Master thesis : Is the use of synthetic datasets a solution to improve object detection models on real data?

Auteur : Tilkin, Nicolas
Promoteur(s) : Louppe, Gilles
Faculté : Faculté des Sciences appliquées
Diplôme : Master : ingénieur civil en science des données, à finalité spécialisée
Année académique : 2022-2023
URI/URL : http://hdl.handle.net/2268.2/16762

University of Liège
Faculty of Applied sciences

Is the use of synthetic datasets a solution to improve object detection models on real data?

Master's thesis submitted in partial fulfillment of the requirements for the degree of *Master of Science in Data Science and Engineering*

by

Nicolas Tilkin

Supervisors

Prof. Gilles Louppe
Loïc Lejoly

Academic year 2022–2023

Contents

Abstract

In the recent years, object detection models have leveraged deep learning architectures to improve performance in many problems. However, these techniques require a large amount of high quality labelled data in order to reach their full potential, and obtaining such data may prove to be an arduous task. In this context, this work explores the possibility of using entirely synthetically generated and labelled images to train an object detection model. In particular, we examine which factors of variations in the synthetic data best transfer to real data. Unsurprisingly, models trained on synthetic data only perform significantly worse than models trained on real data. We explore whether the synthetic images can be enhanced using filtering and generative models, but find the results to be inconclusive. In a setting where both real and synthetic data are available, we experiment to find out how these should be combined to improve performance in the real domain. We find that the synthetic and real datasets should be combined into a single training dataset, and that the object detection model trained in this fashion significantly outperforms the model trained on real data only.

Acknowledgements

I would first like to thank Pr. Gilles Louppe and Loïc Lejoly for their guidance, feedback and understanding during this endeavour.

I also would like to express my appreciation of NRB's data science & artificial intelligence team for their warm welcome and benevolence.

Finally, I would like to signify my gratitude to my family and friends who have provided me with their unconditional support throughout my degree.

Chapter 1

Introduction

1.1 Context

In the past, computer vision relied on successive handcrafted filters to process images in order to identify human understandable components such as edges using the Sobel filter [1], corners using the Harris corner detector [2] and blobs[1] using methods such as MSER [3] and FAST [4]. In the same spirit, more complex algorithms were also designed to extract sets of features from images, for instance in order to use them as descriptors for matching pair of points between objects taken in different poses and scales such as in SIFT [5] and in its later, more robust version SURF[6].

Building upon techniques that output features from an image, it was possible to use them as the inputs for traditional machine learning models. A straightforward approach to object detection consisted in sliding a variable size window across an image and computing features using a chosen feature extractor. These could subsequently be fed to a traditional classifier which had been preemptively trained to discriminate between feature vectors of different objects, or lack thereof.

While these approaches had demonstrated successful results in the variety of problems they were trying to solve, they suffered from a critical drawback : the developed methods were *ad hoc* procedures which did not allow for effortless reusability. This showed to be problematic as research carried out when tackling a new problem required timely and costly development from particularly knowledgeable field experts, making it hardly accessible and perhaps even unaffordable. Moreover, maintaining a reliance on handcrafted methods would inevitably be a limiting factor if super-human performance was sought, as the computed features would arise from what humans thought of as important in the different components of images. In this context, a more flexible, sustainable and ideally learnable approach to computer vision would be highly valuable.

In parallel, a specific branch of machine learning dedicated to the study and design of neural networks had been increasingly gaining traction among the research community. While the foundations had been set long ago by F. Rosenblatt who designed the hierarchical composition of neural units [7] to form neural networks in an effort to imitate the human brain's reaction to visual stimuli [8], it took sev-

[1]In image processing, blobs are sets of visually consistent and spatially connected pixels.

eral decades for deep learning to grow into the buzzing field it is today. Reasons for the recent flourishing of deep learning include the availability of computational resources and adapted hardware, open-source libraries providing complete modules with enough abstraction to avoid getting lost in the intricacies of deep learning while still allowing fine-grained control, and combined efforts which have made available many quality datasets for numerous tasks, simplifying the benchmarking of new proposed network architectures and training procedures.

In the recent years, research in deep learning has been steadily growing and has yielded state-of-the-art performance in various fields such as natural language processing and computer vision. While part of these achievements can be attributed to the ingenious design of new architectural blocks such as convolutional and attention layers that introduce appropriate inductive biases for the target task, it is undeniable that the surge in model size and depth has also largely contributed to these successes. However, this is not the only explaining factor of the recent successes of deep learning. Indeed, neural networks and large models especially severely depend on having access to large datasets. Roughly speaking, it is likely that a simpler model trained with enough data will perform better than a larger model trained with insufficient data, although defining exactly what is and isn't enough data is a challenging task that often is case-specific. In practice, the relationship between model capacity[2] and dataset size is difficult to establish, but it has been empirically observed that to benefit the most from an increase in capacity, the amount of training data should also increase, and conversely.

In 2017, Sun *et al.* [9] observed that while the available computational power and the capacity of computer vision models had consistently been increasing, the size of available datasets had remained fairly constant and argued that part of the focus in research should shift towards data collection. Their findings pointed out the importance of having access to large quantities of relevant data to train deep learning models. However, data collection is often a time-consuming and costly process, especially in a supervised learning setting where the data must be carefully labelled by humans, with the incurred risk of mislabelling. The costs and difficulties associated to data collection may in practice be strong deterrents for companies and organisations to launch deep learning projects. For these reasons, approaches to overcome or somewhat bypass data hungriness could lower barriers to entry and allow more participants to compete on a level playing field by giving them access to deep learning methods.

1.2 Problem statement

The present work aims to tackle the issue of data hungriness through the use of synthetic data, in the context of the object detection task. A loose definition in the context of images could consider common image augmentations such as flipping, cropping, blurring and shifting the colour palette to be synthetic data. These transformations have clearly become part of the standard training procedure as they have demonstrated improvements to the models' generalisation capabilities, but they are

[2]The capacity of a model refers to how complex it can become, and thus its representation capability.

ultimately distortions of available data and thus need data to be indeed used. While they are certainly interesting and it is entirely possible additional augmentations could be engineered and help in the learning process, these light-weight syntheses won't be studied in this work.

Rather, the focus here will be to create data from the ground up. For this purpose, we will make use of three dimensional rendering tools which allow to generate scenes to include foreground objects to be detected, as well as different background layouts. These scenes can then be captured to produce images alongside the localisation information of the foreground objects which have been included. The resulting data is therefore suited to train an object detection model. This work will analyse how synthetic data can be used as a substitute or alongside real data. In particular, the following questions will be considered :

- What is the performance loss observed when training a model purely on synthetic data instead of real data? Which factors of variations in the synthetic data help towards achieving better performance in the real domain?
- Apart from visual inspection, can we characterise what distinguishes synthetic images from real ones?
- Can further enhancements be made to the synthetic data such that it is better suited to train models evaluated on real data?
- Is it interesting to mix both synthetic and real data when training a model, and how should it be done?

The goal here is to examine the effectiveness of synthetic data, but not to get the absolute best performance by scaling model size as much as possible. Rather, the approach taken is to attempt different configurations and procedures in order to understand what works best and how synthetic data should be crafted and used.

Chapter 2

Background

In this chapter, some concepts are explained to ensure that the relevant theoretical background is introduced. Machine learning basics are promptly recalled in Section 2.1, and deep learning essentials are presented in Section 2.2. In particular, we briefly present the general structure and training procedure and detail relevant tweaks which are used in this work. In Section 2.3, some characteristics of images and deep learning building blocks which leverage them are explained. Then, object detection and underlying computer vision tasks are described from a deep learning perspective in Section 2.4. Finally, some deep object detection models are described in Section 2.5, with a particular attention to the one used in this work.

2.1 Machine learning

In the recent years, data in many shapes and forms has been generated and recorded at an ever increasing pace. This massive availability calls for automated techniques that can extract value from data by learning from it. These techniques are broadly regrouped under the term *machine learning*, which can be defined as "*a set of methods that can automatically detect patterns in data, and then use the uncovered patterns to predict future data*" [10]. As such, machine learning methods are designed to be able to learn a model from examples. If these examples are pairs of inputs and outputs, supervised learning methods are used to learn a mapping between them. If there are no such pairs, unsupervised learning methods can be exploited to unveil patterns in the data and attempt to learn the distribution that generated it. Both of these paradigms will be found in this work : object detection is a supervised learning task where the inputs are images and the outputs are labels indicating class and localisation in the image, and unsupervised learning will be used through generative models when attempting to revamp synthetic images into ones which more closely resemble real images.

2.1.1 Supervised learning

In all generality and employing the formalism found in [11], the settings of supervised learning entail access to a dataset of finite size N consisting of inputs $\mathbf{x}_i \in \mathcal{X}$ and outputs $\mathbf{y}_i \in \mathcal{Y}$, $i = 1, ..., N$, where the nature of the sets $\mathcal{X}$ and $\mathcal{Y}$ depends on the particular learning task. Each of these samples is generated i.i.d[1] according to

[1]Independently and identically distributed

an unknown distribution $P(X, Y)$.

The goal of supervised learning is then to learn a function $f : \mathcal{X} \rightarrow \mathcal{Y}$, often called the model, in order to perform inference on unseen inputs. The model is obtained through a learning algorithm by constraining the set of functions f that can be produced to an hypothesis space $\mathcal{F}$, and defining a error measure l often called loss. The loss is a function $l : \mathcal{Y} \times \mathcal{Y} \rightarrow \mathbb{R}_+$ which allows to measure how close a prediction $f(\mathbf{x})$ and corresponding ground truth $\mathbf{y}$ are. The loss of a perfect prediction is thus 0, and increases as the prediction becomes worse.

The best possible function f_* in $\mathcal{F}$ would therefore ideally minimise the expected risk over all possible unseen data, called the generalisation error :

$$f_* = \arg\min_{f \in \mathcal{F}} R(f) \tag{2.1}$$

where

$$R(f) = \mathbb{E}_{(\mathbf{x},\mathbf{y}) \sim P(X,Y)} \left[l(\mathbf{y}, f(\mathbf{x})) \right]$$

However, given that only a finite size dataset is available, we resort to empirical risk minimisation by minimising the average risk over the training data in order to obtain an approximation $\hat{f}_*$:

$$\hat{f}_* = \arg\min_{f \in \mathcal{F}} \hat{R}(f) \tag{2.2}$$

where

$$\hat{R}(f) = \frac{1}{N} \sum_{i=1}^{N} l(\mathbf{y}_i, f(\mathbf{x}_i))$$

It must be mentioned that aiming for the lowest possible empirical risk during the learning process is however not recommended. Indeed, it is very likely that this would lead to overfitting, especially if the training data is scarce and not representative of the entire population. Overfitting occurs when the learned model is too fit to the training data, such that its generalisation ability is diminished. Several strategies exist to prevent this phenomenon from arising. In this work, we will hold out a fraction of the data as a validation set, and use it to evaluate the model at each learning step in order to select the model that would supposedly fare best when confronted to unseen data.

2.1.2 Unsupervised learning

In the case of unsupervised learning, the data provided is not structured into inputs and outputs. Many different families of problems exist depending on the desired goal, such as clustering, dimensionality reduction, density estimation and learning to draw samples from a distribution [12], making it difficult to adopt a common formalism for all problems.

2.2 Deep learning

For many applications, basic models such as linear or logistic regressions which heavily restrict the representation capabilities are too constraining, and a more complex hypothesis space is required. Loosely inspired by the way information is processed in the brain, the idea of neural networks is to compose layers of basic units called neurons in a hierarchical manner and connect them through tunable weights. Hence, deep learning is a framework in which the internal structure appropriate for the task is found by composing multiple representation levels. These are learnt by repeatedly adjusting the model's parameters through correcting them proportionally to their contribution to the error made on training examples. In this section, we cover deep learning fundamentals.

