Of course, domain-specific classes have to be kept and united, and for them the original names are used. The merged dataset eventually contains the common classes and the specific classes of both domains. The Faster R-CNN Resnet101 with FPN can then be trained on such dataset to learn more and better features to detect and classify the traffic signs from both domains. The self-supervised tasks on the NRW should be performed in parallel to this training, also

because it can likely happen that some of the generated labels are not accurate, but *it has been proven that self-supervision also reduces the distortion effect caused by corrupted labels*[17].
The comparisons between the mentioned techniques will be discussed in *Section 4.3*.

3.4.2.1.1 Evaluation of the Generated Pseudo-Labels

Evaluating the quality of the pseudo-labels is quite a difficult task, and it is indeed an active research topic, as such, some pseudo-metric measures have been designed and applied for this specific case.
The pseudo-labels overlapping with the ground truth are used to calculate the first pseudo-metric. The percentage of class concordance between the ground-truth labels and pseudo-labels marked as overlapped is used to compute the first metric, which measures both the efficacy of the self-supervised domain adaptation process and the accuracy of the generated pseudo-labels, this metric is computed over the entire dataset with the following formula:

$$Overlapping - Concordance = \frac{number\ of\ concordant\ overlapping\ labels}{number\ of\ total\ overlapping\ labels}$$

As additional information, the number of inferred annotations and of new classes are also considered. Also such two values should give a (less precise) hint on how the domain is adapted to the target domain. The accuracy of the pseudo-labeling process uses the pseudo-metrics just described and is shown in *section 4.2.2*.

3.4.2.1.2 Filtering of Difficult Images

The difficulty of performing a traffic sign detection task on an image can depend on different factors, for instance, the depicted traffic signs have a small scale or are occluded by other objects. Images with these properties can likely negatively influence the quality of their pseudo-labels, therefore a filtering process is carried out during a fake pseudo-labeling process, before the real pseudo-labeling is performed. The CNN models used to perform such filtering are trained on the entire source domain while being "domain-adapted" to the target domain through self supervision.
Before inferring the pseudo-labels, all the variants of the whole set of self-supervised tasks are predicted on the image. This means that the image is rotated in all four possible rotations,

flipped and not flipped, and a region from each 2x2 grid cell is cropped. The accuracy on each of these variants is calculated, and eventually the mean accuracy is calculated over all pretext tasks, this accuracy takes the name of *self-supervised-accuracy*.

In addition the *Overlapping-Concordance* is computed for each image (this is possible only when the model is trained on all source domain classes).

Eventually, an image is marked as "difficult" and skipped if one between the mean accuracy on the pretext tasks and the image *Overlapping-Concordance* is below 0.5.

3.4.3 Reconstruction-Based Unsupervised Domain Adaptation on NRW Domain

As an additional phase to the domain adaptation, the model obtained by applying the training approach just described is used to generate pseudo ground truth in the NRW target domain. Again, the quality of those pseudo-labels should have been improved by the process of domain adaptation on such domain.

The inferred pseudo-labels are going to generate a pseudo-dataset containing data of the NRW domain, which is used to fine-tune the parameters of the model.

This approach will be tested and its influence on the model performances is shown in *section 4.4.2*.

4. EXPERIMENTAL RESULTS

The first part of this chapter is dedicated to the description of the process of hyperparameter tuning that has been necessary both to achieve an effective training and an adequate accuracy especially on the Mapillary dataset.

Section 4.2 focuses instead on the experimentation of our domain adaptation approach, by investigating the variants and factors that have a more significant influence on its efficacy. The experiments' results presented in this chapter will clarify whether our particular domain adaptation method is suitable for the case of traffic sign detection and what are the aspects that optimize its success.

4.1 Model Architecture and Hyperparameter Selection

The choice of the best convolutional neural network is not trivial, in this instance, a particular model of the main networks used in the context of object detection was chosen to be tested and compared: *Faster R-CNN*[5] and Faster R-CNN with FPN integration; since it represents an adequate good trade-off between accuracy and speed performances.

Faster R-CNN models are two-stage detectors, therefore are better suited to cases where high accuracy is desired and latency is of lower priority. Conversely, if processing time is the most important factor, SSDs are recommended. The raw SSD network was not considered due to its architecture-intrinsic low accuracy in detecting small-scale objects (see *section 2.3.2.2.1*), which are instead common in this specific case (see *figure 19*), so the network with additional FPN was tested, since such integration improves precisely the small object detection accuracy. For the Faster R-CNN the Resnet (*section 2.3.1.1*) was chosen for the backbone feature extractor.

Other models were also considered, such as EfficientDet, which promised very high accuracy, however its complexity resulted to be unsustainable for the available resources.

Many experiments were carried out during the phase of model selection, involving various architectures, for each of which, many different hyperparameter configurations have been tested. The aim of this section is to provide an overview of how this engineering phase has proceeded, which approaches have been adopted and their motivations.

As stated previously, all the implementation attempts described subsequently were made by restoring the parameters from a pre-trained model on the COCO2017 dataset and by

fine-tuning the network on the Mapillary dataset. Except for the Faster R-CNN with FPN, for which it was decided to apply a training from scratch.
Additionally, a process of cross-validation was always carried out along the training in order to monitor the generalization ability of the network.

4.1.1 Faster R-CNN ResNet50 640

The first network architecture chosen for this application case is the Faster R-CNN; this choice was mainly motivated by the fact that this type of network has a higher accuracy compared to the SSD network[5],[6], albeit sacrificing inference speed performance, which in our case is not necessary unlike accuracy.
As for the initial version of this type, Faster R-CNN with ResNet50 as backbone and input resizing to 640x640 was selected. Being the simplest one, the 50-layer variant should represent a good trade-off between accuracy and speed performance[2].
Such a substantial resizing allowed a bigger training batch, but also involved low performances on small objects since the beginning.
With the attempt to improve the model accuray on small objects, the default scales for the anchor boxes (see *section 2.3.2.1.1*) are lowered: `[0.1, 0.25, 0.75, 1.25, 1.75]`.
However, even this final configuration leads to quite low performances (see *table 6* and *7*), confirming that this architecture is not appropriate.

***Accuracy scores on Mapillary and NRW validation samples** (4.1.7)*

Val. Set / Metric	mAP	mAP@0.50IoU	mAP@0.75IoU	mAP (small)	mAP (medium)	mAP (large)
NRW	0.2063	0.3748	0.2486	0.0465	0.2764	0.5445
Mapillary	0.1047	0.1677	0.1234	0.0074	0.1341	0.1893

Table 6 - Mean average precision scores (COCO detection metrics) of Faster R-CNN ResNet50 with input resizing to 640x640

Val. Set / Metric	AR@1	AR@10	AR@100	AR@100 (small)	AR@100 (medium)	AR@100 (large)
NRW	0.2824	0.3264	0.3876	0.1235	0.3947	0.5193
Mapillary	0.1456	0.1520	0.2271	0.0349	0.1425	0.2847

Table 7 - Average recall scores (COCO detection metrics) of Faster R-CNN ResNet50 with input resizing to 640x640

4.1.2 Faster R-CNN ResNet50 1024

A second attempt is made with the same network architecture, changing what was the main problem of the previous one (which is also among the main problematic aspects of the specific case): small-scale objects detection. Here, the image resizing was changed to 1024x1024, using again the paper's default aspect ratio configuration.

The detection of compact objects improves slightly from the previous model. However even lower anchor scales do not help the performances.

Accuracy scores on Mapillary and NRW validation samples

Val. Set / Metric	mAP	mAP@0.50IoU	mAP@0.75IoU	mAP (small)	mAP (medium)	mAP (large)
NRW	0.2641	0.4512	0.3154	0.1034	0.4187	0.7121
Mapillary	0.1689	0.2139	0.1841	0.0127	0.1763	0.2432

Table 8 - Mean average precision scores (COCO detection metrics) of Faster R-CNN ResNet50 with input resizing to 1024x1024

Val. Set / Metric	AR@1	AR@10	AR@100	AR@100 (small)	AR@100 (medium)	AR@100 (large)
NRW	0.3532	0.3927	0.4265	0.2482	0.4927	0.6714
Mapillary	0.2246	0.2538	0.2971	0.1003	0.1679	0.3015

Table 9 - Average recall scores (COCO detection metrics) of Faster R-CNN ResNet50 with input resizing to 1024x1024

4.1.3 Anchors Generation, Dataset Filtering and Faster R-CNN ResNet101 1024

In this implementation test, the network architecture is changed, by deploying the Faster R-CNN Resnet101; generally, a more complex network should gain higher accuracy, renouncing at speed performance, since it should learn more complex and robust features in the deeper layers. Given the previous results, the input data resizing should not be lower than 1024, higher values were avoided as well to allow a batch size of at least 8 examples.

As previously, the first implementation test is made using the default values proposed in the paper, resulting in this initial configuration, for the very first test, a batch of size 8 is chosen with the aim to improve the generalization ability:

Hyperparameter	Value
Feature extractor	ResNet101
Resizing dimension	1024
Anchors scales	[0.1, 0.25, 0.74, 1.25, 1.75]
Anchors aspect ratios	[0.5, 1.0, 2.0]
Anchors width/height stride	16
Batch size	8
Learning rate base	0.0004
Training steps	457300
Momentum	0.9

Training hyperparameters

Such configuration leads to exploding gradients and consequently to huge loss values.

To counteract this problem, a lower learning rate is set in the learning configuration, going, after some attempts, from .00001 to .00000004 for the base learning rate and to .0000000133, also keeping in mind that in a fine-tuning process the learning rate should be around ten times lower than the one used for training the network from scratch.

Such values causes the training convergence to be extremely slow; even enlarging the number of steps from 10000 to 50000 and subsequently to 120000 does not help getting even close to convergence, inability to converge to a (global) minima is in fact a common possible side effect of having a too small batch size.

The results prove that this training configuration is not proper, therefore the batch size is restored back to 8.

Since the network is more complex than the previous one the learning rate is slightly enlarged to .0004 for the base and .000133 for the warmup.

Lower scales for the generated anchors are also added, reaching this initial scales configuration: [0.1, 0.25, 0.75, 1.25, 1.75]

This configuration leads to a better accuracy since the earlier steps, but they were not enough for the network to converge; To ensure convergence, the number of steps is chosen to cover 100 epochs, which with a batch size of 8 would be:

$$\frac{number\ of\ training\ examples}{batch\ size} * num\ epochs\ = number\ of\ steps \Rightarrow \frac{36589}{8} * 100\ =\ 457300\ steps$$

After the training is finished it can be seen that performance on small-scale objects detection however still can be improved.

Accuracy scores on Mapillary and NRW validation samples

Val. Set / Metric	mAP	mAP@0.50IoU	mAP@0.75IoU	mAP (small)	mAP (medium)	mAP (large)
NRW	0.2632	0.4329	0.2857	0.0650	0.5146	0.7460
Mapillary	0.1576	0.2040	0.1694	0.0102	0.1626	0.2278

Table 10 - Mean average precision scores (COCO detection metrics) of Faster R-CNN ResNet101 with input resizing to 1024x1024 with lower scale anchors

Val. Set / Metric	AR@1	AR@10	AR@100	AR@100 (small)	AR@100 (medium)	AR@100 (large)
NRW	0.3327	0.3639	0.3653	0.1725	0.5816	0.8156
Mapillary	0.2134	0.2455	0.2901	0.0841	0.1794	0.2875

Table 11 - Average recall scores (COCO detection metrics) of Faster R-CNN ResNet101 with input resizing to 1024x1024 with lower scale anchors

As discussed before, the quality of the dataset instances is very important to make the model capable of learning the correct features to extract from the images.

Because of this, to help the learning process of the network, bad examples are filtered out from the training and validation datasets; in particular, ambiguous (not classifiable), dummy (objects similar to signs) and occluded annotated objects are ignored when generating the Tensorflow Records.

After performing such filtering, the dataset is composed by 32’879 images and a total of 108’498 annotations.

This alteration improved remarkably the accuracy of the model:

Accuracy scores on Mapillary and NRW validation samples

Val. Set / Metric	mAP	mAP@0.50IoU	mAP@0.75IoU	mAP (small)	mAP (medium)	mAP (large)
NRW	0.3070	0.4888	0.3456	0.1417	0.4932	0.8089
Mapillary	0.2012	0.2328	0.1970	0.0784	0.2053	0.2650

Table 12 - Mean average precision scores (COCO detection metrics) of Faster R-CNN ResNet101 with input resizing to 1024x1024 after filtering of bad examples

Val. Set / Metric	AR@1	AR@10	AR@100	AR@100 (small)	AR@100 (medium)	AR@100 (large)
NRW	0.4396	0.4832	0.4997	0.3260	0.6227	0.8250
Mapillary	0.2534	0.2874	0.3226	0.1209	0.2062	0.3133

Table 13 - Average recall scores (COCO detection metrics) of Faster R-CNN ResNet101 with input resizing to 1024x1024 after filtering of bad examples

Also as stated before, an additional property that the data should have is the uniformity of class distribution in order not to bias the model's predictions.

Since the analysis of the data leaked a non uniformity, especially on the "other-sign" class, a further modification is performed regarding the data, deleting the instances of such class. This decision is also justified by the fact that in this specific case, we are not interested in detecting signs of a general class such as "other-sign".

By removing such instances, the training set size decreases again, with 18'097 images and 52'785 annotations overall. The class distributions can be seen in *figure 43*.

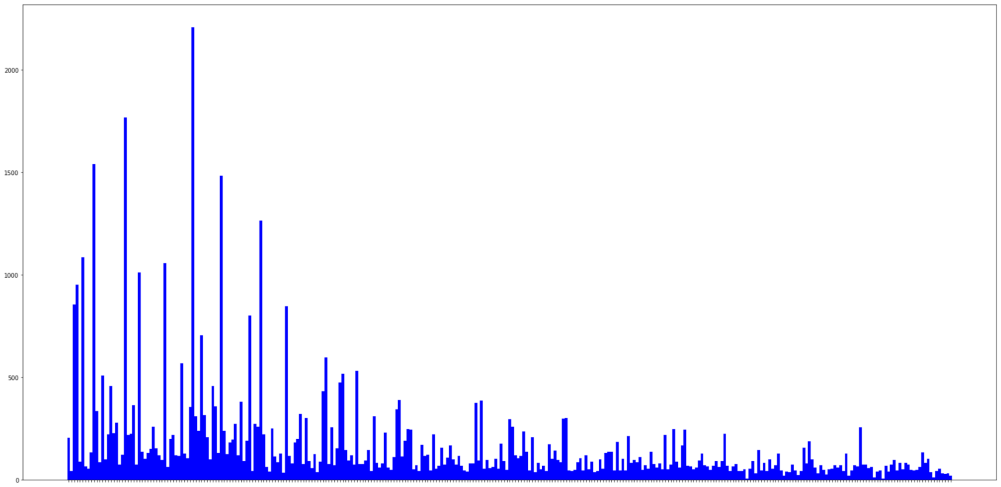


Figure 43 - traffic signs classes distribution of the Mapillary dataset after filtering both bad-property and "other-sign" instances.

The distribution is still not uniform, but is certainly better than before.

After these changes have been applied, an improvement on the performances is noticeable:

Accuracy scores on Mapillary and NRW validation samples

Val. Set / Metric	mAP	mAP@0.50IoU	mAP@0.75IoU	mAP (small)	mAP (medium)	mAP (large)
NRW	0.3175	0.5304	0.3116	0.1483	0.5283	0.8437
Mapillary	0.2114	0.2527	0.2023	0.0844	0.2219	0.2893

Table 14 - Mean average precision scores (COCO detection metrics) of Faster R-CNN ResNet101 with input resizing to 1024x1024 after filtering of "other-sign" instances

Val. Set / Metric	AR@1	AR@10	AR@100	AR@100 (small)	AR@100 (medium)	AR@100 (large)
NRW	0.3778	0.4331	0.4516	0.2719	0.5818	0.8515
Mapillary	0.2371	0.2780	0.3099	0.1191	0.1925	0.3171

Table 15 - Average recall scores (COCO detection metrics) of Faster R-CNN ResNet101 with input resizing to 1024x1024 after filtering of "other-sign" instances

However, there is still room for improvement; it is necessary to refine the network hyperparameters, precisely to the generated anchors' scales and aspect ratios. In order to improve them, a deeper analysis of the training data ground truth boxes is carried out. The distribution of the scales and aspect ratios of those bounding boxes (also considering the resizing performed by the pipeline) are clustered[24]:

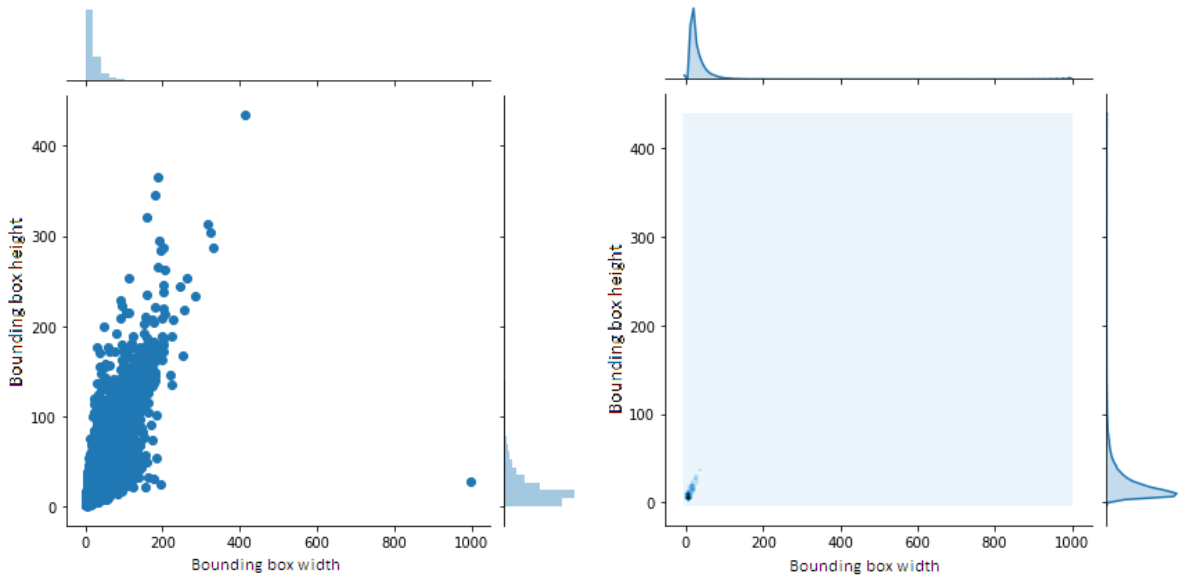


Figure 44 - Clustering of the ground truth bounding boxes of the filtered Mapillary dataset.

Afterwards, the centroids of those clusters are extracted and used as baseline for the scales and aspect ratios of the anchors, leading to this configuration:

Hyperparameter	Value
Feature extractor	ResNet101
Resizing dimension	1024
Anchors scales	[0.0482, 0.1877, 0.0881, 0.0261, 0.2907, 0.5224, 0.77, 1.54]
Anchors aspect ratios	[0.67, 0.9550, 0.9975, 0.9780, 0.9451, 1.04, 2.0]
Anchors width/height stride	16
Batch size	8
Learning rate base	0.0004
Training steps	457300
Momentum	0.9

Training hyperparameters

Applying these modifications led to obtaining the following score metrics:

Accuracy scores on Mapillary and NRW validation samples

Val. Set / Metric	mAP	mAP@0.50IoU	mAP@0.75IoU	mAP (small)	mAP (medium)	mAP (large)
NRW	0.3332	0.5270	0.3401	0.1466	0.4802	0.8294
Mapillary	0.2320	0.2753	0.2215	0.1093	0.2572	0.3029

Table 16 - Mean average precision scores (COCO detection metrics) of Faster R-CNN ResNet101 with input resizing to 1024x1024 after clustering of ground truth scales and aspect ratios

Val. Set / Metric	AR@1	AR@10	AR@100	AR@100 (small)	AR@100 (medium)	AR@100 (large)
NRW	0.4093	0.4237	0.4325	0.2379	0.5832	0.8343
Mapillary	0.2557	0.2912	0.3390	0.1339	0.2170	0.3458

Table 17 - Average recall scores (COCO detection metrics) of Faster R-CNN ResNet101 with input resizing to 1024x1024 after clustering of ground truth scales and aspect ratios

Having reached this step, to achieve better accuracy on the small-scale objects detection, a data augmentation pipeline is added to the training process, with a special attention to image cropping, so to avoid small traffic signs to vanish when resizing the image (which already has a high resolution), and instead performing a sort of zoom in by cropping it. Other augmentation options applied are random variations in brightness, contrast, saturation and conversion to grayscale. Alterations like color distortion, horizontal and vertical flipping have been carefully avoided, since they would have changed the traffic signs semantics, leading to wrong annotations and consequently a wrong learning.

```
data_augmentation_options {
    random_adjust_brightness
    random_adjust_contrast
    random_rgb_to_gray
    random_adjust_saturation
    random_image_scale

    random_crop_image  {
       min_object_covered: 1.0
       min_aspect_ratio: 1.0
       max_aspect_ratio: 1.0
       min area: 0.1
       max area: 1.0
       overlap thresh: 0.3
       clip boxes: true
       random_coef: 0.0
  }
}
```

Final version of the data augmentation pipeline.

Performing the described data augmentation during the training process results in another substantial improvement of the model accuracy and recall (using again the COCO detection metrics);

Accuracy scores on Mapillary and NRW validation samples

Val. Set / Metric	mAP	mAP@0.50IoU	mAP@0.75IoU	mAP (small)	mAP (medium)	mAP (large)
NRW	0.4056	0.6181	0.4631	0.2455	0.6046	0.8303
Mapillary	0.2970	0.3424	0.2860	0.1513	0.3629	0.4172

Table 18 - Mean average precision scores (COCO detection metrics) of Faster R-CNN ResNet101 with input resizing to 1024x1024 with data augmentation pipeline

Val. Set / Metric	AR@1	AR@10	AR@100	AR@100 (small)	AR@100 (medium)	AR@100 (large)
NRW	0.4925	0.5393	0.5541	0.4155	0.6656	0.8375
Mapillary	0.2797	0.3246	0.3738	0.1508	0.3068	0.4247

Table 19 - Average recall scores (COCO detection metrics) of Faster R-CNN ResNet101 with input resizing to 1024x1024 with data augmentation pipeline

4.1.4 SSD ResNet101 with FPN 1024

In the subsequent test, a completely different network architecture is implemented, by deploying the Single Shot Multibox Detector (SSD), still with the integration of a Feature Pyramid Network (FPN), since the simple SSD is already known to have low accuracy on small-scale objects, which is on of the main problems of the case.

The training process of the network is done using the dataset filtered in the way described above. The initial configuration is exploiting all the aspects and modifications yielded by the previous experiments, such as the default bounding boxes' (similar to the Faster R-CNN anchors) scales and aspect ratios. Also in this case the clustering of the ground truth boxes was exploited to guide the tuning of this parameter.

Such model configuration leads to the following detection scores:

Accuracy scores on Mapillary and NRW validation samples

Val. Set / Metric	mAP	mAP@0.50IoU	mAP@0.75IoU	mAP (small)	mAP (medium)	mAP (large)
NRW	0.1102	0.1461	0.1295	0.0406	0.2022	0.5270
Mapillary	0.0810	0.1187	0.1020	0.0193	0.1374	0.1853

Table 20 - Mean average precision scores (COCO detection metrics) of SSD ResNet101with feature pyramid network with input resizing to 1024x1024

Val. Set / Metric	AR@1	AR@10	AR@100	AR@100 (small)	AR@100 (medium)	AR@100 (large)
NRW	0.3370	0.3637	0.3814	0.1701	0.4859	0.7750
Mapillary	0.2089	0.2312	0.2448	0.0956	0.2262	0.2842

Table 21 - Average recall scores (COCO detection metrics) of SSD ResNet101with feature pyramid network with input resizing to 1024x1024

As can be seen, the use of this architecture produces again unsatisfying accuracy.

Further analysis of the inference detections made it possible to have a comparison with Faster R-CNN and led to the following findings:

Accuracy increases with the number of default boundary boxes at the cost of speed.

SSD has lower localization error compared to Faster R-CNN, but more classification error for similar categories; the higher classification errors are likely because it uses the same boundary box to make multiple class predictions.
SSD performs worse than Faster R-CNN for small-scale objects: in fact, SSD can detect small objects only in higher resolution layers, but since they are shallow layers, they only contain low-level features, like edges or color patches, that are less informative for classification.
Given the mentioned discoveries, the architecture was switched again to Faster R-CNN.