2.2.1 Neuron

The elementary building block of any neural network is the neuron unit. A single neuron can be viewed as a simple linear model followed by a non-linearity. It transforms an input vector $\mathbf{x} \in \mathbb{R}^p$ into an output $o \in \mathbb{R}$ through

$$o = g(\mathbf{w}^T\mathbf{x} + b) \tag{2.3}$$

with weights $\mathbf{w} \in \mathbb{R}^p$ and bias $b \in \mathbb{R}$ the parameters of the neuron, and $g(\cdot)$ a non-linear function also called activation function.

Activation functions

The purpose of activation functions is to introduce non-linearities into neural network models. Over the years, many different functions have been proposed to alleviate the shortcomings of previous ones. In the past, commonly used functions, shown in Figure A.1, included the sigmoid

$$\sigma(x) = \frac{1}{1 + \mathrm{e}^{-x}} \tag{2.4}$$

and the hyperbolic tangent

$$\tanh(x) = \frac{\mathrm{e}^{x} - \mathrm{e}^{-x}}{\mathrm{e}^{x} + \mathrm{e}^{-\mathrm{x}}} \tag{2.5}$$

These however suffered a critical issue, which is that the saturation they exhibit towards their extremities imply that their derivative shrinks to 0 as $|x| \to +\infty$. Since neural networks use gradient based optimisation to find their parameter set, this proved to be problematic to build deep networks as vanishing gradients eventually blocked the optimisation process.

To overcome this problem and driven by the will to shift models towards sparse activations which are more biologically plausible, a new activation function called Rectified Linear Unit was proposed by Glorot *et al.* [13] :

$$\mathrm{ReLU}(x) = \max(0, x) \tag{2.6}$$

Its derivative being 0 for $x \leq 0$ (it is actually undefined at $x = 0$ but set to 0 in practice), and 1 for $x > 0$, it indeed both induced sparsity and solved the vanishing gradient issue.

Since then, many other activation functions have been proposed, often motivated by better empirical performance in particular tasks or for particular models. In this work, we pay a special attention to the Sigmoid Linear Unit, also called the swish activation, because it is used in our object detection model. It was successively suggested by Elfwing *et al.* [14] in the context of reinforcement learning, followed by Ramachadran *et al.* [15] who showed it matched or outperformed ReLU across a variety of tasks and datasets in deep networks. It is defined as

$$\mathrm{SiLU}(x) = x \cdot \sigma(x) \tag{2.7}$$

and, similarly to ReLU, is bounded below and unbounded above. Unlike ReLU, it is non-monotonic and non-zero for negative inputs, which Ramachadran *et al.* claim to improve gradient flow. Their graphs are comparatively shown in Figure 2.1.

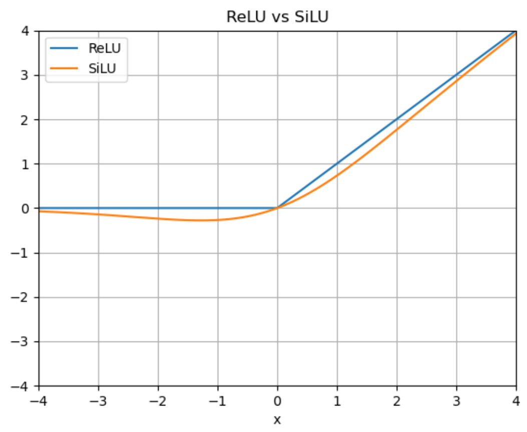


Figure 2.1: ReLU(x) and SiLU(x)

2.2.2 Layers and the multi-layer perceptron

Stacking neurons in parallel forms what is referred to as a layer. Following the formalism found in [16], the output $\mathbf{o} \in \mathbb{R}^q$ of a layer with q neurons is

$$\mathbf{o} = g(\mathbf{W}^T\mathbf{x} + \mathbf{b}) \tag{2.8}$$

with $\mathbf{W} \in \mathbb{R}^{p\times q}$ and $\mathbf{b} \in \mathbb{R}^q$, and g is a non-linear activation function applied element-wise. Composing L of these layers in series, the input can be successively processed as follows :

$$\begin{aligned} \mathbf{h}_0 &= \mathbf{x} \\ \mathbf{h}_1 &= g(\mathbf{W}_1^T\mathbf{h}_0 + \mathbf{b}_1) \\ &\dots \\ \mathbf{h}_L &= g(\mathbf{W}_L^T\mathbf{h}_{L-1} + \mathbf{b}_L) \\ f(\mathbf{x};\boldsymbol{\theta}) = \hat{\mathbf{y}} &= \mathbf{h}_L \end{aligned} \tag{2.9}$$

where $f(\mathbf{x};\boldsymbol{\theta})$ denotes the output of the model parametrised by
$\boldsymbol{\theta} = \{\mathbf{W}_k, \mathbf{b}_k, ... \mid k = 1, ..., L\}$.

This gives rise to the multi-layer perceptron model, also referred to as the fully connected feedforward network. This is the starting point of deep learning models, and can be enhanced with more advanced architectural designs adapted to the model's purpose.

2.2.3 Training

Optimisation

In deep learning, the optimisation process which intends to find the parameter set deemed best for the target task is often referred to as the training step. It consists in finding the parameters $\boldsymbol{\theta}^*$ of the model $f(\cdot\,; \boldsymbol{\theta})$ which minimise the total loss $\mathcal{L}(\boldsymbol{\theta})$ over the training data :

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N} l(\mathbf{y}_i, f(\mathbf{x}_i; \boldsymbol{\theta})) \tag{2.10}$$

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) \tag{2.11}$$

where $l(\cdot\,, \cdot)$ is some loss function.

In general, a closed-form analytical solution to the minimisation problem is not available, and we must resort to numerical optimisation techniques. In deep learning, the method of gradient descent is used and can be applied to any differentiable loss function. It is worth noting that a neural network may be seen as a composition of functions, and as such, the partial derivatives of the loss with respect to each parameter can be computed using the chain rule. Starting from a parameter set $\boldsymbol{\theta}_0$, gradient descent iteratively updates the parameter values using

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \gamma \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_t) \tag{2.12}$$

where γ is a chosen step size referred to as the learning rate in this context, and

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_t) = \frac{1}{N}\sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}}\, l(\mathbf{y}_i, f(\mathbf{x}_i; \boldsymbol{\theta}_t))$$

is the gradient of the total loss at the current time step with respect to the parameters of the model. This procedure is referred to as *batch* gradient descent, as it uses the entire dataset at each parameter update. The computational complexity of an update therefore scales linearly with the size of the training dataset, which is very prohibitive for even remotely large ones, and in practice would often lead to unacceptable training times.

An alternative to batch gradient descent is *stochastic* gradient descent, in which an individual example is picked from the dataset at each step. The gradient of the loss used to update the parameters depends on that example only, rather than the entire dataset. The update rule is then

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \gamma \nabla_{\boldsymbol{\theta}}\, l(\mathbf{y}_{i(t+1)}, f(\mathbf{x}_{i(t+1)}; \boldsymbol{\theta}_t)) \tag{2.13}$$

The sample loss $\nabla_{\boldsymbol{\theta}}\, l(\mathbf{y}_{i(t+1)}, f(\mathbf{x}_{i(t+1)}; \boldsymbol{\theta}_t))$ is an unbiased estimate of the batch loss although of great variance. The stochastic nature of the process $\{\boldsymbol{\theta}_t \,|\, t = 0, 1, ...\}$ is actually desirable to help escape local minima in the parameter space, which is high-dimensional in deep learning.

This time, the iteration complexity is independent of the dataset size. A drawback of stochastic gradient descent is that it does not use computational resources to their fullest potential : namely, it cannot benefit from batch processing, which is the processing of multiple examples in parallel and is possible on most modern architectures.

A middle ground between batch gradient descent and stochastic gradient descent is called *minibatch* stochastic gradient descent. In this case, the training examples are grouped in batches of size B, and the update rule now writes

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \gamma \frac{1}{B} \sum_{b=1}^{B} \nabla_{\boldsymbol{\theta}}\, l(\mathbf{y}_{n(t,b)}, f(\mathbf{x}_{n(t,b)}; \boldsymbol{\theta}_t)) \tag{2.14}$$

Setting an appropriate batch size depends on the desired trade-offs in the optimisation process, and the characteristics of available computational resources. Some observations towards smaller or larger batch sizes, and possible adjustments to the optimisation procedure are mentioned hereafter, inspired by the observations made by Goodfellow *et al.* [17, p. 276] and Russel & Norvig [18, p. 765-766].

While large batch sizes provide gradient estimate with less variance and would intuitively be preferable, it must be noted that smaller batch sizes which introduce greater variance in the learning process have a regularising effect, and that the generalisation error is often best for smallest batch sizes. Therefore, the batch size should be small enough to retain the regularising effect of stochastic optimisation, while utilising the available computational resources to benefit from the speed-up of batch processing.

Learning rate

Generally, strategies which are more elaborate than using a constant learning rate are preferred. In particular, the learning rate is usually scheduled to be annealed over time to favour convergence and avoid oscillations. Conversely, it is usually advised to start with a large learning rate in order to speed up training and escape spurious local minima. That being said, choosing the best learning rate value and schedule remains more of an art than a science since the precise interactions with other optimisation hyper parameters are still unclear, and therefore is a matter of trial and error.

A common strategy for learning rate scheduling involves decaying the learning rate by starting from an initial value γ_0 and multiplying it at the end of each epoch[2] by a factor that decreases as the training progresses. Denoting this factor by ζ_t where t is the current epoch number, many different ways of computing this factor exist, such as :

[2]An epoch is an entire pass through the training data during the optimisation procedure.

- Step-based decay, where the learning rate is divided by k every n epochs : $\zeta_t = \left(\frac{1}{k}\right)^{\left\lfloor \frac{t}{n} \right\rfloor}$
- Time-based decay, where the learning rate decreases inversely proportionally to the epoch number : $\zeta_t = \frac{1}{1+kt}$
- Exponential decay, where the learning rate decreases exponentially across epochs : $\zeta_t = \mathrm{e}^{-\mathrm{kt}}$
- Linear decay, where the multiplying factor linearly decreases from 1 to l_f over the duration of the training : $\zeta_t = (1 - \frac{t}{E})(1 - l_f) + l_f$ where E is the total number of epochs

These are illustrated in Figure A.2 in an hypothetical setting of 100 epochs.

Beyond these schedules, another idea that can be introduced in the optimisation process is the so-called warm-up. It involves using a lower learning rate at the very start of training to overcome troubles that may be encountered early on in the optimisation. Indeed, when minibatch stochastic gradient descent is used, the first examples being observed by the model will have a large impact on the optimisation as it is likely that the initial parameter values are inappropriate and induce significant corrections. Thus, if these examples happen to be somewhat unrepresentative of the training data as a whole, the parameter set will move towards an inappropriate direction and will need later iterations to correct it, hence lengthening the time needed for the optimisation process to converge. To address this issues, Goyal *et al.* [19] proposed a warm-up which starts with a small learning rate that is progressively increased over the first few training steps to reduce the primacy effect of early training examples, before shrinking according to a chosen schedule such as the ones previously outlined. This idea will be used in the training procedure, and an example is provided in Figure 2.2.