4.1.5 Faster R-CNN with FPN

The results of the previous experiments show that the current network architecture would not gain further improvements of the model accuracy, in this particular case. Using the 152-layer ResNet as backbone was considered, but it was then concluded that it would not have made improvements big enough to justify an experiment on it, considering his higher resource usage. After some analysis, Faster R-CNN with FPN integration seemed to be more promising.
Also to have a better comparison, again the ResNet-101 network is chosen as backbone; This choice was also due to that network being the right compromise between accuracy and complexity (number of training parameters). The latter aspect is important because of the limitations of the available hardware on the training server and to reduce training time.
In this implementation a *training from scratch* is implemented with the objective of achieving even higher accuracy. Thus, some hyperparameters are changed from the previous configuration (apart from the new ones deriving from the different architecture): the learning rate is enlarged and the batch normalization parameters are also updated during the training process. The batch size is halved to 4 because the Feature Pyramid Network architecture generates five feature maps for each image (instead of just one), that causes the network to require a lot more memory, which is instead quite limited in the training server.
The main parameters of the configuration used are shown below:

Hyperparameter	Value
Feature extractor	ResNet101 + FPN
Resizing dimension	1024
Feature Pyramid levels	2 - 6
Anchor scale	4.0
Anchors aspect ratios	[0.67, 0.9550, 0.9975, 0.9780, 0.9451, 1.04, 2.0]
Batch size	4
Learning rate base	0.04
Training steps	457300
Momentum	0.9
Data augmentation	...

Training hyperparameters

This network architecture, added up to the alterations carried on the training dataset leads to a quite satisfying accuracy. Again, mean average precisions and recalls for different scales, intersection over union (IoU) thresholds for the generated anchors and number of proposals per-image are shown below:

Accuracy scores on Mapillary and NRW validation samples

Val. Set / Metric	mAP	mAP@0.50IoU	mAP@0.75IoU	mAP (small)	mAP (medium)	mAP (large)
NRW	0.4791	0.5997	0.4872	0.2985	0.6895	0.8496
Mapillary	0.4178	0.4432	0.4250	0.1618	0.4843	0.5723

Table 22 - Mean average precision scores (COCO detection metrics) of Faster R-CNN ResNet101 with feature pyramid network, input resizing to 1024x1024 and data augmentation pipeline

Val. Set / Metric	AR@1	AR@10	AR@100	AR@100 (small)	AR@100 (medium)	AR@100 (large)
NRW	0.6537	0.7054	0.7454	0.3463	0.6564	0.8359
Mapillary	0.5462	0.5718	0.6597	0.1729	0.5116	0.6345

Table 23 - Average recall scores (COCO detection metrics) of Faster R-CNN ResNet101 with feature pyramid network, input resizing to 1024x1024 and data augmentation pipeline

4.1.6 EfficientDet

Another possible network that was considered is the modern EfficientDet, due to its high accuracy. However, such accuracy is produced by a very complex network architecture which in turn causes the speed performance to significantly drop. Furthermore, such a complex network would need much more time for the training process and the latter would also require excessively high memory resources of the GPUs on the training server, which were not available. For these reasons this possibility soon turned out to be inapplicable, so it will not be further analyzed.

4.1.7 Inference Testing of the Model on the NRW Target Domain

The final test of the model performances is computed on the target domain dataset of the region of North Rhine-Westphalia. The desired goal is to achieve a recall (detected traffic signs over the total number of ground truth signs) very close to one, which means that every sign in the dataset must be detected at least one time, of course with the correct classification.
To obtain a qualitative result on the NRW target domain, a small dataset sample has been manually annotated, using the Mapillary class taxonomy. This allowed to retrieve score metrics of mean average precision and mean average recall on it, which were shown in the previous section for each model.
Such annotated samples will also be used to generate the metrics scores to measure the efficacy of the domain adaptation process.

4.1.8 Final Network Setting

Given the dataset accuracy, the high number of classes and the importance of having correct classifications of road signs and a recall close to one rather than speed performances, Faster R-CNN with Feature Pyramid Network integration resulted to be the most appropriate for this specific case. Generally speaking, having more layers leads to a better performance in terms of accuracy, but more layers means more parameters to tune, and that also means slower training; By taking into account all these aspects and the accuracy results, the 101-layer Resnet represents the right compromise for the backbone feature extractor network. The final configuration details are shown in *section 4.1.5*.

4.1.9 Findings

One of the most notable improvements in this process of hyperparameter tuning is due to the change of the image resizing from 512x512 to 1024x1024 (*section 4.1.2 table 8* and *9*) since it enables the CNN to detect small-scale objects. The removal of objects with bad characteristics also improved the performances considerably (*section 4.1.3 table 12* and *13*). A minor contribution was given by the clusterization of ground truth bounding boxes for the generation of default anchors (*section 4.1.3 figure 44* and *table 16,17*). The final and more substantial improvement was given by the further addition of the Feature Pyramid Network within the Faster R-CNN architecture (*section 4.1.5 table 22* and *23*).

4.2 Domain Adaptation Experimentation

This section provides a comparison between the different variants of domain adaptation approaches discussed in *Chapter* 3, with the aim of disclosing which techniques and factors lead to better results in terms of adaptation for the traffic sign detection task and trying to understand the reasons behind these results.

All of the performances graphs resulting from the experiments of the subsequent sections are measured according to the COCO detection metrics (*section 3.3.7.2*); to avoid redundancy, in many cases only the $mAP^{IoU=.50:.55:.60:..95}$ (mAP averaged over 10 IoU thresholds) and mAR^{100} are shown. For efficiency reasons, the experiments are performed fine tuning a FasterRcnn with a ResNet50 as a backbone without a feature pyramid network pre-trained on the COCO2017 dataset for faster training. The final implementation is instead deployed with the FasterRcnn ResNet101 with FPN.

Unless otherwise stated, a batch size of 4 is used, containing two images of each domain; the momentum optimizer is set with a momentum value of 0.9, a warmup learning rate of .000133, a learning rate base of .0004, 5000 warmup steps and with the total steps increased to 10’000 for the reasons discussed in *section 3.4.1.3*.

If not otherwise specified, all the trainings are stopped after 730k steps, which corresponds to 40,34 epochs on the filtered Mapillary dataset and 138,12 on the DFG dataset, when using a batch size equal to one.

The other training hyperparameters are the same of the final configuration of the previous section.

4.2.1 Experiments and Results of Domain Adaptation through Self-Supervision

Since the DFG domain also contains classes specific to the Slovenian country, it represents a suitable case for domain adaptation. All the experiments provided in this section are performed with the Mapillary dataset as source domain and the DFG as target domain, at the end of the section the performances obtained in both directions with this domain adaptation approach will be compared.

Once the accuracy results on these domains are adequate, the NRW German domain can be taken as the target for the domain adaptation together with the training on the fused dataset.

In order to evaluate the efficacy of the domain adaptation on the target domain it has been necessary to create a mutual correspondence map of the DFG and Mapillary common classes in a manual manner. Domain specific classes have been kept with the original name. Such mapping has been used to create a dataset for both domains with the common classes converted to the other domain taxonomy; so to have the possibility to evaluate the mean average precision improvement on the target domain as the domain adaptation proceeds.

The experiments discussed in this section aim to disclose which are the factors that have a more important role for the success of our particular approach of domain adaptation.

Unless otherwise specified, the experiments are performed using all pretext tasks in the following order: rotation, flipping, cropping, gaussian filtering and edge classification, before the main detection task.

In many cases the experiments are performed only by adapting the CNN model from the Mapillary domain to the DFG domain, since the results are assumed to be a general evaluation of the efficacy of the process of domain adaptation through self-supervision for the traffic sign detection.

4.2.1.1 Architectures Comparison

This experiment compares the three architectures designed in *section 3.4.1.2*. The results are shown in *figure 45* and *46* and prove that the first one has a higher gain of accuracy.

The figure compares the accuracy of the architectures on the DFG dataset after performing domain adaptation on it from the Mapillary domain, the pretext tasks losses were all statically fully weighted, and the task order used is: rotation, flipping, cropping, gaussian filtering, edge classification and finally the main detection task. Such results clearly state that the first

architecture yields a higher accuracy within the domain adaptation process, in all cases apart from the small-scale objects.

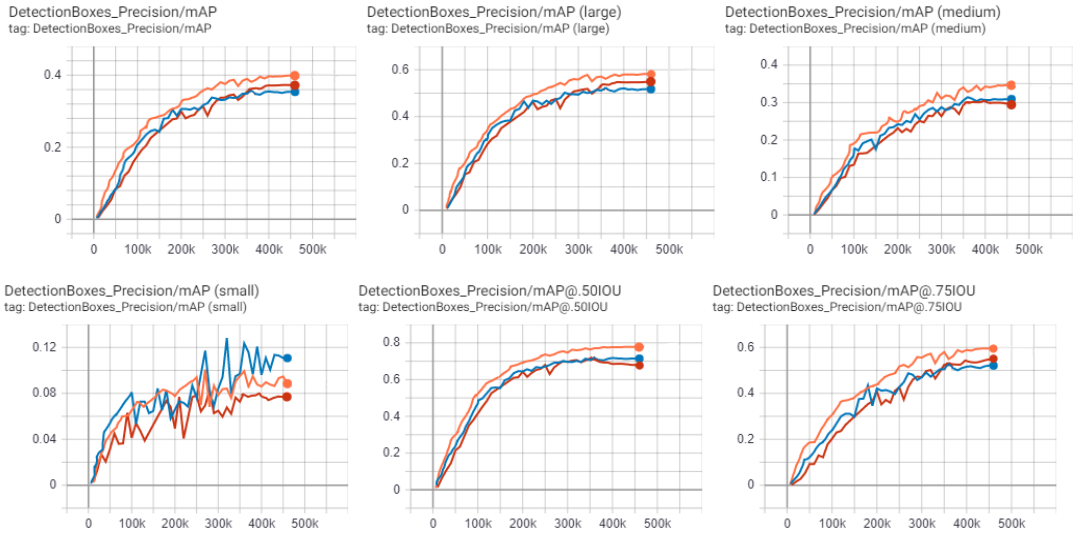


Figure 45 - Comparison of the COCO mAP scores on the DFG target domain for the three architectures after domain adaptation with self-supervision. ***Blue:*** *self supervised tasks predictions made in the second stage (second architecture),* ***Red****: self supervised tasks predictions made in the second stage and on the backbone feature maps (third architecture)-* ***Orange:*** *self supervised tasks predictions made on the backbone feature maps (first architecture). The trainings were stopped after 470k steps.*

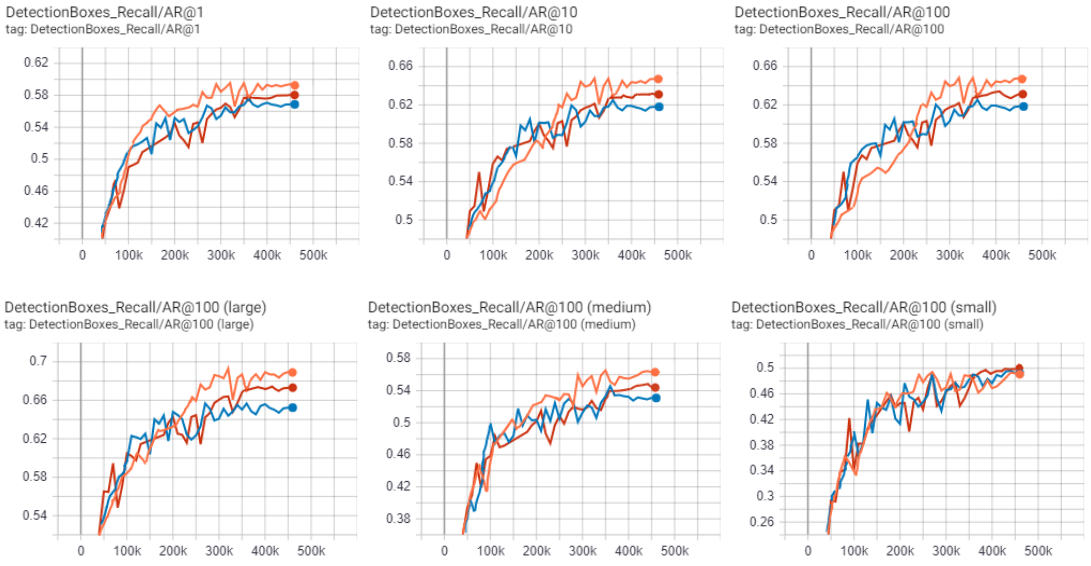


Figure 46 - Comparison of the COCO recall scores on the DFG target domain for the three architectures after domain adaptation with self-supervision. ***Blue:*** *self supervised tasks predictions made in the second stage (second architecture),* ***Red****: self supervised tasks predictions made in the second stage and on the backbone feature maps (third architecture)-* ***Orange:*** *self supervised tasks predictions made on the backbone feature maps (first architecture). The trainings were stopped after 470k steps.*

The most probable cause of this result is that by only using the features provided by the backbone for the pretext tasks, the CNN model has to focus and adjust the default features extracted by it, therefore making the feature extractor more robust to the domain variation with respect to the case where also the second stage is used.