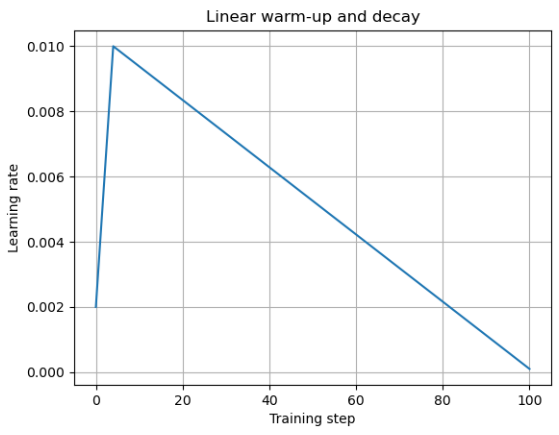


Figure 2.2: Learning rate which is linearly warmed up to $\gamma_0 = 0.01$ in the first 5 training steps before being linearly decayed with $l_f = 0.01$

Momentum

In the gradient descent methods described thus far, only the gradient at the current training step is used in order to update the parameters. In certain cases, the loss curvature may be such that the optimisation oscillates between the two ridges of a ravine, or significantly slows down in flat regions. While second order methods such as Newton's method may give information about the curvature and provide adapted learning steps, they require the computation of the Hessian matrix, which is intractable for deep learning models as it would involve computing $|\boldsymbol{\theta}|^2$ partial derivatives. Instead, we resort to a heuristic that guides the optimisation by incorporating information about the past behaviour of the gradient, adding inertia to the update direction at each training step. This approach is called *momentum* and is implemented by the following equations:

$$\mathbf{g}_t = \frac{1}{B}\sum_{b=1}^{B} \nabla_{\boldsymbol{\theta}}\, l(\mathbf{y}_{n(t,b)}, f(\mathbf{x}_{n(t,b)}; \boldsymbol{\theta}_t)) \tag{2.15}$$

$$\mathbf{u}_t = \alpha \mathbf{u}_{t-1} - \gamma \mathbf{g}_t \tag{2.16}$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \mathbf{u}_t \tag{2.17}$$

where $\alpha > 0$ is the momentum coefficient. The dynamics of the optimisation process are therefore modelled through an exponentially decaying average of the past gradient steps.

A clever alternative related to momentum is the so-called Nesterov momentum [20], which approximately looks one optimisation step ahead in order to preemptively adapt the direction of the next parameter update. Indeed, since it is known that the momentum term $\alpha \mathbf{u}_{t-1}$ will be used to update the parameters, then $\boldsymbol{\theta}_t + \alpha \mathbf{u}_{t-1}$ roughly gives the position of the next parameter set. Hence, the look-ahead is implemented by computing the gradient with respect to the approximate future parameters:

$$\mathbf{g}_t = \frac{1}{B}\sum_{b=1}^{B} \nabla_{\boldsymbol{\theta}}\, l(\mathbf{y}_{n(t,b)}, f(\mathbf{x}_{n(t,b)}; \boldsymbol{\theta}_t + \alpha \mathbf{u}_{t-1})) \tag{2.18}$$

$$\mathbf{u}_t = \alpha \mathbf{u}_{t-1} - \gamma \mathbf{g}_t \tag{2.19}$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \mathbf{u}_t \tag{2.20}$$

It is also worth mentioning that other optimisers which incorporate the idea of momentum, alongside an adaptive mechanism that allows the learning rate to be tuned for each parameter instead of using the same one for all parameters, since the loss curvature is most likely not isotropic in the parameter space. These include RMSProp [21] and Adam [22], the latter being the most widely used optimiser in deep learning nowadays.

Batch normalisation

In a continued effort to improve the efficiency of neural network training, Ioffe and Szegedy [23] proposed in 2015 a mechanism to alleviate the consequences of a phenomenon that significantly slowed down training. This phenomenon, which they referred to as internal covariate shift, is the fact that the distribution of the network's activations changes as the network parameters are updated during training, causing instabilities in the training process and requiring the use of smaller learning rates. To address this issue, they suggested to normalise the layer inputs by using batch normalisation in order to fix their mean and variance. To this end, each d-dimensional input $\mathbf{x}_i = (x^{(1)}, x^{(2)}, ..., x^{(d)})_i,\ i = 1, ..., B$ of a layer in the network for a minibatch of size B is re-centred and re-scaled through

$$\hat{x}_i^{(k)} = \frac{x_i^{(k)} - \mu^{(k)}}{\sqrt{\sigma^{2(k)} + \epsilon}} \cdot \gamma^{(k)} + \beta^{(k)} \tag{2.21}$$

where $\gamma^{(k)}$ and $\beta^{(k)}$ are learnable parameters, ϵ is a small constant which ensures numerical stability, and $\mu^{(k)}$ and $\sigma^{2(k)}$ are respectively per-dimension mean and variance computed over the minibatch examples. At inference time, the mean and variance used to normalise are instead computed over the entire training population to ensure that the model is deterministic.

Weight decay

Deep learning models can be quite prone to overfitting when naively trained, without proper regularisation methods included in the training process. Among these, weight decay is frequently used. It consists in penalising the magnitude of the parameters in the model, hence encouraging smaller parameter values and simpler models. Generally, the penalty concerns the weight but not the bias parameters of the model. Denoting the weights of the model by $\mathbf{w}$, weight decay is usually implemented by adding a fraction of the l_2 norm of $\mathbf{w}$ to the loss function to form the weight-decayed loss $\tilde{\mathcal{L}}(\boldsymbol{\theta})$:

$$\tilde{\mathcal{L}}(\boldsymbol{\theta}) = \mathcal{L}(\boldsymbol{\theta}) + \lambda \|\mathbf{w}\|^2 \tag{2.22}$$

where $\lambda > 0$ controls the strength of the penalty.

2.3 Deep learning for computer vision

Over the years, deep learning models have proven to be outstanding at carrying out computer vision tasks, often yielding state-of-the-art performance. However, these achievements could not have been possible if fully connected feedforward networks had simply been thrown at these tasks without any refinements. Indeed, images are a particular type of data whose characteristics can and should be exploited. In this section, these particularities will be presented, as well as the adapted architectural blocks.

2.3.1 Images as structured data

In Section 2.2.1, the input to a unit of a neural network was in all generality referred to as a vector $\mathbf{x}$, implying that any permutation of its values would not alter its meaning. This does not hold in the case of an image, as the numerical values in the input represent pixels whose particular arrangement constitute visual coherence. Rather than a vector, an image is naturally represented as $H \times W$ matrix, where H and W are the image's height and width respectively. Images often have an additional dimension C representing its channels. A coloured image is often represented through its RGB[3] components, hence possessing 3 channels.

[3]Stands for Red-Green-Blue, a color model in which an image is represented as the addition of shades of red, green and blue

While the channel index is only the consequence of a representation choice and usually has no meaning, the height and width indexes are of utmost importance because images have a spatial structure. Moreover, they are high dimensional and can exhibit extreme variability for a same output, which would require very high capacity models to make sense of if proper inductive biases were not included in the architecture. As such, it would be naive and inefficient to use a fully connected architecture : for instance, if a flattened 256×256 pixels coloured image, hence described by $256 \times 256 \times 3 = 196608$ values, was fed as input and fully connected to a hidden layer containing as many units, it would already require over 38 billion weights and as many biases. Building a model in this fashion would therefore be impractical.

Instead, a more clever approach that capitalises on the spatial structure of images and induces the model to hierarchically process their components while also reducing the number of parameters should be taken. Architectural components which achieve this are described in Section 2.3.2.

2.3.2 Architectural components

Convolution

A convolution is a particular linear operation in which a convolutional kernel is slid across a signal. In the case of images, the kernel slides across its spatial dimensions. Mathematically, following the formalism found in [24], an input feature map $\mathbf{x} \in \mathbb{R}^{C\times H\times W}$ convoluted by a kernel $\mathbf{u} \in \mathbb{R}^{C\times h\times w}$ produces an output feature map $\mathbf{o} \in \mathbb{R}^{(H-h+1)\times(W-w+1)}$ such that

$$\mathbf{o}_{j,i} = \mathbf{b}_{j,i} + \sum_{c=0}^{C-1}\sum_{n=0}^{h-1}\sum_{m=0}^{w-1} \mathbf{x}_{c,n+j,m+i} \cdot \mathbf{u}_{c,n,m} \tag{2.23}$$

where the bias $\mathbf{b}_{j,i}$ and the kernel $\mathbf{u}$ are learnable parameters. An illustration of such a convolution is provided in Figure 2.3, in a simplified single channel setting.

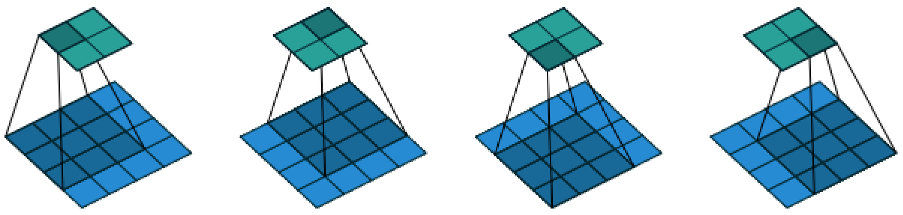


Figure 2.3: A 4×4 input feature map convoluted by a 3×3 kernel produces a 2×2 output feature map (Credits : Dumoulin & Visin [25])

A unit in the output is connected only to a subset of the input, known as the unit's receptive field, which is why they are said to be sparsely connected. Moreover, since the kernel is slid across, its weights are shared rather than specific to each output unit. Hence, the number of parameters is drastically reduced compared to a fully connected layer. Moreover, the locality principle is enforced since each output unit is the consequence of a local region of the input only, and weight sharing ensures that the output is equivariant to a translation of the input.

Multiple convolutions can be stacked in the same fashion, each with its own parameters, to form a convolutional layer of depth D and produce an output feature map $\mathbf{o} \in \mathbb{R}^{D\times(H-h+1)\times(W-w+1)}$. In this case, each convolution can be seen as a filter that locally detects a learnt pattern in the input feature map. Convolutional layers can also be composed in series to successively process the feature maps. Intuitively, this makes sense as this induces the model to deal with the input image as a hierarchical composition of features, each successive layer dealing with a more and more global context. Indeed, the effective receptive field of a unit is larger the deeper it is in the architecture. This is illustrated in Figure 2.4.

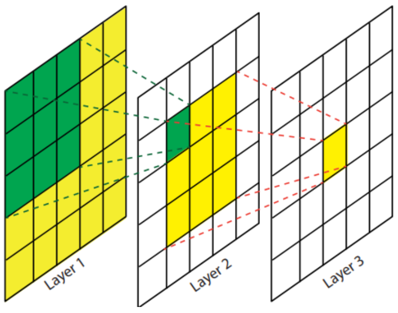


Figure 2.4: Growth of a unit's effective receptive field (Credits : Lin *et al.* [26])

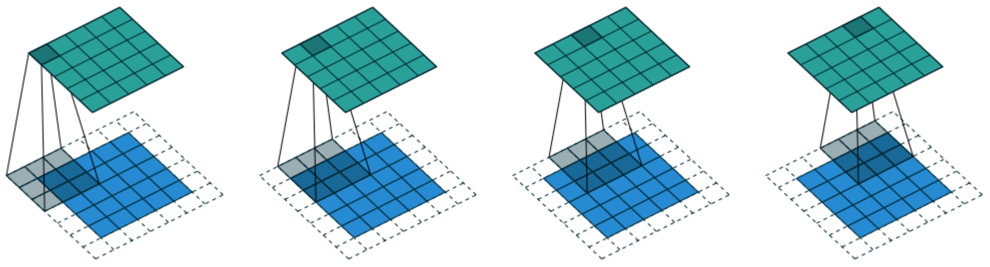
Some parameters may also be specified to a convolutional layer to slightly alter its behaviour. Padding may be used to add a blank frame around the input to artificially increase the spatial dimension of the output feature map. Stride may be used to specify the step size used by the kernel when sliding across the input in order to reduce the spatial dimension of the output feature map. Dilation may also be used to specify if the kernel points should be spaced, hence increasing the size of resulting units' receptive field without increasing the number of parameters. These different behaviours are illustrated in Figure 2.5.

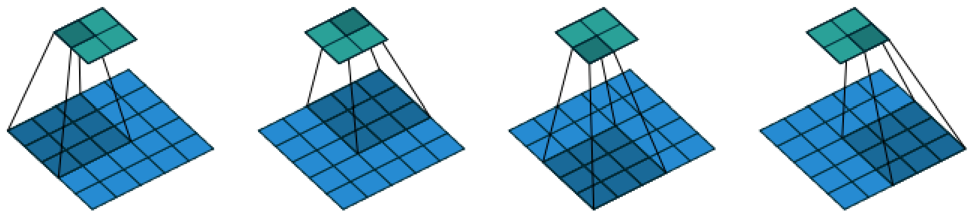
Pooling

Pooling is an operation which usually follows one or multiple convolutions to reduce the spatial dimension of the feature map. It summarises subsets of spatially adjacent units in the input through a statistic computed over them. The most commonly used statistic is the maximum of the values found in the subset. Indeed, in computer vision the existence of a feature in a region is often more important than its exact location, and summarising small regions by their maximum allows the model to be approximately invariant to local translations. As in convolutional layers, padding, stride and dilation may also be specified. An example is given in Figure 2.6.

Convolutional neural networks

Using the building blocks presented previously in this work, simple yet already quite powerful convolutional neural networks can be designed. Such a model can be designed by combining convolutional layers, activation functions and pooling layers to form a backbone whose role is to extract features from the image. These features can then be processed, typically by fully connected layers as described in Section 2.2.2, to produce an output that depends on the computer vision task. Such architectures constitute the foundations of deep learning models in computer vision and have been incrementally improved over time. Since then, several architectural


(a) (1,1) padding adds a border of 0 values around the spatial dimensions of the input


(b) (2,2) stride slides across the input with a step size of 2 in both spatial directions

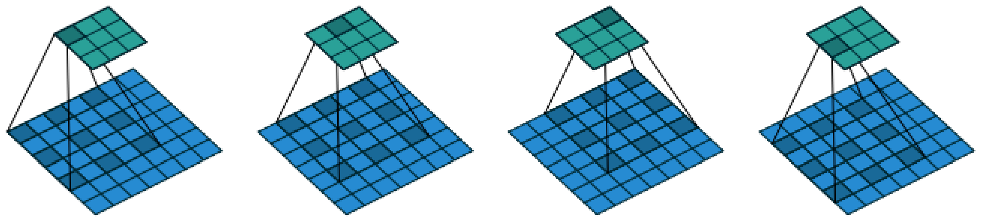

(c) (2,2) dilation increases the units' receptive field by 2 in both spatial directions

Figure 2.5: Convolution variants (Credits : Dumoulin & Visin [25])

tweaks have been designed to surpass previous performance or solve new tasks, but the groundwork remains relevant.

2.4 Computer vision tasks

In order to understand how deep learning models carry out object detection, it can be interesting to dissect it into individual tasks. Namely, object detection can be described as the localisation and classification of objects in an image. Classification will first be described, followed by what the localisation task entails in the case of a single object, and finally how these can be combined to formulate multiple object detection.

2.4.1 Classification

Classification is a statistical task which exists in various fields and involves associating an input to a class, which is chosen among a predefined, finite set of classes. In the context of computer vision, the input is an image supposedly representing one of these classes. This image is processed by a model which extracts its features and uses them to predict which class it belongs to.

In convolutional neural networks, this prediction comes in the form of a vector $\mathbf{z}$ representing a categorical probability distribution where each index corresponds to a class, and the class prediction corresponds to the highest probability in that vector. Assuming a setting where C classes are possible, this vector is produced by having

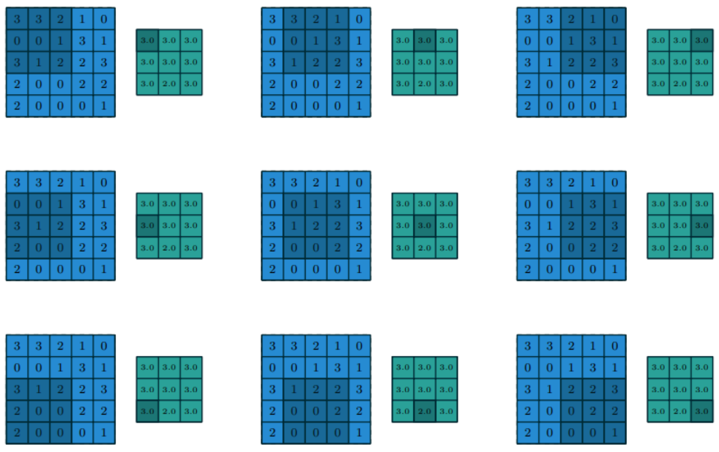


Figure 2.6: 3×3 max pooling with no padding, unit stride and dilation (Credits : Dumoulin & Visin [25])

the network output C real values in the last layer of its fully connected component. To convert these C real values into C class probabilities, a softmax activation is used, which transforms these values as follows :

$$\text{softmax}(\mathbf{z})_i = \frac{\mathrm{e}^{\mathrm{z_i}}}{\sum_{j=1}^{C} \mathrm{e}^{\mathrm{z_j}}} \quad , \quad i = 1, ..., C \tag{2.24}$$

This ensures that the vector $\mathbf{z}$ contains only values between 0 and 1 such that their sum is equal to 1.

To train convolutional neural networks to perform classification, one must define a loss function with respect to which they will be optimised. Denoting by $(p_1, ..., p_C)$ the components of $\mathbf{z}$, a quadratic loss is sometimes used :

$$l(\mathbf{z}, \mathbf{y}) = \sum_{i=1}^{C} (y_i - p_i)^2 \tag{2.25}$$

where $\mathbf{y}$ is a one-hot vector encoding the true class of the input image, *i.e.* it has value 1 at the index of that class and 0 elsewhere. However, the preferred and by far most commonly used loss is the cross entropy loss, which is equal to

$$l(\mathbf{z}, \mathbf{y}) = -\sum_{i=1}^{C} y_i \log p_i \tag{2.26}$$

Hence, the cross-entropy loss increases steeply as the predicted probability of the ground truth class decreases to 0 and heavily penalises large errors.

2.4.2 Localisation

In classification tasks, a class frequently refers to a real-life object which more often than not occupies only part of the whole image. Localisation consists in delimiting a rectangular area around the object, often referred to as a *bounding box*, such that it is as tight as possible. A bounding box is characterised by a 4-tuple $\mathbf{t} = (x, y, w, h)$, where x and y are the Cartesian coordinates of the bounding box centre, and w and h are its width and height respectively. Predicting this tuple can therefore be formulated as a regression problem. Ordinarily, a quadratic loss is used to train localisation models :

$$l(\mathbf{t}, \hat{\mathbf{t}}) = (x - \hat{x})^2 + (y - \hat{y})^2 + (w - \hat{w})^2 + (h - \hat{h})^2 \tag{2.27}$$

where $\mathbf{t}$ is the ground truth and $\hat{\mathbf{t}}$ is the prediction of the model. While this is a straightforward and simple formulation of the loss, it has a major flaw in that it is not scale invariant. To mitigate this issue, a common workaround consisted in taking the square root of the width and height terms when computing the loss. While effective, it still did not achieve scale invariance, which prompted the research community to design an alternative.

The quality of a localisation is usually measured by its intersection over union score, often denoted by its acronym IoU. This metric measures the overlap between the ground truth and the predicted bounding box, respectively A and B. As its name suggests, it is defined as

$$\text{IoU}(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{2.28}$$

While it is scale invariant, $l_{IoU} = 1 - \text{IoU}$ cannot be used as a loss function. Indeed, if A and B are disjoint, it has a constant value of 0 and it would therefore halt the gradient-based optimisation process. The design of an IoU based loss function was incremental. First, Rezatofighi *et al.* [27] proposed the Generalised Intersection over Union and the corresponding loss $l_{GIoU} = 1 - \text{GIoU}$ where the GIoU is

$$\text{GIoU}(A, B) = \text{IoU}(A, B) - \frac{|C - (A \cup B)|}{|C|} \tag{2.29}$$

where C is the smallest box enclosing both A and B. While it undeniably remedied the case where bounding boxes were non-overlapping, Zheng *et al.* [28] showed it converged quite slowly because it did not incorporate the distance between bounding boxes. They subsequently introduced the Distance-Intersection over Union and loss $l_{DIoU} = 1 - \text{DIoU}$ such that

$$\text{DIoU}(A, B) = \text{IoU}(A, B) - \frac{d^2}{c^2} \tag{2.30}$$

where d is the distance between the central points of A and B, and c is the diagonal length of the smallest box enclosing A and B and is used to maintain scale invariance. Then, Zheng *et al.* [29] added onto the DIoU by proposing the Complete Intersection over Union, which integrated a term capturing aspect ratio :

$$\text{CIoU}(A, B) = \text{IoU}(A, B) - \frac{d^2}{c^2} - \alpha V \tag{2.31}$$

where $V = \frac{4}{\pi^2}\left(\arctan \frac{w^A}{h^A} - \arctan \frac{w^B}{h^B}\right)^2$ in which w^A and h^A indicate the width and height of A, and where trade-off parameter

$$\alpha = \begin{cases} 0 & \text{if IoU} < 0.5 \\ \frac{V}{(1-\text{IoU})+V} & \text{if IoU} \geq 0.5 \end{cases} \tag{2.32}$$

is designed such that the loss $l_{CIoU} = 1 - \text{CIoU}$ ignores the aspect ratio until the bounding boxes overlap enough, before taking it more and more into account as the bounding boxes' IoU increases. The authors found this loss to improve both convergence speed and final performance.

2.4.3 Object detection

The task of object detection involves predicting the bounding box and class of all the objects in the input image that belong to a predefined set of classes. In the simplified setting where it is known that there is exactly one object per image, an object detection model could be trained using a loss that combines the localisation and classification losses by summing them up. However, the number of objects is generally not known beforehand, and the model should be able to adjust its number of predictions based on the input image.

Generally, object detection models predict a large number of bounding boxes in predefined regions before merging them. This is usually done through non-maximum suppression (see Appendix B).

2.5 Object detection models

Different approaches may be taken in order to implement object detection models. A first one could consist in sliding a window across the input image and classifying each crop using a convolutional neural network. However, it would prove to be either ineffective if a fixed-sized window was used as objects may appear at many different scales, or inefficient if windows of varying scales were used as an input image of $n \times n$ pixels could hold $O(n^4)$ axis-aligned rectangular windows and thus require just as many evaluations by the classifier. Instead, modern object detection models seek less computationally expensive approaches. Two main families exist and will be presented in Sections 2.5.1 and 2.5.2, with a particular attention to the model which will be used in this work.

2.5.1 Two-stage detectors

This family of models performs object detection by first extracting candidate regions which likely contain an object before classifying them. The first breakthrough was a model called R-CNN [30], which relied on a region proposal algorithm, namely selective search, to extract 2000 regions in the image that were likely to contain an object. It then processed each region using a convolutional neural network to extract one feature vector per region. Finally, these features were fed into C class-specific linear support vector machines to classify the object, and a linear regression model to predict the bounding box. While this model showed substantial improvements compared to previous state-of-the-art, it was quite slow since extracting the features required 2000 passes through the CNN.

Following this, Fast R-CNN [31] introduced improvements that significantly boosted the detection speed. Instead of computing the feature vector for each region proposal, it computed a single feature map for the entire image and matched each region proposal to a fixed-size feature vector by using a region of interest (RoI) pooling layer. This feature vector was then fed into fully connected layers that branched into two output layers to predict class and bounding box. This change made object detection 213 times faster than R-CNN, while also improving performance.

Finally, since the quality of object detection was tied to the quality of the region proposal algorithm which took most of the compute time, Faster R-CNN [32] proposed to use another convolutional network to generate the region proposals. It introduced the Region Proposal Network, which reduced the time needed for region proposal by a factor 200 compared to selective search.

2.5.2 One-stage detectors

Instead of formulating object detection as a two-stage process, one-stage detectors rather frame object detection as a regression problem. The predictions of both bounding boxes and class probabilities are made using a single model and do not rely anymore on separate region proposals, and can thus be optimised end-to-end on the detection task.

Among notable models, the Single Shot Multibox Detector (SSD) [33] performs object detection by predicting class probabilities and bounding box coordinate offsets relative to default boxes on feature maps at different scales. The predictions are then merged using non-maximum suppression. The model's architecture is illustrated in Figure 2.7.