Table 24 and *25* report the precision and recall scores respectively, using the COCO detection metrics.

Architecture / Metric	$mAP^{IoU=.50:..95}$	mAP@0.50IoU	mAP@0.75IoU	mAP (small)	mAP (med.)	mAP (large)
Standard Training	0.4906	0.8316	0.7437	0.1347	0.3921	0.6334
Backbone	0.3991	0.7857	0.5926	0.0874	0.3487	0.5872
Second Stage	0.3564	0.7208	0.5149	0.1141	0.3142	0.5185
Backbone + Second Stage	0.3796	0.6853	0.5514	0.0773	0.2897	0.5603

Table 24 - Comparison of the mean average precision scores (COCO detection metrics) on DFG after 470k steps of training with domain adaptation with the three architectures designed. The network used is the Faster R-CNN ResNet50 with input resizing to 1024x1024, with the final configuration of the previous section.

Architecture / Metric	AR@1	AR@10	AR@100	AR@100 (small)	AR@100 (med.)	AR@100 (large)
Standard Training	0.7103	0.7784	0.7995	0.5136	0.5912	0.7864
Backbone	0.5962	0.6448	0.6488	0.4931	0.5616	0.6908
Second Stage	0.5629	0.6175	0.6162	0.4976	0.5310	0.6762
Backbone + Second Stage	0.5812	0.6327	0.6324	0.5002	0.5435	0.6537

Table 25 - Comparison of the average recall scores (COCO detection metrics) on DFG after 470k steps of training with domain adaptation with the three architectures designed. The network used is the Faster R-CNN ResNet50 with input resizing to 1024x1024, with the final configuration of the previous section.

Still, the standard training yields a higher performance, suggesting that this raw application of our approach is yet not effective.

4.2.1.2 Pretext Tasks Order

Choosing the optimal ordering of the tasks to obtain the highest improvement on the target dataset is not simple as it may seem, of course blindly testing all possible order configurations is not desirable. The most influencing order is between the pretext tasks and the main detection task, therefore only this case is being experimented, whose results are shown in *figure 47*. Such experiment is carried out deploying the first architecture, since it led to the highest domain adaptation efficacy (see the previous section), the pretext task losses are all fully weighted.

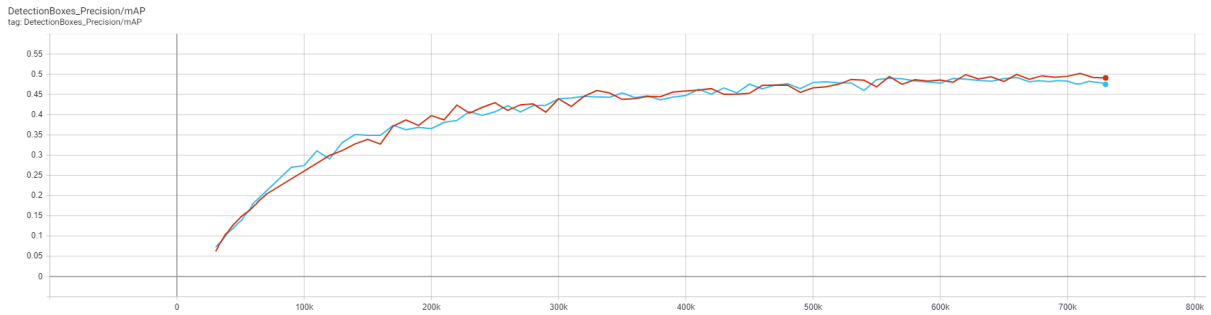


*Figure 47 - Comparison of the accuracy attained on the DFG target domain while performing domain adaptation on it. The **red** line represents the case where the detection task was the last performed. Conversely, the **azure** line depicts the case where the detection task was the first performed. In both cases the pretext tasks order was rotation, flipping, cropping, gaussian filtering and edge classification. The two trainings were stopped at step 730k*

Precision and recall values are reported in *Table 26*.

Architecture / Metric	$mAP^{IoU=.50:..95}$	AR@100
Pretext Tasks + Main Task	0.4872	0.7857
Main Task + Pretext Tasks	0.4712	0.7208

Table 26 - Mean average precision and recall scores (COCO detection metrics) on the DFG validation set after 730k steps of training with domain adaptation. The first row presents the results obtained by applying first the pretext tasks and then the main task, whereas the results of the second row are obtained by switching the order.

Performing the detection task as last leads to slightly better accuracy; the intuition behind this result is that performing the pretext tasks before it, makes the network learn to deal with the effects caused by those tasks on the itself, since at the time of performing the detection, the weights have been previously optimized for such tasks. Such effects should in fact have a

positive influence on the network behaviour, especially on the target domain, since this is the exact aim of the domain adaptation process.

However, these training hyperparameters did not improve the accuracy over the standard training on the source domain (which has a mAP of *0.4906* on DFG) but they rather made them slightly worse.

4.2.1.3 Self-Supervised Tasks Losses Weighting

As mentioned in *Section 3.4.1.3*, to better understand the magnitude of the role of the self-supervised tasks, three different weightings of their losses are experimented.

The first case is a static weighting, where all the losses of the self-supervised tasks are divided by the number of the tasks, so that the sum of their weights is equal to one, which is the weight of the main detection task. This technique makes sure that all pretext tasks together have roughly the same influence on the network as the detection, and do not predominate it, since the main task is the most important. The second case is a dynamic weighting, where all the self-supervised losses are fully weighted at the beginning, and each of their weights decays to $\frac{1}{number\ of\ pretext\ tasks}$ (weight of the previous case for each loss), with a decrease of 0.1 every 25k steps. This should yield a great update of the CNN parameters at the beginning with respect to the tasks, and then a fine-tuning process once the weights are decayed.

The final case is very similar to the previous one, but the weights decay to zero, with the same rate. This should make sure that the final steps are only focused on the main detection task, so to adjust the weights specifically for it.

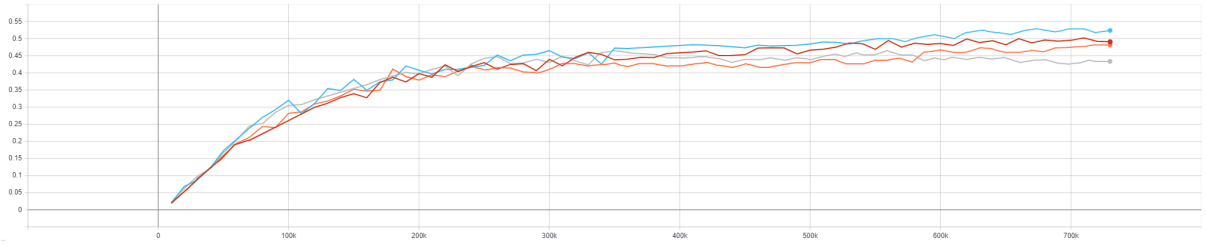


Figure 48 - Comparison of the accuracy attained on the DFG target domain while performing domain adaptation on it with different weighting of the pretext task losses. ***Red****: Standard training on the source domain (Mapillary) without domain adaptation,* ***Azure****: all weights are set to* $\frac{1}{number\ of\ pretext\ tasks}$*,* ***Orange****:The weights start from 1 and decay to* $\frac{1}{number\ of\ pretext\ tasks}$ *decreasing by 0.1 every 25k steps,* ***Gray****: The weights start from 1 and decay to 0 decreasing by 0.1 every 25k steps. The trainings were stopped after 730k steps.*

Figure 48 compares the mAP obtained on the DFG target domain with a standard training on the Mapillary source domain (without domain adaptation) to the ones obtained with domain adaptation for each different weighting of the pretext tasks during the training.
Table 27 reports precision and recall values obtained with the different weightings.

Weighting / Metric	**mAP$^{IoU=.50:..95}$**	**AR@100**
None (Standard Training)	0.4906	0.7995
$\frac{1}{number\ of\ pretext\ tasks}$	0.5302	0.8139
Decay to $\frac{1}{number\ of\ pretext\ tasks}$	0.4887	0.7971
Decay to 0	0.4365	0.7643

Table 27 - Mean average precision and recall scores (COCO detection metrics) on the DFG validation set after 730k steps of training. The standard training is compared to the trainings with domain adaptation with different weighting for the self-supervised tasks losses.

From the experiments' results it emerges that a static weighting of $\frac{1}{number\ of\ pretext\ tasks}$ for all the losses leads to a better generalization on the target domain, which surpasses the accuracy obtained with the standard training on the source domain. This is likely due to the equal balancing of the pretext task losses, which together weigh as the detection one. Therefore, the main task does not get dominated during the training so the weights remain in a good range for the detection, and at the same time the losses of the pretext tasks make the network learn to better generalize on the target domain.
Having an initial weight of one is instead likely to induce a distortion in the network parameters that inevitably lose accuracy on the main task which is not recovered.
Eventually, also the case of weight decayment to zero likely causes the network to get specialized on the detection task, but not to learn to properly generalize on the target domain.
The case of a static full weighting of all pretext tasks losses is not repeated since the previous test has shown that it does not lead to a significant accuracy improvement on the target domain, which is likely caused by the pretext tasks dominating the main one

The most important aspect about the results of this experiment is that they show the success of the technique of domain adaptation through self-supervision implemented, in terms of better generalization and accuracy on the target domain, although in a moderate amount.

4.2.1.4 Traffic Signs Crops for Pretext Tasks

As mentioned in *Section 3.4.1.3*, a variant of this domain adaptation approach has been tested, which consists in using traffic signs crops for the pretext tasks instead of the whole images. This experiment was performed to investigate whether making the network focus directly on the traffic signs features for the pretext tasks could lead to obtain a further accuracy improvement. *Figure 49* again compares the mAP on the DFG target domain with a standard training on the Mapillary source domain (red line) to the domain adaptation using entire images (azure line) and using traffic signs crops (orange line). In both cases the weights were again set to $\frac{1}{number\ of\ pretext\ tasks}$. Given the higher amount of samples for the pretext tasks, in this case a batch size of 8 is used (containing four images for each domain); for a valid comparison, the previous standard training and domain adaptation using entire images are repeated as well with the new batch size.

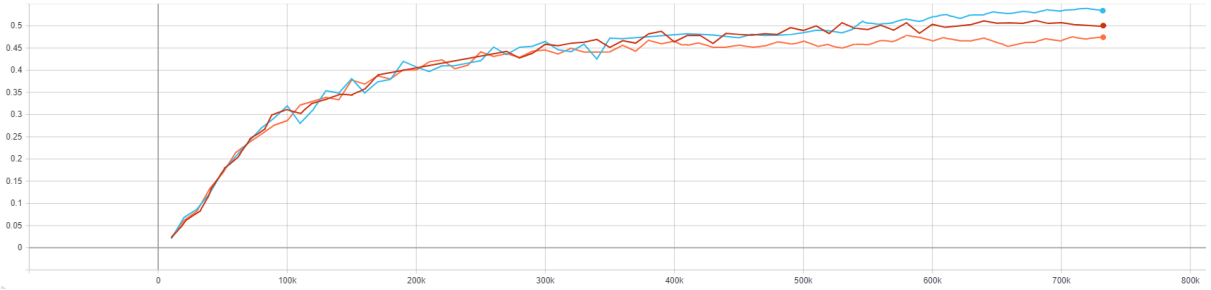


Figure 49 - Comparison of the accuracy attained on the DFG target domain. ***Red****: Standard training on the source domain (Mapillary) without domain adaptation,* ***Azure****: entire images are used, all weights are set to* $\frac{1}{number\ of\ pretext\ tasks}$, ***Orange****: images crops are used for the self-supervised tasks on both domains, all weights are set to* $\frac{1}{number\ of\ pretext\ tasks}$. *The trainings were stopped after 730k steps.*

The results demonstrate that making the network focus on the traffic signs features during the self-supervised tasks does not yield any improvement, in fact it instead leads to worse accuracy with respect to using the entire images. The reason behind this result is most likely caused by the CNN model better learning features relating to the context of a typical structure of the target domain scenes when entire images are used for the pretext tasks. Given the outcome, this approach will not be further tested on the NRW domain. The results also show that a bigger batch size is slightly better in terms of mAP and AR.