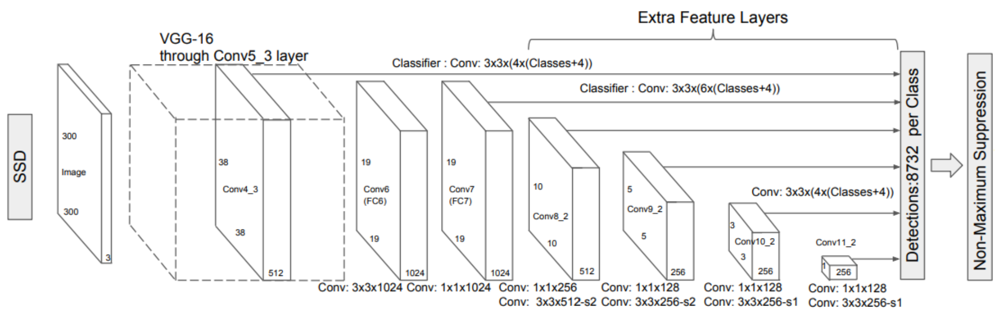


Figure 2.7: Architecture of SSD (Credits : Liu *et al.* [33])

In this work, we will use an evolution of another popular one-stage detector. Proposed in 2016 by Redmon *et al.*, You Only Look Once (YOLO) has since then been successively enhanced to integrate the latest innovations. YOLO and its relevant subsequent modifications are described hereafter.

YOLO [34]

YOLO is a convolutional neural network which outputs both bounding boxes and class probabilities in a single forward pass through a unified model. It proceeds by dividing the input image into an $S \times S$ grid, and predicting B bounding boxes and confidence scores as well as C class probabilities for each grid cell, hence making the assumption that each cell contains the centre of a single object at most. Each bounding box B is predicted through a 4-tuple (x, y, w, h) where x and y are the bounding box central coordinates relative to the bounds of the grid cell, and w and h are the width and height relative to the whole image. The confidence score associated to each bounding box reflects both the model's confidence that it contains an object and how accurate it is, and is thus defined as $P(\text{Object}) \cdot \text{IoU}(truth, pred)$ where *truth* and *pred* are the ground truth and predicted bounding boxes respectively. This formulation makes the model strive for 0 confidence predictions when no objects are contained the cell, and confidence equal to the IoU with the corresponding ground truth bounding box when there are.

The bounding boxes are associated to the predicted class of the grid cell that produced them to yield the detections. These are then combined through non-maximum suppression to avoid multiple detection of the same object. An overview of the system is provided in Figure 2.8.

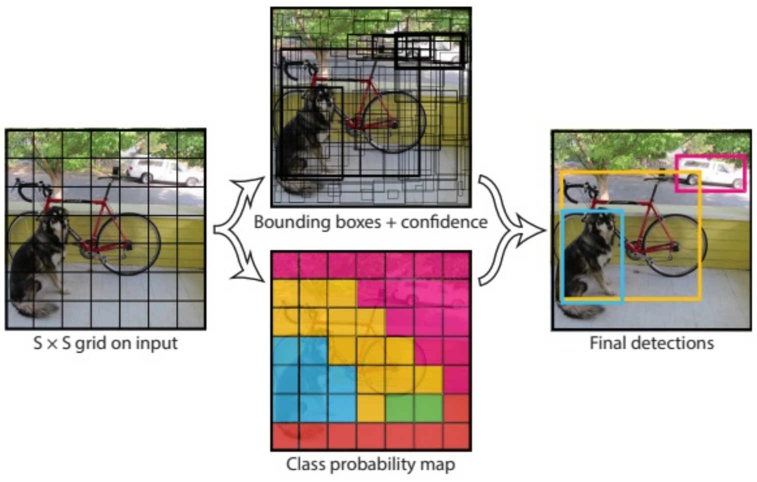


Figure 2.8: Object detection with YOLO (Credits : Redmon *et al.* [34])

YOLO takes a fixed size 448 × 448 input image and is composed of 24 convolutional layers followed by 2 fully connected layers that produce the final output tensor of shape $S \times S \times (B * 5 + C)$. The first 20 convolutional layers were pre-trained on the ImageNet classification dataset, while the last 4 convolutional and the 2 fully connected layers were trained on the PASCAL VOC 2007 [35] object detection dataset. In this case, the authors used $S = 7$ and $B = 2$ to detect the $C = 20$ classes of the PASCAL VOC dataset, yielding the architecture and $7 \times 7 \times 30$ output tensor shown in Figure 2.9.

YOLO pioneered the one-stage detection approach to strive towards real-time object detection. It achieved detection at 45 frames per second which was must faster than contemporary two-stage detectors, but lagged slightly behind them in terms of performance. In particular, only predicting a single vector of class probabilities and a small number of bounding boxes per grid cell imposed strong spatial constraints which severely limited the number of nearby objects that could be detected by the model.

YOLOv2 [36]

In YOLOv2, the same authors bring in new innovations, some being taken from or inspired by other contemporary works. Among others, these include using batch normalisation and anchor boxes, which are sets of bounding boxes with predefined width and height. The network predicts offsets to the anchor boxes' centre, width

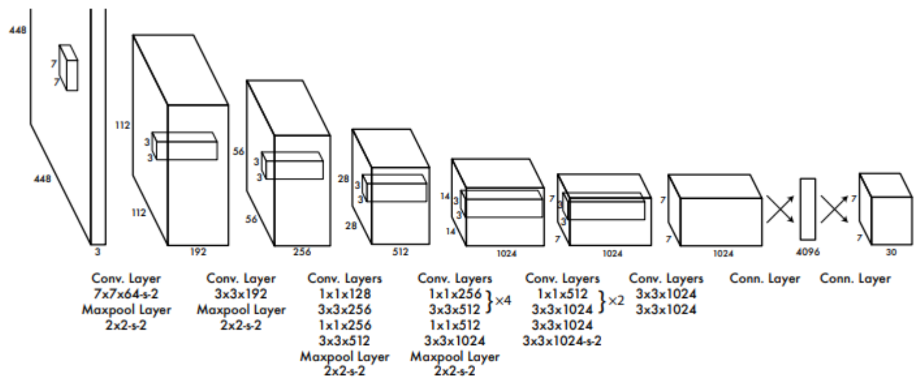


Figure 2.9: Architecture of YOLO (Credits : Redmon *et al.* [34]), where optional -s-2 after convolution or max pooling operations indicate that a stride 2 was used

and height instead of predicting raw values, which eases the learning process as prior dimensions can be readily used instead of learnt from the ground up. While previous works often used handpicked ones, YOLOv2 formulates the choice of anchor boxes as a preliminary learning step. It runs k-means clustering on the training set bounding boxes using 1-IoU(G, C) as the distance metric, where G is any ground truth box and C any centroid. YOLOv2 uses $k = 5$ bounding box priors. Moreover, instead of predicting a single class probability vector per grid cell, YOLOv2 predicts one per anchor box and therefore has a $S \times S \times B * (5 + C)$ output tensor.

YOLOv2 changes its convolutional feature extractor to an architecture of their own called Darknet-19 that allows fast processing while maintaining performance. It also removes the fully connected layers at the end of the model, and instead uses a convolutional layer to produce the final output tensor. Inspired by Faster R-CNN and SSD which rely on multiple feature maps at various resolutions to produce their detections, YOLOv2 includes passthrough connections which stack adjacent high resolution features into different channels to match the shape of low resolution features before concatenating them. Thus, both high resolution and low resolution features can be used by the final convolutional layer to output the model's detections.

YOLOv3 [37]

The main contributions of YOLOv3 are the introduction of a new feature extractor called Darknet-53, a deeper hybrid between YOLOv2's Darknet-19 and He *et al.* residual networks [38], and multi-scale predictions. The latter is achieved by predicting bounding boxes at 3 different scales. This is done by upsampling feature maps at given depths before concatenating them with feature maps of earlier layers, and using the results to output detections such that these benefit both from coarser but semantically meaningful feature maps as well as finer-grained ones. YOLOv3 uses this at 3 different depths and combines the resulting detections as usual through non-maximum suppression to produce its final predictions.

YOLOv5 [39]

YOLOv5 is an evolution of YOLOv3, despite its misleading name that would suggest it follows YOLOv4 [40]. YOLOv4 was released a few months prior to the first release of YOLOv5, both works improving YOLOv3 by incorporating some of the latest innovations. To avoid giving both models the same name, which would lead to even more confusion, the authors decided to use YOLOv5 even though their work only partially builds upon YOLOv4.

YOLOv5 is a model developed by Ultralytics, a company whose main activity is to implement YOLO architectures in the PyTorch framework such that the models can easily be trained, integrated and shared in common formats. YOLOv5 was initially released by Ultralytics in May 2020, and has since then undergone small changes in subsequent major releases, which are available on GitHub[4]. This work uses the version 6.1 of the model to perform object detection, described hereafter.

YOLOv5 uses a CSP-Darknet as its backbone, where CSP indicates cross-stage partial connections. This mechanism proposed by Wang *et al.* [41] partitions the feature maps into two parts such that one is processed by the current stage of the model, while the other is transmitted as-is. Both parts are then concatenated before moving on to the next stage of the model.

Before being transmitted to the prediction head, the feature map resulting from the backbone passes through what is often referred to as the model's neck. In YOLOv5, the neck is composed of PANet-like[5] [42] passthrough connections, and a SPP[6] block [43]. The PANet-like connections' purpose is to aggregate features from different backbone levels in a bottom-up fashion. The SPP-block was adapted to fully convolutional YOLO architectures by Huang *et al.* [44] and produces an output feature map of constant spatial size by pooling the input feature map at different scales and concatenating the results along the channel dimension.

Finally, the detection head of YOLOv5 is the same as the one used in YOLOv3. Namely, it still uses learnt anchor-based detection at 3 different resolutions by exploiting the upsampling and concatenation mechanism described earlier. Objects are detected by using 3 anchor boxes per grid cell at each resolution.

The network is trained end-to-end by minimising a 3-part loss function capturing localisation using a CIoU-based loss, and class prediction and confidence using cross-entropy based-losses, and which can be written

$$\begin{aligned}\sum_{r\in\mathcal{R}}\Bigg[&\lambda_{box}\sum_{i=1}^{S_r\times S_r}\sum_{j=1}^{B}\mathbb{1}_{i,j}^{o}\Big(1-\mathrm{CIoU}(T_o,\hat{P}_{i,j})\Big)\\ &-\lambda_r\lambda_{conf}\sum_{i=1}^{S_r\times S_r}\sum_{j=1}^{B}\big(c_{i,j}\log(\hat{c}_{i,j})+(1-c_{i,j})\log(1-\hat{c}_{i,j})\big)\\ &-\lambda_{cls}\sum_{i=1}^{S_r\times S_r}\sum_{j=1}^{B}\mathbb{1}_{i,j}^{o}\sum_{c=1}^{C}p_{o,c}\log(\hat{p}_{i,j,c})+(1-p_{o,c})\log(1-\hat{p}_{i,j,c})\Bigg]\end{aligned} \tag{2.33}$$

[4]`https://github.com/ultralytics/yolov5/`

[5]Path Aggregation Network

[6]Spatial Pyramid Pooling : a pooling technique which produces fixed-length representations out of variable size inputs by pooling subregions in a fixed number of local spatial bins whose sizes are proportional to the input's size

Each ground truth label in the input image is assigned the anchors whose aspect ratios best match with the ground truth bounding box, as explained in Figure 2.10. With this in mind, the notations used in the expression of the loss are defined as follows :

- $\mathbb{1}^{o}_{i,j}$ is equal to 1 if the anchor j of cell i is assigned to the ground truth bounding box of any object o, and 0 otherwise;
- T_o is the ground truth bounding box for object o;
- If anchor j of cell i is assigned to any ground truth bounding box, $c_{i,j}$ is equal to the IoU between this ground truth bounding box and predicted bounding box derived from that anchor. Otherwise, $c_{i,j}$ is 0;
- $p_{o,c}$ is equal to 1 if the ground truth class of object o is c, and 0 otherwise;
- $\hat{P}_{i,j}$ is the predicted bounding box obtained by offsetting the anchor j of cell i;
- $\hat{c}_{i,j}$ is the predicted confidence of the bounding box obtained from anchor j of cell i;
- $\hat{p}_{i,j,c}$ is the predicted probability that the bounding box obtained from anchor j of cell i contains an object of class c;

$$r_w = w_{gt}/w_{at}$$
$$r_h = h_{gt}/h_{at}$$
$$r_w^{max} = max(r_w, 1/r_w)$$
$$r_h^{max} = max(r_h, 1/r_h)$$
$$r^{max} = max(r_w^{max}, r_h^{max})$$
$$r^{max} < \text{anchor}_\text{t}$$

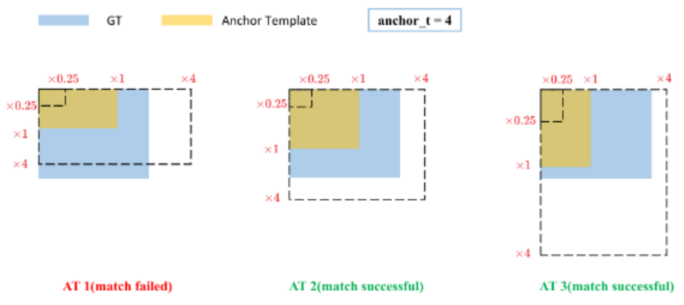


Figure 2.10: Matching anchor templates to ground truths. In Yolov5, the threshold anchor$_t$ is set to 4.

The loss is computed for each resolution $r \in \mathcal{R}$, and the results are summed up. The parameters λ_r allow to weigh the contribution of the confidence to the loss differently based on the resolution at which the detections were output. The parameters λ_{box}, λ_{conf} and λ_{cls} allow to weigh the contribution of each part of the loss and are set to ensure that training progresses smoothly.

Chapter 3

Related work

Synthetic data generation for machine learning is a fairly recent technique which has been increasingly used to circumvent the time, cost and potential privacy issues that real data collection entails. Each field has come up with its own solutions depending on the nature of the required data. In this chapter, we focus on visual tasks and review relevant works which synthetically generate training data for computer vision models.

Photo-realistic synthetic data

A possible approach consists in generating synthetic images which are visually similar to the real images on which the models will ultimately operate. These usually require to simulate environments with a high degree of photo-realism. To avoid the time-consuming process of creating such a simulator, Richter *et al.* [45] exploited a popular open-world modern computer game to create pixel-level semantic labels of scenes. These scenes included realistic first person footage in different ambient conditions such as driving through both urban and rural landscapes or walking on sidewalks. They showed that the same semantic segmentation performance could be reached by using only a fraction of the real data supplemented by their synthetic data. Following this work, the authors subsequently released a benchmark suite for a variety of visual perception tasks [46], notably useful for autonomous driving research. Compelled by the prospects of self-driving cars and their dire need for high variety training footage, other authors have put up datasets [47] or open-source simulators [48] for use.

For object detection, creating synthetic data often involves composing scenes with the foreground objects of interest, and different backgrounds meant to replicate plausible situations in which these objects could be found. Georgakis *et al.* [49] showed these scenes can for instance be created by composing crops of real foreground objects taken from some dataset, and backgrounds of other datasets. Dwibedi *et al.* [50] showcased a similar approach and demonstrated that integrating foreground objects without any form of blending created local boundary artifacts which models learned to identify rather the visual appearance of the objects, hence failing to generalise well to real images. When these foreground objects are very specific, similar-looking models may not be available at all. In such cases, 3D replicas can be created through photogrammetry techniques[1] and used in the synthetic data generation process, as demonstrated by Wong *et al.* [51].

[1]Photogrammetry involves reconstructing a 3D surface from several 2D images

Domain randomisation

Photo-realistic scene composition is not trivial and usually requires careful adjustments. Moreover, some simulators may compose scenes in such a way that they fool the human eye to appear realistic, but may not reflect reality and therefore provide neural networks with exploitable features which do not generalise to real data. Another radically different approach to synthetic data generation is domain randomisation, which consists in making the synthetic training data exhibit extreme variability with much less concern for coherence. The underlying premise is that if there is enough randomisation in the training data, then the model should learn how to discriminate between relevant and irrelevant image features and thus generalise well, since the reality would appear to the model as just another randomisation.

Among the first to bring this approach to deep neural networks, Tobin *et al.* [52] trained a detector using synthetic images constructed with random camera and object positions and unrealistic textures, and successfully deployed in a simple but real environment. Tremblay *et al.* [53] extended their work to detection with non-trivial backgrounds by including various distractors alongside objects of interest in their synthetic images, and showed that these could be used as a pre-training step when real images are available. Capitalising on the principle of distractors, Hinterstoisser *et al.* [54] cluttered the background with objects scaled to appear with approximately the same size as foreground objects, and included occluders to partially cover them up.

Instead of full-on domain randomisation, a less drastic approach consists in randomising scenes while preserving contextual properties of the objects appearing. Known as structured domain randomisation, the objects and distractors are placed according to problem-specific distributions rather than uniform ones. Prakash *et al.* [55] reported slightly better results in vehicle detection, but the approach is admittedly more involved.

Domain adaptation

Yet another method to integrate synthetic data, domain adaptation is a technique involving the transformation of the synthetic data to make it better suited to the real domain in which the model will operate. Usually, the environment in which the synthetic data is generated is referred to as the source domain, and the real environment is referred to as the target domain. There exist models which learn mappings from one domain to another through labelled pairs, such as pix2pix [56]. However, the unsupervised method requiring only unpaired images of source and target domains is simpler and applicable to many more practical situations, hence why this approach only is reviewed.

Unsupervised GAN-based domain adaptation have exhibited promising results. Bousmalis *et al.* [57] use a pixel-based content similarity loss on the foreground objects to learn how to change objects in a consistent way that preserves shape. Thus, it assumes that the difference between the domains are low level features such as noise, resolution or illumination, but not high level features such as geometric variations. A looser formulation such as the one of Shrivastava *et al.* [58] allows more flexibility in the domain adaptation, while preventing artifacts to appear through a local adversarial loss which enforces realism separately onto local patches.

One way to consider the problem of domain adaptation is through the prism of cycle-consistency, which enforces that the mapping from source to target back to source should approximate the identity function. Hoffman *et al.* [59] showed it could achieve state-of-the-art results in a variety of adaptations tasks. CycleGAN [60] jointly learns two generative networks constrained to produce roughly inverse mappings through cycle-consistency. CUT [61] reformulates the cycle-consistency to be patch-wise rather than a complete bijection, and uses contrastive learning to ensure correspondence of patches in the same spatial locations.

Training strategies

While the characteristics and the quality of the synthetic data is of foremost importance, the way it is employed crucially matters too. Training procedures can be adapted to accommodate for synthetic data and use it as effectively as possible. Hinterstoisser *et al.* [62] observe that recent architectures are often composed of feature extractors pre-trained on large datasets followed by task-specific layers. Postulating that these feature extractors are already rich enough for visual tasks and do not need further training, especially on data which may perturb them due to the domain shift, the authors argue that their weights should be frozen when training on synthetic data. They show that object detectors trained this way outperform those who are fully re-trained, almost reaching the performance of detectors trained on real data. However, these findings must be tempered as Tremblay *et al.* [53] find this method to decrease performance in their experiments, hence the benefits of this approach may be data-dependent.

When both synthetic and real data are available, finding the most effective way to combine them may not be obvious. Nowruzi *et al.* [63] provide an analysis which compares a training procedure which mixes both types of data to a transfer learning approach which pre-trains the model on the synthetic data before fine-tuning on the real data. They find that the latter generally yields better models, with especially improved recall.

Domain-invariant representations

Although it won't be used in this work, another idea worth mentioning involves designing models such that the representations in the synthetic and real domains are shared. For instance, Bousmalis *et al.* [64] use an architecture and training procedure separating the representation components which are specific to each domain from the ones common to both, such that the downstream task can then be learnt on the shared representation only. Ganin *et al.* [65] describe domain-adversarial training, which inserts a domain classification head to any model and backpropagates the reverse classification loss throughout the entire network such that the learnt representation cannot be used to distinguish one domain from another. A similar idea suggested by Tzeng *et al.* [66] backpropagates a loss which intends to maximise the entropy of the distribution output by the domain classification head, such that the learnt representation is directed towards one that yields a uniform probability over the domains.

Chapter 4

Dataset and methodology

In this chapter, we present in Section 4.1 the dataset on which the experiments were carried out, and in particular which changes were made with respect to the original dataset. Then, the model and the settings used for training and evaluation are described in Section 4.2.

4.1 Dataset

The dataset used in this work is provided by the data science & artificial intelligence department of NRB and consists of objects classified according to the waste category they belong to. The images were mostly collected by taking photographs of the different objects being held or lying on different surfaces. The bounding boxes were then manually annotated. The initial dataset consisted of 11 482 images, out of which 10 453 contained labels. While this could have been due to background-only images being present in the dataset, a manual check revealed that these were actually images who contained objects but had simply not been annotated yet. Therefore, these 1029 images were removed from the dataset. The initial dataset contains 12 classes, which are unevenly represented in the labels. Their distribution among classes is illustrated in Figure 4.1.

As it is, the dataset has some notable flaws. Since the classes are based on waste categories, some classes have visual overlap, which makes it prominently difficult to correctly classify detected objects. Notably, *DSM*, *FR* and *Huile de friture* objects are very similar to *Bout./flac. en plastique < 8L*, as illustrated through handpicked samples in Figure E.3. Other visual overlaps exist, for instance between *Bout./flac. en plastique > 8L* and *PDP* as illustrated in Figure E.1, but these are not frequent enough to warrant a merge. Lastly, the class *Emballage métallique* suffers from labels being quite arbitrary. Indeed, as illustrated in Figure E.2, most backgrounds are cluttered with metallic objects, but these were not consistently labelled. This could both confuse the model during training, and produce misleading metrics during validation and testing.

Considering the previous observations, *DSM*, *FR* and *Huile de friture* are merged under the *Bout./flac. en plastique < 8L* label to avoid unpredictable impacts on the performance that these similarities may bring when observing the benefits of synthetic data. To carry out the experiments that will follow, a reduced set of classes was considered such that these exhibit enough visual consistency and are sufficiently

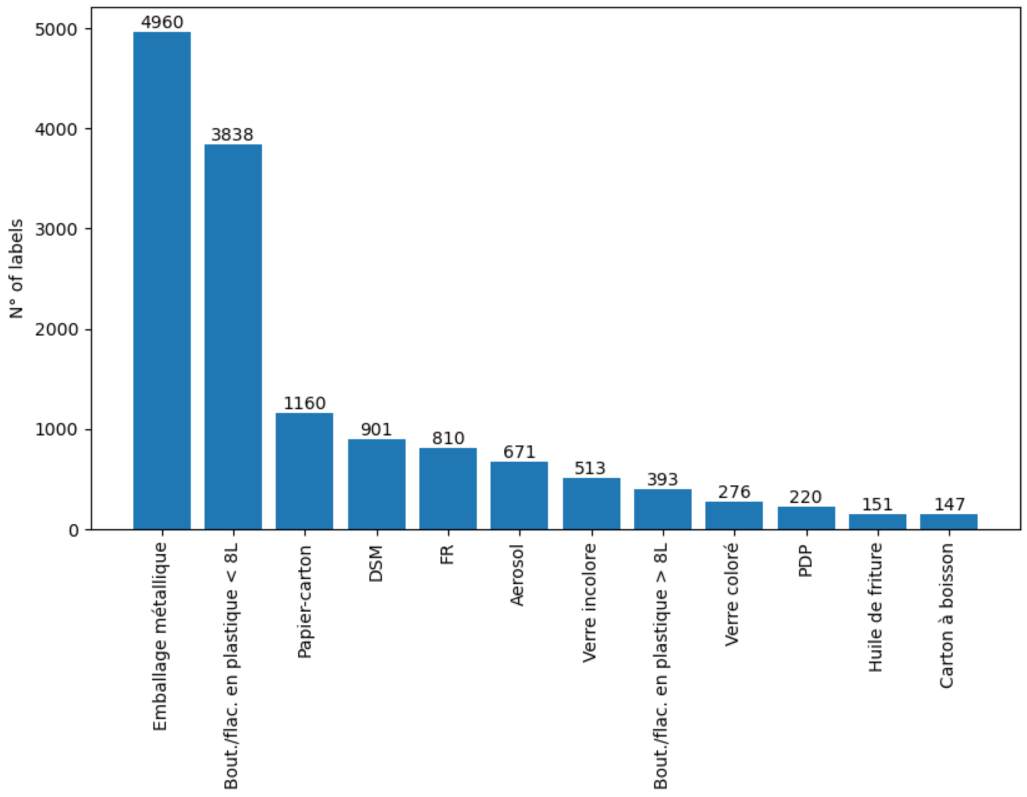


Figure 4.1: Class distribution of labels in the original dataset

distinct from one another, have steady labels, and for which enough 3D models can be freely found on common asset marketplaces. The set of classes was thus fixed after scraping through these marketplaces, and is admittedly somewhat arbitrary, but finds its rationale in the above observations. The resulting distribution of labels is depicted in Figure 4.2.