Pretext Tasks Input / Metric	$mAP^{IoU=.50:..95}$	AR@100
None (Standard Training)	0.4996	0.8014
Entire Images	0.5361	0.7922
Traffic Signs Crops	0.4326	0.8035

Table 28 - Mean average precision and recall scores (COCO detection metrics) on the DFG validation set after 730k steps of training. The standard training is compared to the training with domain adaptation using entire images or crops for the self-supervised tasks.

4.2.1.5 Adaptation from Mapillary to DFG vs Adaptation from DFG to Mapillary

Comparing the accuracy gained changing the direction of the domain adaptation should clarify how much the dataset sizes of both the source and target domain affect the efficacy of the domain adaptation process. For limited resources reasons, the batch size was lowered to 2 on this occasion, with one image for each domain.

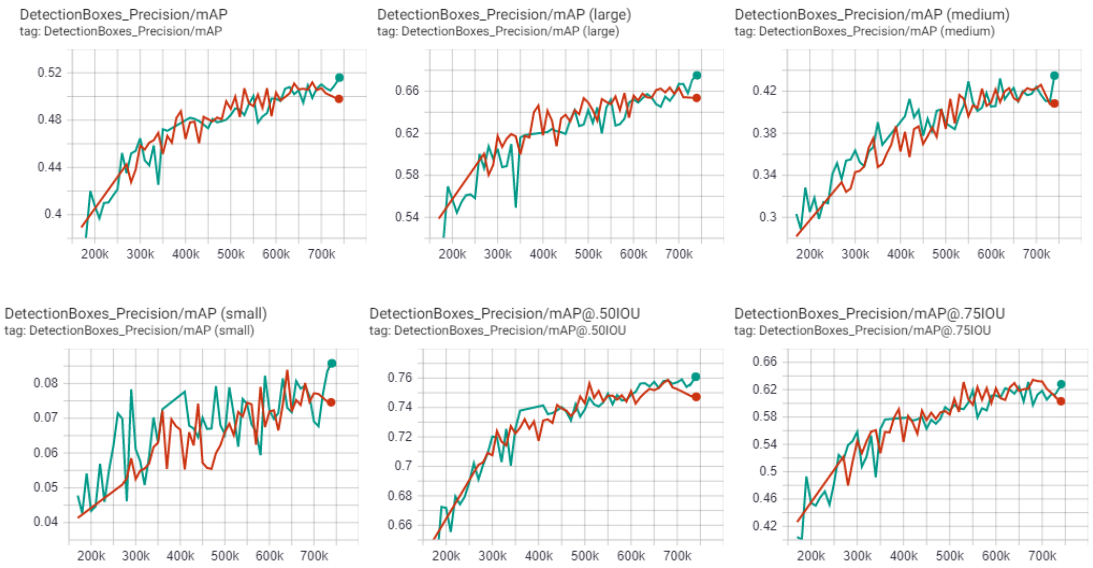


Figure 50 - Comparison of the mAP (COCO detection metrics) on the DFG validation split after 730k steps of training. ***Red:*** *simple training on the Mapillary dataset.* ***Green****: domain adaptation from Mapillary to DFG.*

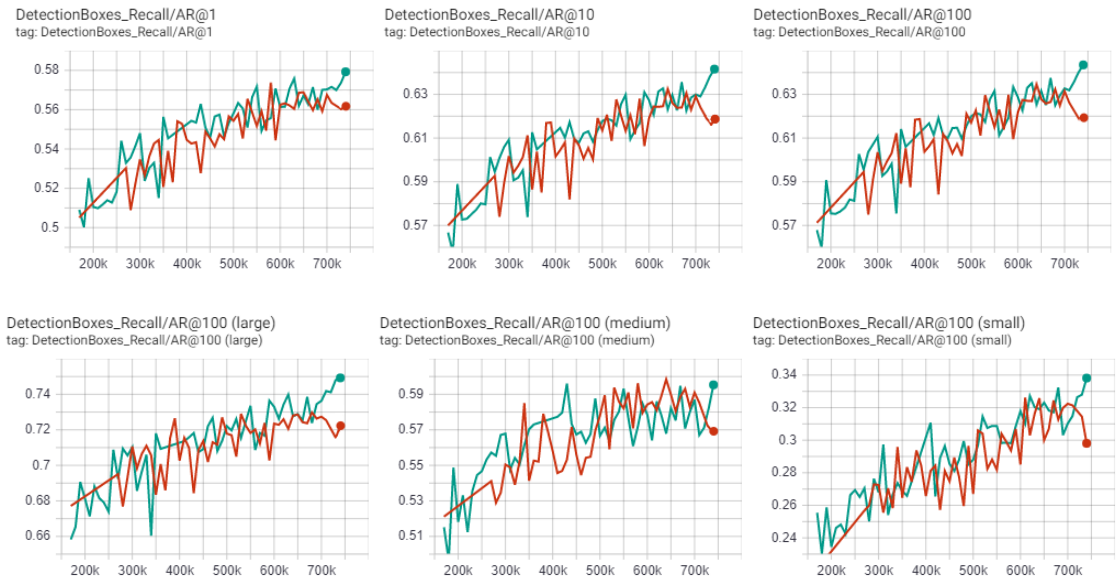


Figure 51 - Comparison of the AR (COCO detection metrics) on the DFG validation split after 730k steps of training. ***Red:*** *simple training on the Mapillary dataset. -* ***Green****: domain adaptation from Mapillary to DFG.*

Training Type / Metric	$mAP^{IoU=.50:..95}$	mAP@0.50IoU	mAP@0.75IoU	mAP (small)	mAP (med.)	mAP (large)
Standard	0.4983	0.7479	0.6062	0.0746	0.4087	0.6541
Domain Adaptation	0.5179	0.7623	0.6297	0.0822	0.4375	0.6774

Table 29 - Comparison of the mean average precision scores (COCO detection metrics) on the DFG validation split after 730k steps of standard training on Mapillary and training with domain adaptation on DFG. The network used is the Faster R-CNN ResNet50 with input resizing to 1024x1024, with the final configuration of the previous section.

Training Type / Metric	AR@1	AR@10	AR@100	AR@100 (small)	AR@100 (med.)	AR@100 (large)
Standard	0.5618	0.6189	0.6197	0.2971	0.5693	0.7231
Domain Adaptation	0.5795	0.6420	0.6437	0.3381	0.5964	0.7494

Table 30 - Comparison of the average recall scores (COCO detection metrics) on the DFG validation split after 730k steps of standard training on Mapillary and training with domain adaptation on DFG. The network used is the Faster R-CNN ResNet50 with input resizing to 1024x1024, with the final configuration of the previous section.

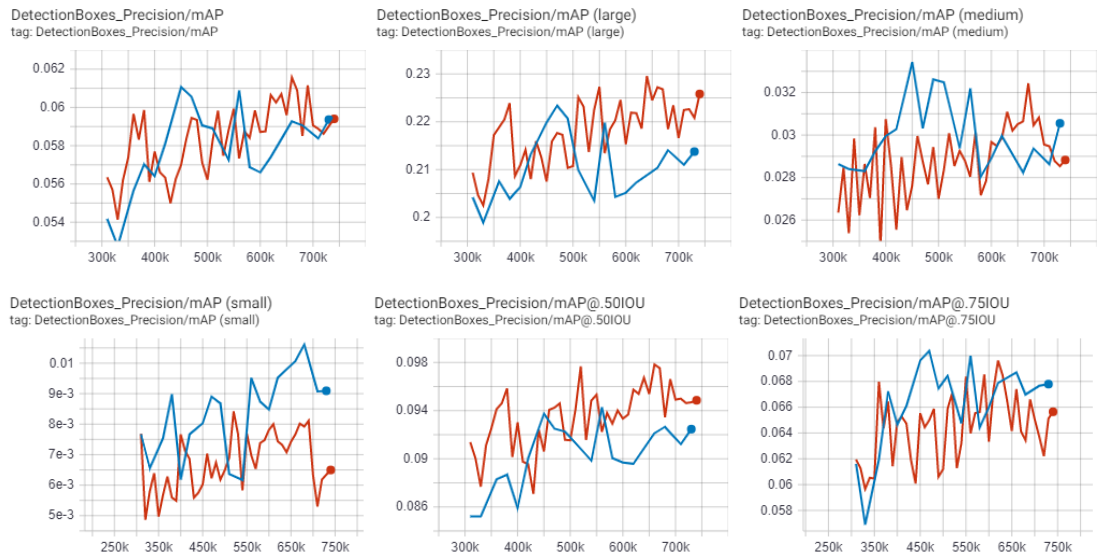


Figure 52 - Comparison of the mAP (COCO detection metrics) on the Mapillary validation split after 730k steps of training. ***Red:*** *simple training on the DFG dataset. -* ***Blue****: domain adaptation from DFG to Mapillary.*

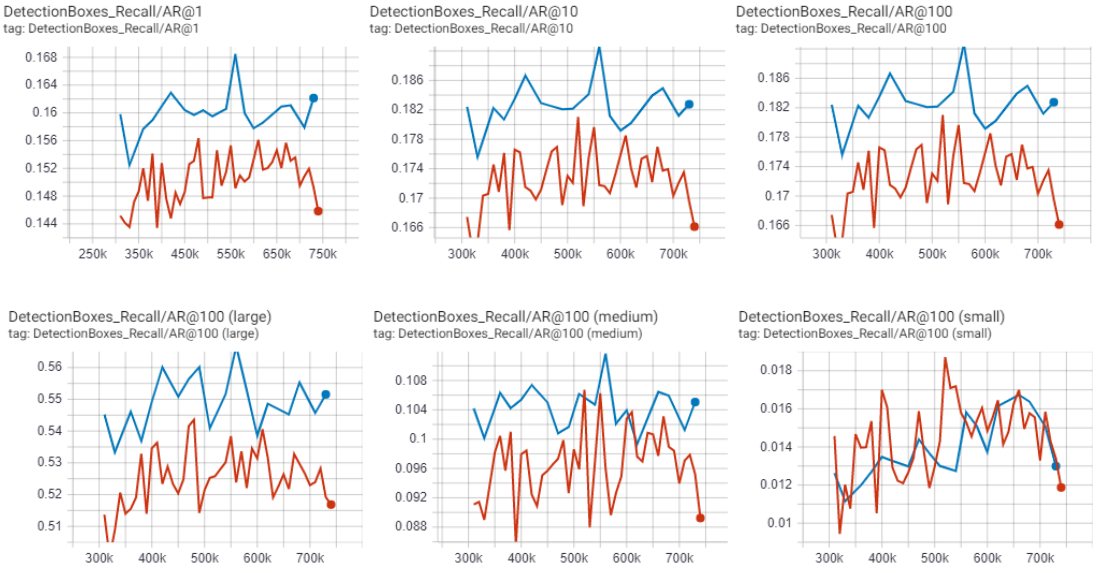


Figure 53 - Comparison of the AR (COCO detection metrics) on the Mapillary validation split after 730k steps of training. ***Red:*** *simple training on the DFG dataset. -* ***Blue****: domain adaptation from DFG to Mapillary.*

As can be seen, the adaptation from the Mapillary domain to the DFG domain yields a more noticeable improvement on the recall rather than the precision. Nevertheless, it is a good result since for the specific pseudo-labeling final task, the CNN model recall is more important, since it is better to have false positives rather than false negatives within the pseudo-labels; false negatives in the training data in fact generally lead to an higher weakening of the model's performance with respect to false positives. In addition, the

adaptation process does not have a considerably negative effect on the accuracy on the source domain, in fact the $mAP^{IoU=.50:..95}$ decreases by -1.01% and the AR@100 by -0.84%.

Training Type / Metric	$mAP^{IoU=.50:..95}$	mAP@0.50IoU	mAP@0.75IoU	mAP (small)	mAP (med.)	mAP (large)
Standard	0.0594	0.0949	0.0658	6.5e-3	0.0298	0.2256
Domain Adaptation	0.0593	0.0926	0.0679	9.2e-3	0.0377	0.2184

Table 31 - Comparison of the mean average precision scores (COCO detection metrics) on the Mapillary validation split after 730k steps of standard training on DFG and training with domain adaptation on Mapillary. The network used is the Faster R-CNN ResNet50 with input resizing to 1024x1024, with the final configuration of the previous section.

Training Type / Metric	AR@1	AR@10	AR@100	AR@100 (small)	AR@100 (med.)	AR@100 (large)
Standard	0.1457	0.1661	0.1662	0.0117	0.0894	0.5157
Domain Adaptation	0.1621	0.1828	0.1829	0.0130	0.1057	0.5518

Table 32 - Comparison of the average recall scores (COCO detection metrics) on the Mapillary validation split after 730k steps of standard training on DFG and training with domain adaptation on Mapillary. The network used is the Faster R-CNN ResNet50 with input resizing to 1024x1024, with the final configuration of the previous section.

The domain adaptation is effective even from the DFG domain to the Mapillary one, again mostly for the recall. The overall accuracy is quite low, however this is caused by the more complex characteristics of the Mapillary dataset, mostly due to the different devices and resolutions. Again, the accuracy on the source domain has only a slight decrease, with a decrease of -2.07% for the $mAP^{IoU=.50:..95}$ and of -2,23% for the AR@100.

Given the data and network modeling that has been necessary to obtain a good accuracy on that domain, it is clear that such task is not trivial, and it was in fact the aim of the experimentations of *section 4.1*. For this reason, the network of *section 4.1.8* will be used for the final implementation.

4.2.1.6 Feature-Space Distance

Figure 54 shows the mean feature-space distance defined in *section 3.4.1.4* during the domain adaptation from the Mapillary domain to the DFG domain using the first architecture, with the losses of the pretext tasks statically weighted to $\frac{1}{number\ of\ pretext\ tasks}$. The task order used

is rotation, flipping, cropping, gaussian filtering, edge classification and finally the main task. For such tasks the entire images are used.

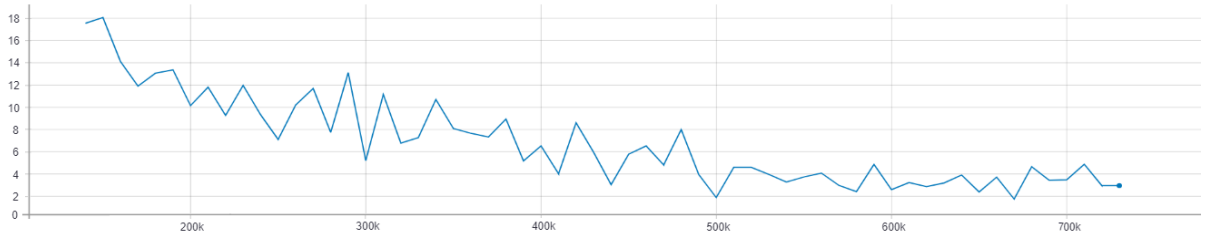


Figure 54 - Mean feature distance between images of the two domains while performing domain adaptation from Mapillary to DFG; as the domain adaptation proceeds the distance in the shared feature space becomes smaller.

As was desirable, the feature-space distance becomes smaller while the training proceeds, indicating that the feature representations of the images of the two domains are getting closer as the training proceeds, therefore making the network better at generalizing in the target domain, since the features of samples of the two domains become similar.

Figure 55 shows the mean feature distance during the adaptation from DFG to Mapillary, even in this case the distance gets lower as the training proceeds.

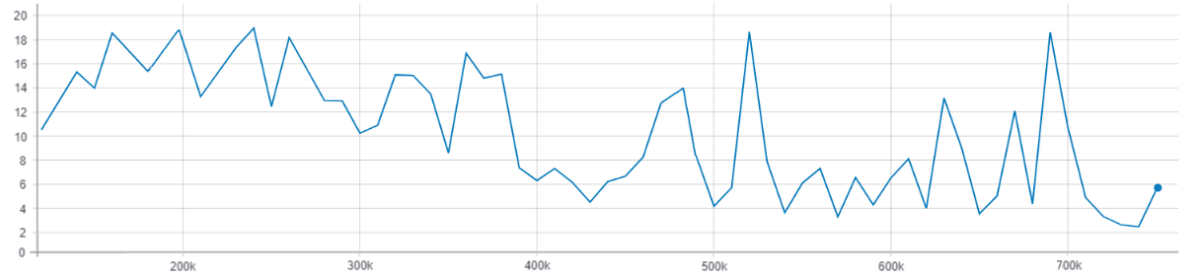


Figure 55 - Mean feature distance between images of the two domains while performing domain adaptation from DFG to Mapillary; as the domain adaptation proceeds the distance in the shared feature space becomes smaller.

4.2.1.7 Domain Adaptation on Domain-Specific Classes

This variant was the last tested, since it can not be evaluated in terms of generalization on the target domain. The evaluation is therefore performed on the images of the source domain containing at least one of its specific classes, while for the generalization on the target domain, it is assumed that the results of the previous experiments are valid even for this case. It is desirable that the performances on the source domain specific classes remain roughly the same, while the domain adaptation should improve the accuracy on the target domain.

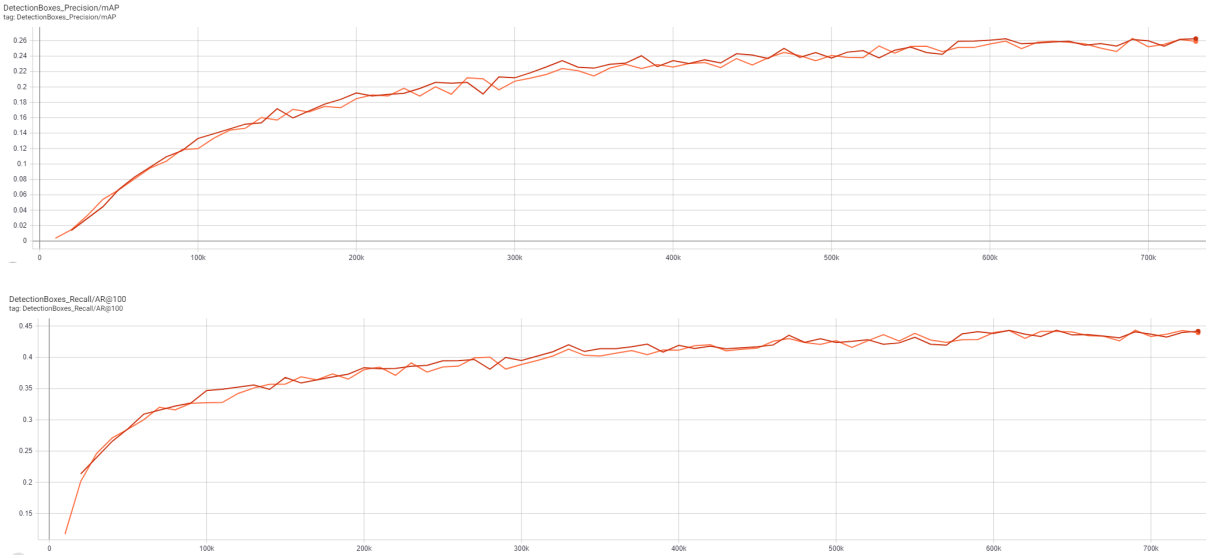


*Figure 56 - **Upper**: mean average precision (*$\text{mAP}^{\text{IoU=.50:..95}}$*) curve on the set of Mapillary specific classes of the model standardly trained on it (**red**) and the model trained with domain adaptation on DFG with the same set as source domain (**orange**). **Lower**: average recall (*AR@100*) curve on the Mapillary specific classes of the model standardly trained on them (**red**) and the model trained with domain adaptation on DFG (**orange**). In both cases the trainings were stopped after 730k steps.*

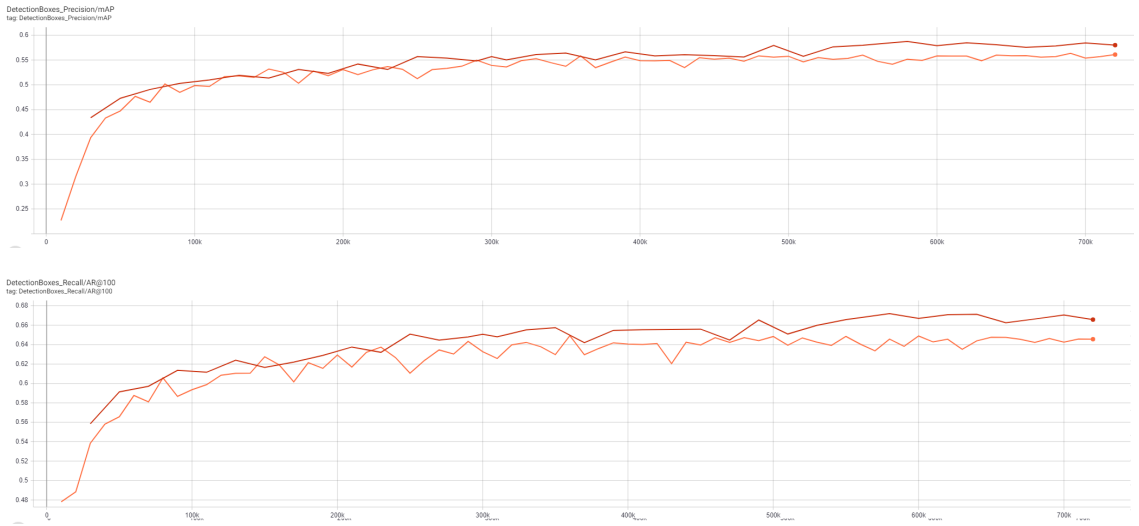


*Figure 57 - **Upper**: mean average precision (*$\text{mAP}^{\text{IoU=.50:..95}}$*) curve on the set of DFG specific classes of the model standardly trained on it (**red**) and the model trained with domain adaptation on Mapillary with the same set as source domain (**orange**). **Lower**: average recall (*AR@100*) curve on the DFG specific classes of the model standardly trained on them (**red**) and the model trained with domain adaptation on Mapillary (**orange**). In both cases the trainings were stopped after 730k steps.*

The results show that while performing the adaptation to DFG the accuracy on the source domain specific classes remains almost the same as well. In the other direction, such accuracy

is slightly lower, however the adaptation should improve the performance on the target domain, which is for obvious reasons more important.

These results also demonstrate that the approach of domain adaptation with self-supervision adopted does not deteriorate significantly the accuracy on the source domain while adapting to the target domain.

Training Type / Metric	$mAP^{IoU=.50:..95}$	AR@100
Standard	0.2625	0.4415
Domain Adaptation	0.2591	0.4391

Table 33 - Final mean average precision (mAPIoU=.50:..95) and average recall (AR@100) on the validation set of Mapillary specific classes of a model with standard training on the training set of Mapillary specific classes and the model trained with domain adaptation on DFG with the same set as source domain. The trainings were stopped after 730k steps.

Training Type / Metric	$mAP^{IoU=.50:..95}$	AR@100
Standard	0.5803	0.6657
Domain Adaptation	0.5610	0.6456

Table 34 - Final mean average precision ($mAP^{IoU=.50:..95}$) and average recall (AR@100) on the validation set of DFG specific classes of a model with standard training on the training set of DFG specific classes and the model trained with domain adaptation on Mapillary with the same set as source domain. The trainings were stopped after 730k steps.

4.2.2 Influence of Domain Adaptation on Pseudo-Labeling Accuracy

In order to perform the pseudo-labeling task, again the mapping from/to DFG to/from Mapillary is used. Only the training splits are used for the pseudo-labeling.

The subsequent experiments have been executed following the indications and the pseudo-metrics defined in *Section 3.4.2* and they should help to understand whether our domain adaptation approach (and its variants previously described) are suitable both for the case of traffic signs detection and for the specific domains used, since the datasets also play a significant role for the success of the adaptation.

To select a good confidence threshold for the pseudo-labels filtering, the mean precision and recall are calculated for each class for a set of confidence thresholds, in particular for 10 thresholds: starting from 0.0 to 0.9 with a step of 0.1.

Subsequently, the mean precision and recall are calculated over all classes for each threshold, these values are then used to generate a single precision-recall curve, which graphically displays the mean overall precision and recall for each threshold. The curves have been computed with a CNN model trained with domain adaptation on the target domain, using only specific classes of the source domain; the precision and recall values for each class have been measured on the validation split of the source domain containing only specific classes.
figure 58 and *59* show such curves for the Mapillary and DFG specific classes respectively.