In the vast majority of cases, images containing the label of a given class do not contain labels of other classes. Hence, to break down the images into training, validation, and test sets, we consider the first label of each image and perform a stratified split according to it, allocating 60% of the images to training, 20% to validation and 20% to testing. Doing so, the class-wise proportions of labels in each set are roughly equal to the ones of the reduced dataset. The proportions are shown in Table 4.1.

	Bout./flac. en plastique < 8L	Aerosol	Verre coloré	PDP
Reduced dataset	83.01	9.77	4.02	3.20
Training set	83.70	9.59	4.01	2.70
Validation set	81.86	9.92	4.03	4.19
Test set	82.08	10.15	4.03	3.74

Table 4.1: Proportions (in %) of labels in the reduced dataset and in the training, validation and test set obtained through stratified sampling

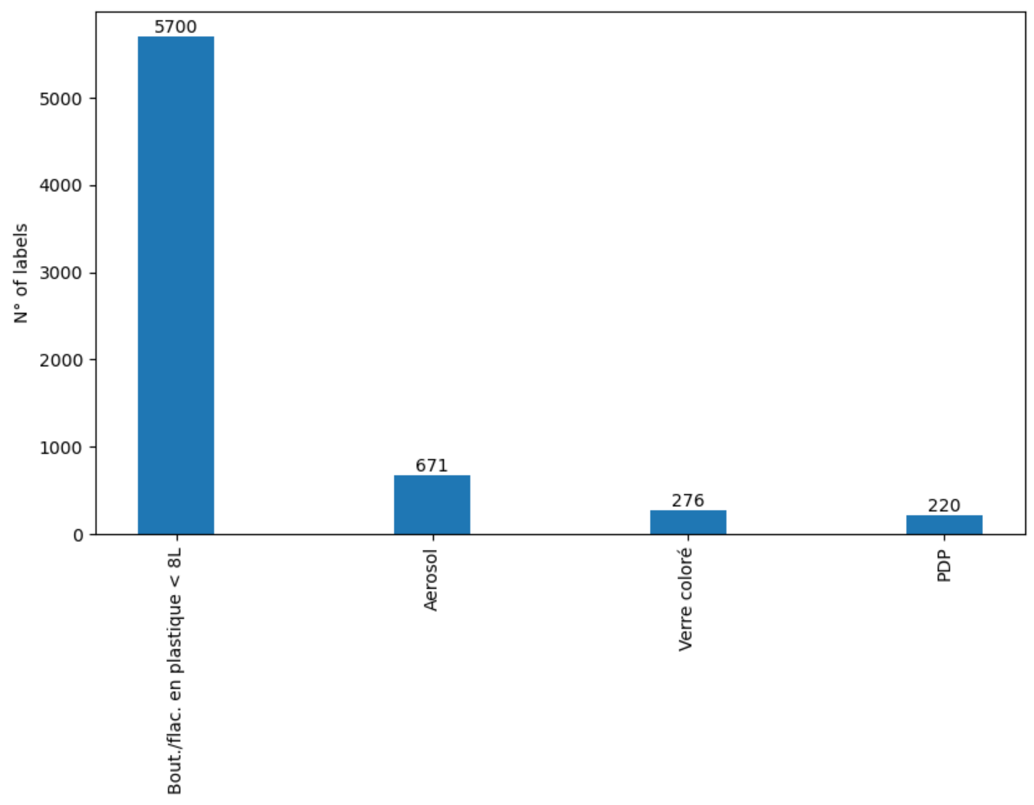


Figure 4.2: Class distribution of labels in the reduced dataset

4.2 Model, training and evaluation

Model

The object detection model used in this work is YOLOv5. Specifically, the open-source Ultralytics implementation [39] is used. YOLOv5 comes in differents sizes, which differ by depth and number of filters used in convolutional layers. YOLOv5s, the second smallest model which nonetheless comprises 7.2 million parameters, was chosen to ensure acceptable training times. Its architecture is detailed in Appendix C. Models are not trained from scratch; rather, the initial parameters are those obtained by pre-training the model on the 2017 version of the Common Objects in Context (COCO) dataset [67], which comprises 118 287 training images spanning 80 objects classes.

Hyper-parameters

Unless stated otherwise, the models in this work are trained using the procedure and hyper-parameters described hereafter. These are mostly kept to the default values provided by Ultralytics' implementation, which were found to perform best on the COCO dataset.

The training is warmed up for the first 100 batches. During this phase, the learning rate of the weights are linearly increased from 0 to 0.01, while the learning rate of the biases are linearly decreased from 0.1 to 0.01. Following warm-up, both learning rates are linearly decreased from 0.01 to 0.0001 over the remainder of the training.

The parameters are optimised using minibatch stochastic gradient descent using batches of size 32, on which the statistics are computed for the batch normalisations which follow convolutions. The optimiser uses Nesterov momentum, with momentum coefficient α being linearly increased from 0.8 to 0.937 during warm-up, and then kept constant for the remainder of training.

It uses the loss function described in equation (2.33), with parameters $\lambda_{box} = 0.05$, $\lambda_{cls} = 0.5$ and $\lambda_{conf} = 1$. Since images are resized to 640×640, detections are output at resolutions $\mathcal{R} = \{20 \times 20, 40 \times 40, 80 \times 80\}$, and their contributions in the confidence loss are respectively weighed by $\lambda_r = \{0.4, 1.0, 4.0\}$. Finally, the weights of convolutions are decayed with weight decay parameter $\lambda = 0.0005$. Neither the weights of batch normalisations nor any biases are decayed.

Data augmentation

Several common data augmentation techniques are used to slightly alter the training data. The colors of input training images are modified in their HSV[1] representation, by positively or negatively modifying uniformly at random the hue, saturation and value by respectively 0.015, 0.7 and 0.4 times their initial value. Once again uniformly at random, images are horizontally and vertically translated up to 10% of their size, and are scaled up or down up to 50% of their initial size. Images have a 50% chance of being horizontally flipped. Finally, mosaic data augmentation is used, which combines 4 input images at different scales into a single one. An example is given in Figure 4.3.

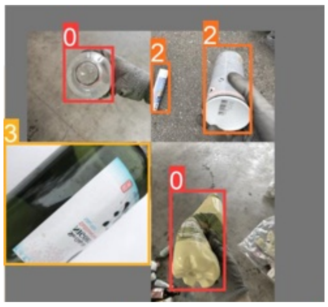


Figure 4.3: Mosaic data augmentation

Evaluation

All models trained are evaluated at each epoch on a validation set. During this evaluation, several performance metrics are computed. These are detailed in Appendix D. To determine what constitutes the best model, we use the mean average precision metric $\mathrm{mAP}_{0.5}$. The model which, during training, achieved the best $\mathrm{mAP}_{0.5}$ on the validation set is then evaluated on the test set. During validation and testing, the final bounding boxes are obtained through non maximum suppression with threshold $\mathrm{NMS}_t = 0.6$. Validation and test metrics are computed using thresholds $\mathrm{conf}_t =$

[1]Hue-Saturation-Value color model, an alternative to the RGB representation

0.25 and $\mathrm{IoU}_t = 0.5$.

Chapter 5

Synthetic data for object detection

In this chapter, we describe how we crafted the synthetic dataset in Section 5.1, and highlight the performance discrepancy between models trained synthetic and real data in Section 5.2. We analyse the frequency content of synthetic and real images to find out whether or not a filter could be applied to enhance our synthetic data in Section 5.3, and explore domain adaptation in Section 5.4. Finally, different training strategies are compared in Section 5.5.

5.1 Image synthesis

Synthesising images to train object detection models can be done in multiple ways, depending on the desired degree of synthesis. Applying data augmentations can be seen as a form of synthesis, as well as creating images by composing patches of real images. However, this requires real, annotated images to be available in the first place. In this work, the approach taken does not assume access to real data, but creates fully synthetic scenes using a 3D engine.

To generate our synthetic data, several general purpose 3D engines could be used. We chose to use Unity since they had recently released a package called Unity Perception [68], which greatly facilitates synthetic data generation. The package provides scripts which allow to label captures of generated scenes with perfect accuracy, and save them in easy to handle JSON files. Depending on the desired type of data, different types of labelers are available, such as semantic and instance segmentation, keypoints for pose estimation, or 2D and 3D bounding boxes. We will make use of the 2D bounding box labeler to create our synthetic object detection dataset. Conveniently, Unity Perception is also equipped with tools which support domain randomisation, which will be detailed in Section 5.1.2.

5.1.1 Asset collection

The creation of synthetic scenes requires access to 3D models and textures to be rendered. Background composition can be done using fairly generic assets, especially when taking advantage of the domain randomisation principle. When it comes to the foreground objects, these require more attention as they should intuitively have appropriate shapes, sizes, and textures such that the model can learn useful features and transfer to the real world. While, in theory, we could fabricate these assets from scratch, the design of quality models and textures has a steep learning curve and requires substantial artistic talent and time. In practice, we collected assets from existing projects which were suited to our use-case.

The SynthDet project [69] includes generic shapes such as cubes, cylinders and spheres which can be cluttered together to form irregular backgrounds. To cover these shapes, the project also provides textures obtained by cropping pictures of different fruits found in supermarket crates. We worked with these assets to create our backgrounds.

On the other hand, foreground models and textures which are a good fit to our dataset can hardly be found in a single project, and require a more thorough search. To this end, we manually scraped through 3D models marketplaces such as Sketchfab[1], CGTrader[2], Turbosquid[3], Free3D[4], and the Unity Asset Store[5] to find free assets. In total, 37 models and their textures were downloaded and rebuilt in Unity. Out of these, 20 of them corresponded to *Bout./flac. en plastique < 8L*, 4 to *Aerosol*, 4 to *Verre coloré*, and 9 to *PDP*. These are shown in Appendix F.

5.1.2 Simulation scenarios

To efficiently generate our synthetic data, the scene composition and capture should be automated. Unity Perception provides a scenario script which allows to create a given number of scenes and capture their appearance and labels. Scenarios are customised by adding components which successively place objects in the scene, and possibly modify them depending on the specified randomisations. In this work, we will start by using the domain randomisation principle and create increasingly randomised synthetic data. Each configuration described in the following section will be separately used as training data to our object detection model to determine which randomisation settings are useful to transfer as well as possible to the real domain. Regardless of randomisation settings, foreground objects are randomly placed at positions within a placement area using Poisson disk sampling, and randomly rotated around their axes by uniformly sampling an angle between 0° and 360° for each axis independently.

Default background

Our first and most simple iteration of synthetic data places the foreground objects in front of the default Unity scene background, without any distractors, as illustrated in Figure 5.1.

Background shape

Next, instead of having a constant background, we clutter it with randomly rotated shapes (Figure 5.2), producing edges which serve as primitive distractors.