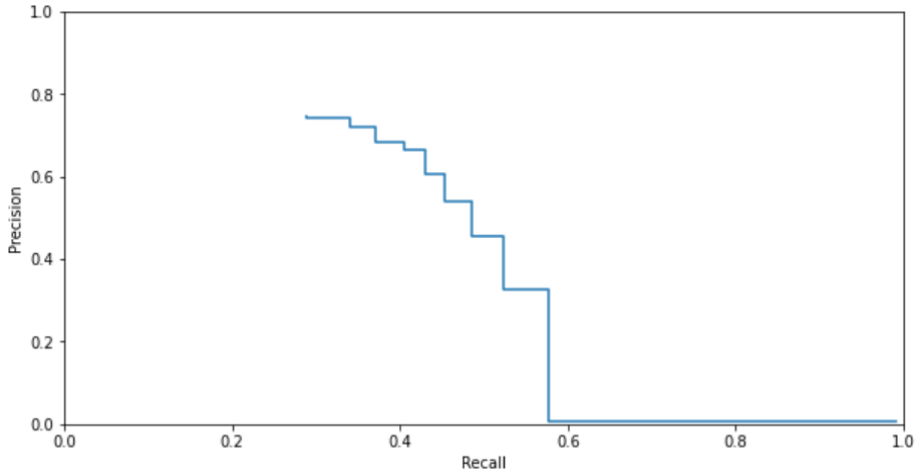


Figure 58 - Mean precision-recall curve over the validation set of Mapillary specific classes after 730k steps of training with domain adaptation on DFG.

Given the precision-recall trade-off, a threshold of 0.4 is selected when generating labels of Mapillary classes instances, since it represents a good trade-off between precision and recall, slightly favoring the recall, which is more important for the aforementioned reasons. In fact, such a threshold yields a mean precision of 0.504 and a mean recall of 0.553.

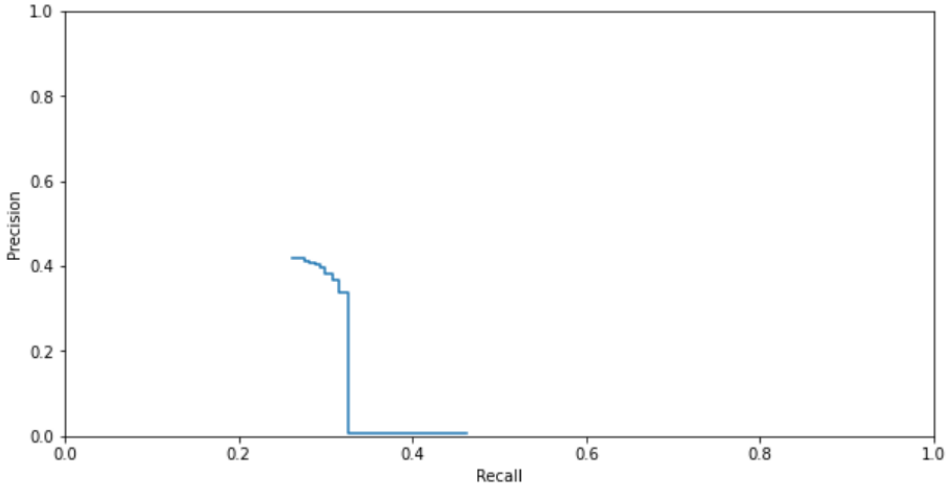


Figure 59 - Mean precision-recall curve over the validation set of DFG specific classes after 730k steps of training with domain adaptation on Mapillary.

For the generation of DFG classes labels, a threshold of 0.3 is used, that yields an average precision of 0.39 and a recall of 0.32.

4.2.2.1 Pseudo-Labeling Accuracy Without Domain Adaptation

Pseudo-Labeling of the DFG Domain

The DFG domain has originally 200 classes and a total of 5254 images in the training split.

Metric	Value
Overlapping-Concordance	0.7775
Mean Confidence Score	0.7247
Injected Annotations	3142
Injected Classes	204

Table 35 - Performance of the pseudo-labeling process on DFG without domain adaptation based on the defined pseudo-metrics, number of new annotations and classes injected in the DFG domain.

Pseudo-Labeling of the Mapillary Domain

The Mapillary domain has originally 313 classes and a total of 36589 images in the training split.

Metric	Value
Overlapping-Concordance	0.4924
Mean Confidence Score	0.6483
Injected Annotations	7190
Injected Classes	146

Table 36 - Performance of the pseudo-labeling process on Mapillary without domain adaptation based on the defined pseudo-metrics, number of new annotations and classes injected in the Mapillary domain.

Even from this result, it is noticeable the better quality of the Mapillary dataset, causing it to be more prone to a better generalization to other domains, even without a process of domain adaptation. As previously mentioned, this is caused by its “generic” characteristics, which are

formed by the images coming from different countries and taken with different camera devices.

4.2.2.2 Pseudo-Labeling Accuracy with Domain Adaptation

Besides the comparison, this specific case is also used to perform the marking of difficult images in both domains, as described in *section 3.4.2.1.2*.

Pseudo-Labeling of the DFG Domain

By comparing the results of *table 35* and *table 37* it can be seen that the domain adaptation improves considerably the accuracy of the pseudo-labels along with the accuracy on the target domain. The *Overlapping-Concordance* has an improvement of 4.6% by applying the domain adaptation on the DFG domain. The effectiveness is also proved by the slight increase of the confidence score.

By filtering the difficult images, a total of 4648 out of 5254 images were annotated. The injected classes and annotations are less with respect to the standard training, but this is assumed to be caused both by the dataset filtering and by the better accuracy on the target domain, which is likely to further remove bad annotations.

Metric	Value
Number of images marked as difficult	606/5254
Overlapping-Concordance	0.8241
Mean Confidence Score	0.7994
Injected Annotations	2741
Injected Classes	196

Table 37 - Performance of the pseudo-labeling process on DFG after applying the domain adaptation based on the defined pseudo-metrics, number of new annotations and classes injected in the DFG domain.

Pseudo-Labeling of the Mapillary Domain

Metric	Value
Number of images marked as difficult	2985/36589
Overlapping-Concordance	0.5279
Mean Confidence Score	0.6937
Injected Annotations	5957
Injected Classes	121

Table 38 - Performance of the pseudo-labeling process on Mapillary after applying the domain adaptation based on the defined pseudo-metrics, number of new annotations and classes injected in the Mapillary domain.

Again, the accuracy of the pseudo-labels has increased after applying the domain adaptation through self-supervision, the *Overlapping-Concordance* has an improvement of 3.5%.

4.2.2.3 Pseudo-Labeling Accuracy with Domain Adaptation using Source Domain Specific Classes

For the reasons discussed in *section 3.4.2*, in this instance only the source domain specific classes are being inferred in the target domain, since this is the final objective of the pseudo-labeling. Therefore, in this case it is not possible to compute the *Overlapping-Concordance*.

The dataset of Mapillary specific classes has 9’149 images, 26’688 annotations for a total of 225 classes over the whole 313, while the DFG has 2’429 images, 4’539 annotations and a total of 105 classes over the whole 200.

For the mentioned reasons the labels generated in this instance are the actual used to generate the fused dataset.

Pseudo-Labeling of the DFG Domain

Given the previous filtering, 4648 images of the DFG domain are annotated in this phase.

Metric	Value
Mean Confidence Score	0.7916
Injected Annotations	2335
Injected Mapillary Specific Classes	157

Table 39 - Performance of the pseudo-labeling process on DFG after applying the domain adaptation using only samples containing specific classes for the source domain.

By comparing *table 37* and *39*, it is clearly visible that the exclusive training on specific classes resulted in a significant improvement of the pseudo-labeling process, which translates in more pseudo ground truth annotations and classes being inserted. The latter is in fact a sign of better specific classes learning and adaptation.

With the new annotations, the DFG domain has a total of 12'150 labels, and a set of 357 classes.

Pseudo-Labeling of the Mapillary Domain

Given the previous filtering, 33604 images of the Mapillary domain are annotated in this phase.

Metric	Value
Injected Annotations	5739
Mean Confidence Score	0.6874
Injected DFG Specific Classes	94

Table 40 - Performance of the pseudo-labeling process on Mapillary after applying the domain adaptation using only samples containing specific classes for the source domain.

The number of new annotations is instead lower with respect to the case of using all source domain classes. The likely cause of this is the slight accuracy loss, which is also shown in *figure 57*, however, the pseudo-labels should be more accurate. With the new annotations, the Mapillary domain has a total of 415 classes.

4.2.3 Findings

The experiments presented in this section proved, among the other aspects, that in the context of the Faster R-CNN architecture predicting the self-supervised tasks solely on the feature representation produced by the backbone network leads to a better adaptation to the target domain (*section 4.2.1.1*). Furthermore, it is shown that performing pretext tasks first and then the main prediction task order produces a better generalization to the target domain (*section 4.2.1.2*).

Experiments have also shown that weighting the pretext tasks losses with a static coefficient of $\frac{1}{number\ of\ pretext\ tasks}$ also yields a better adaptation (*section 4.2.1.3*).

In addition, we discovered that using entire images as input data for the self-supervised tasks is better than using crops of the traffic signs (*section 4.2.1.4*).

Also, the efficacy of our adaptation is more noticeable from Mapillary domain to the DFG one (*section 4.2.1.5*).

Eventually, one of the most important findings is that our domain adaptation (both when using the entire source domain and when using only specific classes of it) improves the pseudo-labeling accuracy (section *4.2.2.1 4.2.2.2 4.2.2.3*).

4.3 Multi-Domain Training Influence

The experiments presented in this section aim to clarify whether training a CNN model on the fused dataset leads to better performances on a given domain, with respect to a model that is trained exclusively on it. The accuracy of such model is tested both on the domains that form the fused dataset (i.e. Mapillary and DFG) and on the NRW domain.

For the Mapillary and DFG domains, the accuracy on their specific classes is also compared to a CNN model trained solely on them, since the injected pseudo-labels are of specific classes, this yields more information about the quality of such annotations.

Since the fused dataset is quite bigger and includes much more classes, it is unlikely that a network trained on the fused dataset achieves a higher accuracy with respect to the models trained directly on the specific classes, however achieving a similar performance would indicate that the pseudo-labels' accuracy is good as well. Instead, using the fused dataset as training data should lead to better performances on the actual target domain of NRW.

The generated fused dataset has 419 different classes, comprising the common classes and the specific ones of each domain; the training set is composed of 30'419 images and the validation set of 5'249, a total of 8'074 pseudo-annotations are distributed in the whole fused dataset. All metrics have been measured after 900k steps of training with a batch size of 4 on the fused dataset and of 1 on the other datasets, which corresponds to ~55,3 epochs on Mapillary (same epochs on the specific dataset), ~171,2 epochs on DFG (same epochs on the specific dataset) and ~118,3 epochs on the fused dataset.

4.3.1 Multi-Domain Training Influence on Performances on Mapillary

Table 41 and *42* shows the accuracy of a CNN model trained on the fused dataset, obtained on both the whole set of Mapillary classes and on the set of its specific classes. Such measurements are compared to the accuracy of a CNN model trained on the Mapillary dataset only, on the two same sets of validation data.

Training Set Used / Metric	$mAP^{IoU=.50:..95}$	AR@100
Mapillary (All Classes)	0.250	0.439
Fused Dataset	0.226	0.411

Table 41 - Comparison of $mAP^{IoU=.50:..95}$ and AR@100 on the standard validation set of Mapillary.

Training Set Used / Metric	$mAP^{IoU=.50:..95}$	AR@100
Mapillary (All Classes)	0.260	0.438
Mapillary (Specific Classes)	0.262	0.442
Fused Dataset	0.234	0.415

Table 42 - Comparison of $mAP^{IoU=.50:..95}$ and AR@100 on the validation set of Mapillary containing only specific classes.

The results show that the training on the fused dataset yields a similar performance, primarily for the specific classes. Since the pseudo-labels are generated only for the domain-specific classes, this is an indicator that their quality is good, therefore that the pseudo-labeling process has a fine accuracy and no bias is introduced through the pseudo-labeling.

4.3.2 Multi-Domain Training Influence on Performances on DFG

Table 43 and *44* shows the same cases of performance comparisons of the previous section on the DFG domain.

Training Set Used / Metric	$mAP^{IoU=.50:..95}$	AR@100
DFG (All Classes)	0.580	0.652
Fused Dataset	0.502	0.604

Table 43 - Comparison of $mAP^{IoU=.50:..95}$ and AR@100 on the standard validation set of DFG.

Training Set Used / Metric	$mAP^{IoU=.50:..95}$	AR@100
DFG (All Classes)	0.589	0.667
DFG (Specific Classes)	0.586	0.673
Fused Dataset	0.507	0.605

Table 44 - Comparison of $mAP^{IoU=.50:..95}$ and AR@100 on the validation set of DFG containing only specific classes.

As foreseeable, the accuracy on the smaller DFG dataset has a greater worsening, this is likely caused by the lower accuracy of the DFG pseudo-labels injected in the Mapillary domain, as can be seen also from the pseudo-metrics. This result is understandable given that the DFG dataset is significantly smaller than the Mapillary one, therefore the CNN model has less data to learn from.

4.3.3 Multi-Domain Training Influence on Performances on NRW

Since the NRW domain does not have any ground truth, to evaluate the performance on it, another process of labeling is first applied. In particular, considering that the focus of interest is on the specific classes of the two domains, a small sample of images containing at least one instance of those classes (besides the common classes) is extracted from the whole dataset and inserted in the validation set. To extract the images, two CNN models with feature pyramid networks are used to add pseudo-annotations on such data, one trained on DFG and the other

on Mapillary; The annotations are written in Pascal VOC format and are manually adjusted to remove the errors made by the predictors. Eventually, three validation sets are created, one containing DFG classes, one containing Mapillary classes and the last one containing classes of the fused dataset (i.e. from both domains).

The validation sets resulting from this process are used to evaluate and compare the performance of the networks trained on different datasets, so to determine whether the fused dataset produces better or similar performances on the NRW domain when used as training data. A similar accuracy would already be a good result, since the model would detect classes of both domains in the NRW dataset, yielding more information on such data.

In addition to the standard training case, a process of domain adaptation with self-supervision on NRW is also experimented, since no ground truth is necessary its implementation is straightforward. The self-supervised tasks are not performed on all the NRW domain, but on a sample of 10'602 images that contains a wide variety of different scenery, from urban to rural areas.

Training Set Used / Metric	**$mAP^{IoU=.50:..95}$ (DFG)**	**AR@100 (DFG)**	**$mAP^{IoU=.50:..95}$ (Mapillary)**	**AR@100 (Mapillary)**	**$mAP^{IoU=.50:..95}$ (Fused)**	**AR@100 (Fused)**
DFG	0.350	0.503	/	/	0.350	0.503
DFG + Domain Adaptation on NRW	0.423	0.548	/	/	0.423	0.548
Mapillary	/	/	0.526	0.652	0.526	0.652
Mapillary + Domain Adaptation on NRW	/	/	0.478	0.611	0.478	0.611
Fused Dataset	0.431	0.588	0.486	0.608	0.437	0.584
Fused Dataset + Domain Adaptation on NRW	0.410	0.573	0.452	0.606	0.418	0.582

Table 45 - Comparison of $mAP^{IoU=.50:..95}$ and AR@100 on the different validation sets of the NRW domain of CNN models trained with the different training sets.

For a better comparison, the validation sets for all domains are generated using the same set of 63 images, the DFG set with 298 annotations, the Mapillary with 246 annotations and the fused with 358 annotations. Results of all test instances are shown in *table 45*.

An interesting result is that using the fused dataset as training data leads to better performances on the DFG classes even with respect to the model specifically trained on the corresponding dataset. The most likely cause of this is the lower quality of the DFG classes, which are less complete and more confusing than the ones of Mapillary; therefore a CNN model that only learns from them has, as a result, weaker and more confused features.

4.4 Pseudo-Labeling and Fine-Tuning on the NRW Target Domain

To further improve the accuracy on the NRW domain, pseudo-labels are generated on the entire dataset and subsequently used for fine-tuning purposes. Such annotations are generated using a CNN model trained on the fused dataset while performing domain adaptation on the NRW samples.

Due to the lack of groundtruth, it is impossible to use the *Overlapping-Concordance* during the process of pseudo-labeling. In addition, the *self-supervised-accuracy* (see *section 3.4.2.1.2* for the definition) is calculated for each image; afterwards, the mean accuracy of the self-supervised tasks is calculated over the whole NRW dataset. The result is then used as an additional pseudo-metric, shown in the results with the name of *mean self-supervised accuracy.* The significance of this metric is exactly the core of the domain adaptation through self-supervision: if during the adaptation to the NRW domain the network has learned to predict the self-supervised tasks accurately, the performance on the main detection task should have improved as well; therefore an high value should be an indirect indicator of the detection accuracy.

The number of annotations inferred and of the total classes added are again used as a sort of measure of the adaptation.

4.4.1 Pseudo-Labeling Accuracy on the NRW Dataset with Multi-Domain Training and Domain Adaptation

The CNN model is trained on the fused dataset with domain adaptation on NRW for ~63,1 epochs on the fused dataset and ~181 on the NRW samples.

Metric	Value
Mean Self-Supervised Accuracy	0.9343
Mean Confidence Score	0.8022
Injected Annotations	98789
Injected Classes	264

Table 46 - Metrics of the pseudo-labeling process on the NRW samples

The fine-tuning dataset contains all 31’238 NRW samples, with a total of 98’789 annotations. However, since this dataset does not contain all the classes of the fused dataset, performing a fine-tuning directly on it would cause the model to “unlearn” the classes that are not present in it. Therefore we do not perform a “pure” fine-tuning, the annotated samples are instead merged to the fused dataset, so to extend it and to obtain a training set containing every class of interest. The fine-tuning dataset is made of 61’657 images with 169’463 annotations.

4.4.2 Performance Changes with the NRW-Fused Dataset Training

The CNN model is trained on the fine-tuning dataset for around 30,2 epochs.

Metric	Value
$mAP^{IoU=.50:..95}$	0.383
AR@100	0.538

Table 47 - $mAP^{IoU=.50:..95}$ and AR@100 obtained on the fused NRW validation set when using the fused dataset and NRW pseudo-annotated samples as training data

As the results show, the fine-tuning using the additional pseudo-annotations on the NRW domain leads to a lower mAP and AR compared to the case of simple training on the fused dataset. The most probable cause of this are the errors that are certainly present in those pseudo-annotations, despite the high metrics regarding the annotation process (*table 46*).
The result is however still informative since it shows that a process of fine-tuning with labels that are not perfect as the ones created by humans is ineffective and rather noxious for the performances of a convolutional neural network. This also confirms how the semi-supervised learning through pseudo-labeling is still not effective in many cases, and indeed it is an active research field.

5. CONCLUSIONS

The major aim of the thesis was to develop a convolutional neural network that achieves an high detection recall and precision on the NRW region. This is not a trivial problem since when the domain at inference time is different from the domain of training time, the performances inevitably get worse.

The results show that our technique of domain adaptation through self supervision along with the training on a dataset composed by the fusion of different domains is effective against the change of domain of inference. In addition, the work shows that the domain adaptation is functional also to improve the quality of a pseudo-labeling process and introduces some techniques of evaluation of those annotations in the context of semi-supervised learning.

Since the problem of domain adaptation is not considerably covered yet, this work can be a useful contribution to this topic.

Eventually it is shown that processes of semi-supervised learning through pseudo-labeling are still not much effective in complex cases like this, which is most likely owing to the non-perfection of the predicted annotations, that cause the performances of a CNN to decrease.

Another minor contribution is about the techniques for the mutual adjustment between the training dataset and a CNN detection model, in particular for the two-stage object detection architectures.

5.1 Future Work

The particular technique of domain adaptation adopted is effective and at the same time very flexible and configurable, where the addition of a new pretext task is straightforward; therefore new structural tasks could be designed and implemented, along with the experimentation of different variations of the training with domain adaptation, like different batch sizes, orders or weightings for the tasks in order to find the optimal hyperparameters configuration in terms of domain adaptation.

Moreover, additional pseudo-metrics for the self-predicted annotations can be designed, so as to retrieve a deeper quality analysis of the cross-domain fusion in a single dataset.

REFERENCES

[1] Ian Goodfellow, Yoshua Bengio and Aaron Courville, "Deep Learning", *MIT Press*, 2016

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren and Jian Sun, "Deep Residual Learning for Image Recognition", *Microsoft Research*, 2015

[3] Karen Simonyan and Andrew Zisserman, "Very deep convolutional networks for large-scale image recognition", *Visual Geometry Group, Department of Engineering Science*, University of Oxford, 2015

[4] Wei Li , Kai Liu, Lizhe Zhang and Fei Cheng, "Object detection based on an adaptive attention mechanism", Scientific Reports, *natureresearch*, 2020

[5] Shaoqing Ren, Kaiming He, Ross Girshick and Jian Sun, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks", 2016

[6] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu and Alexander C. Berg, "SSD: Single Shot MultiBox Detector", *Google Inc.*, University of Michigan, Ann-Arbor, 2016

[7] Tsung-Yi Lin, Piotr Dollar, Ross Girshick, Kaiming He, Bharath Hariharan and Serge Belongie, "Feature Pyramid Networks for Object Detection", *Facebook AI Research (FAIR)*, Cornell University and Cornell Tech, 2017

[8] Jonathan Huang, Vivek Rathod, Chen Sun, Menglong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer, Zbigniew Wojna, Yang Song, Sergio Guadarrama and Kevin Murphy, "Speed/accuracy trade-offs for modern convolutional object detectors", *Google Research*, 2017

[9] Wanli Ouyang, Xiaogang Wang, Cong Zhang and Xiaokang Yang, "Factors in Finetuning Deep Model for Object Detection with Long-tail Distribution", The Chinese University of Hong Kong, 2016

[10] Christian Ertler, Jerneja Mislej, Tobias Ollmann, Lorenzo Porzi, Gerhard Neuhold and Yubin Kuang, "The Mapillary Traffic Sign Dataset for Detection and Classification on a Global Scale", 2020

[11] Mingxing Tan, Ruoming Pang and Quoc V. Le, “EfficientDet: Scalable and Efficient Object Detection”, *Google Research,* 2020

[12] Yaroslav Ganin, Evgeniya Ustinova, Hana Ajakan, Pascal Germain, Hugo Larochelle, Francois Laviolette, Mario Marchand and Victor Lempitsky, “Domain-Adversarial Training of Neural Networks”, 2016

[13] Muhammad Ghifary, W. Bastiaan Kleijn, Mengjie Zhang, David Balduzzi and Wen Li, “Deep Reconstruction-Classification Networks for Unsupervised Domain Adaptation”, *Victoria University of Wellington*, 2016

[14] Sicheng Zhao, Xiangyu Yue, Bo Li, Han Zhao, Bichen Wu, Ravi Krishna, Joseph E. Gonzalez, Alberto L. Sangiovanni-Vincentelli, Sanjit A. Seshia and Kurt Keutzer, “A Review of Single-Source Deep Unsupervised Visual Domain Adaptation”, *IEEE, 2020*

[15] Spyros Gidaris, Praveer Singh and Nikos Komodakis, “Unsupervised Representation Learning By Predicting Image Rotations”, *LIGM Ecole des Ponts ParisTech*, University Paris-Est, 2018

[16] Yu Sun, Eric Tzeng, Trevor Darrell and Alexei A. Efros , “Unsupervised Domain Adaptation Through Self-Supervision”, University of California, Berkeley, 2019

[17] Dan Hendrycks, Mantas Mazeika, Saurav Kadavath and Dawn Song, “Using Self-Supervised Learning Can Improve Model Robustness and Uncertainty”, *33rd Conference on Neural Information Processing Systems (NeurIPS)*, Vancouver, Canada, 2019

[18] Fabio M Carlucci, Antonio D’Innocente, Silvia Bucci, Barbara Caputo, and Tatiana Tommasi, “Domain generalization by solving jigsaw puzzles”, *IEEE Conference on Computer Vision and Pattern Recognition*, 2019

[19] Yuhua Chen, Wen Li, Christos Sakaridis, Dengxin Dai and Luc Van Gool, “Domain Adaptive Faster R-CNN for Object Detection in the Wild”, *Computer Vision Lab, ETH Zurich*, 2018

[20] Yaroslav Ganin and Victor Lempitsky, “Unsupervised Domain Adaptation by Backpropagation”, Skolkovo Institute of Science and Technology (Skoltech), Moscow, 2015

[21] French G., Mackiewicz M., Fisher, M., “Self-ensembling for visual domain adaptation”, 2017

[22] Eric Arazo, Diego Ortego, Paul Albert, Noel E. O’Connor and Kevin McGuinness, “Pseudo-Labeling and Confirmation Bias in Deep Semi-Supervised Learning”, *Insight Centre for Data Analytics, Dublin City University (DCU)*, 2020

[23] TensorFlow, *https://www.tensorflow.org*

[24] TensorFlow Object Detection API by TF Object Detection Team, retrieved from *https://github.com/tensorflow/models/tree/master/research/object_detection*

[25] Faster-RCNN Anchor Boxes Clusterization using K-Means, based on *https://github.com/joydeepmedhi/Anchor-Boxes-with-KMeans*