[1]`https://sketchfab.com/store`
[2]`https://www.cgtrader.com/3d-models`
[3]`https://www.turbosquid.com/Search/3D-Models`
[4]`https://free3d.com/3d-models/`
[5]`https://assetstore.unity.com/3d`

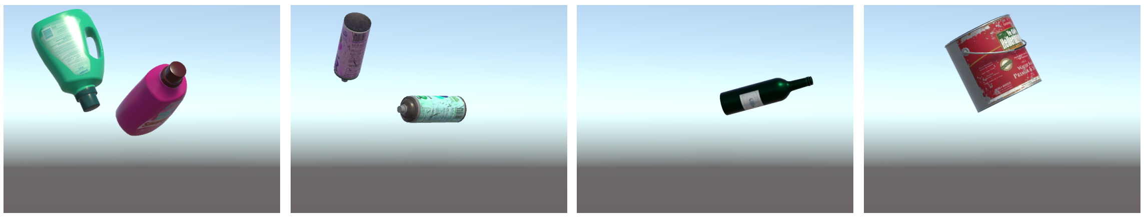


Figure 5.1: Synthetic data with default background

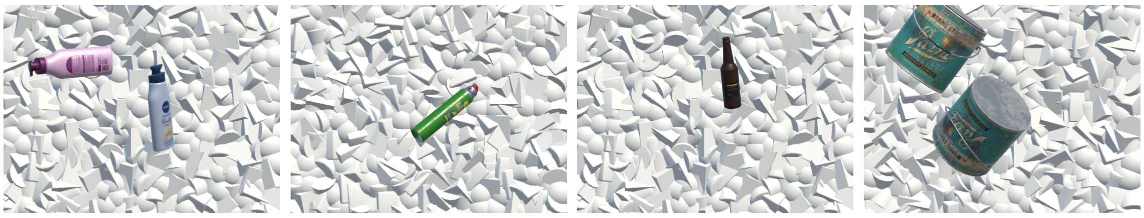


Figure 5.2: Synthetic data with a coarse background

Background texture

Still exploiting domain randomisation, we use the textures provided by SynthDet and apply them to the background clutter. Their hues are completely randomised (Figure 5.3).

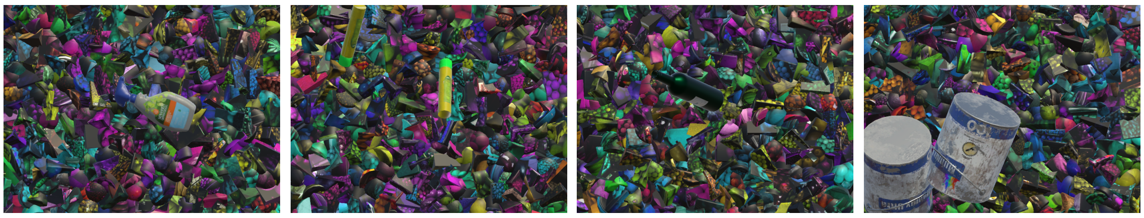


Figure 5.3: Synthetic data with textured background

Foreground object depth

Then, we uniformly vary the depth of the foreground objects relative to the camera instead of keeping it constant (Figure 5.4). This should help the model learn to detect objects at different scales, which is essential as the real images are not taken at a constant distance from the objects.

Lighting colour and intensity

Finally, since the lighting conditions may vary in the real data, we randomise them in the synthetic data. While the previous white-yellowish light was set to intensity 1 and RGB components (1, 0.96, 0.84), the new light source takes an intensity uniformly distributed between 0 and 1 and RGB components taken uniformly at random between 0.4 and 1 (Figure 5.5).

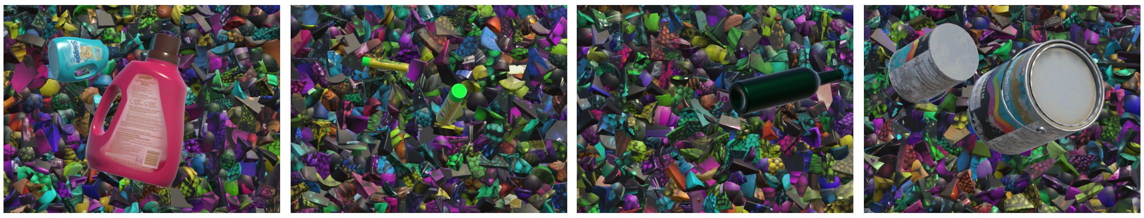


Figure 5.4: Synthetic data with objects placed at varying depths

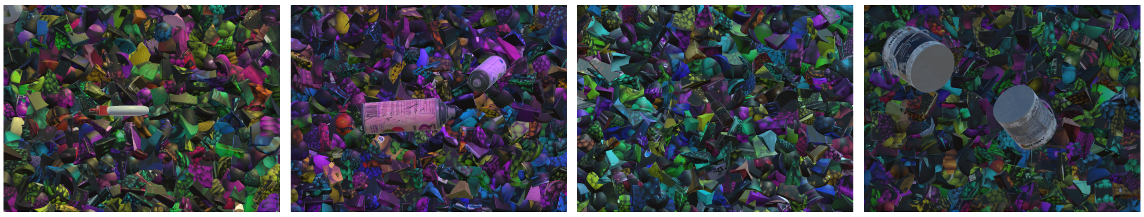
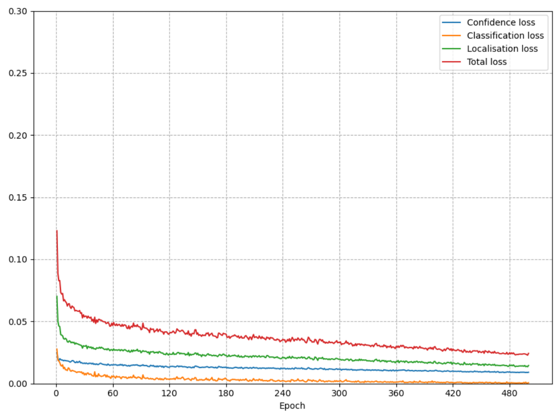


Figure 5.5: Synthetic data with varying lighting conditions

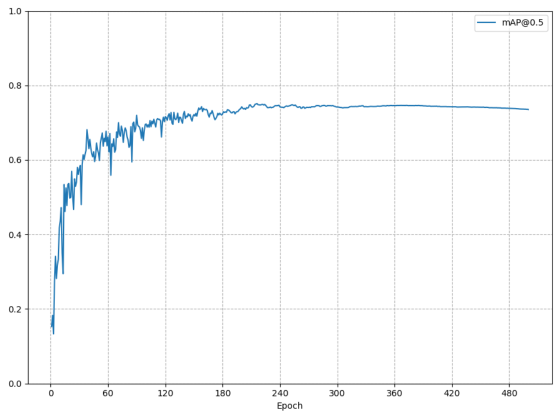
5.2 The sim-to-real gap

5.2.1 Baseline

We start by establishing our baseline performance by training on real data only, using the reduced dataset presented in Section 4.1 and the training settings described in Section 4.2. The number of epochs was initially arbitrarily set to 500, and found to be more than enough for the training to stabilise. We can observe in Figure 5.6 that the training loss decreases rather smoothly, and that the validation $\mathrm{mAP}_{0.5}$ increases with some oscillations for, roughly, the first 200 epochs, before stabilising. Further training does not improve the validation metric, and even slightly degrades it.



(a) Training loss



(b) Validation $\mathrm{mAP}_{0.5}$

Figure 5.6: Baseline training

The best performing model on the validation set is evaluated against the test set, for which the metrics are presented in Table 5.1.

5.2.2 Synthetic training data

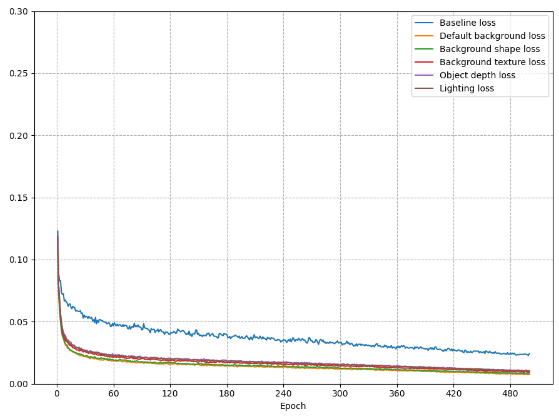
In the following experiments, we will use fully synthetic data to train our object detection models. For a fair comparison with the baseline, the number of images

Training data	$\text{mAP}_{0.5}$	$\text{mAP}_{0.5:0.95}$
Real data baseline	72.55	56.88

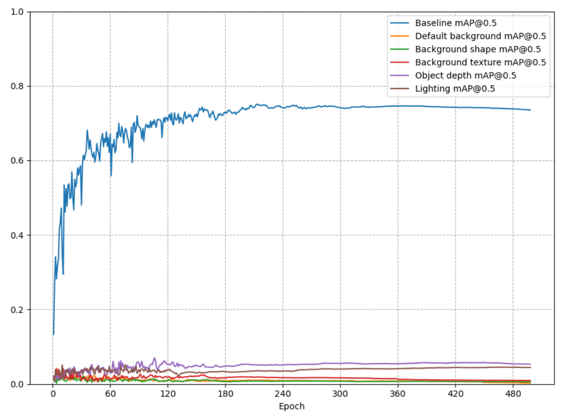
Table 5.1: Test metrics of the baseline

and label proportions should match those of the real training data. To this end and similarly to what we did in Section 4.1, we associate each image to a class and count the number of images per class. We generate as many class-wise synthetic images, placing each time between 1 and 2 foreground objects such that the number of labels in the real data is approximately matched. The validation and test datasets are the same as the ones used for the baseline.

We generate a training dataset for each variation of synthetic data listed in Section 5.1.2. We train the models using the same procedure, and present the training progress in Figure 5.7. We observe that the losses decrease in a stable fashion over the duration of the training, regardless of synthetic configuration. They reach values significantly below the baseline training loss, perhaps indicating that some features of the synthetic domain make it easier to learn. On the other hand, the models trained on synthetic data exhibit very poor performance on real data as captured by the validation $\text{mAP}_{0.5}$, regardless of synthetic data configuration. We observe that the 3 most basic configurations yield models with worse performance, and that in their case training on synthetic data is actually detrimental to their performance on real data. Varying the depths of the foreground objects brings a significant performance increase, while randomising the lighting conditions slightly degrades it.



(a) Total training loss for each configuration



(b) Validation $\text{mAP}_{0.5}$ for each configuration

Figure 5.7: Synthetic and baseline training

We evaluate the best performing model of each configuration on the test dataset and summarise the results in Table 5.2. Unsurprisingly, the test metrics of models trained on synthetic data are abysmal compared to the baseline. The configuration with varying object depth still performs best according to the $\text{mAP}_{0.5}$ metric, and has a marginally lower $\text{mAP}_{0.5:0.95}$ than the randomised lighting. We ultimately use the former to discriminate between the configurations' performances. In the remainder of the work, all further transformations of synthetic data will be done on the varying object depth configuration only, as it was deemed best according to the chosen metric.

Training data	$\text{mAP}_{0.5}$	$\text{mAP}_{0.5:0.95}$
Real data baseline	72.55	56.88
Default background	4.24	1.72
Background shape	5.59	1.86
Background texture	7.66	2.44
Object depth	**8.09**	2.69
Lighting	7.23	**2.74**

Table 5.2: Test metrics of the baseline and the different synthetic data configurations

5.3 Image content analysis and filtering

Following the observed discrepancies between models trained on real data and models trained on synthetic data, we want to investigate if the synthetic data can be enhanced to at least partially bridge this gap. Inspired by Frank *et al.* [70] who compared the frequency contents of real images and images synthetically created by generative adversarial networks, we examine the frequency contents of our synthetic data. If our analysis reveals that usual filters can alleviate shortcomings of our synthetic data to make it closer to real data in the frequency domain, we will apply these and use the resulting data to train our object detection model.

To explore the content of our images in the frequency domain, we apply the Discrete Cosine Transform, denoted DCT, to our training images. Mathematically, the DCT expresses a finite sequence of data points as a sum of cosine functions oscillating at different frequencies. In the case of an image $I = \{I_{x,y}\}$ of width W and height H, its DCT is given by $D = \{D_{k_x k_y}\}$ where

$$D_{k_x k_y} = 2 \sum_{x=0}^{W-1} \sum_{y=0}^{H-1} I_{x,y} \cos\left[\frac{\pi k_x}{W}\left(x + \frac{1}{2}\right)\right] \cos\left[\frac{\pi k_y}{H}\left(y + \frac{1}{2}\right)\right] \tag{5.1}$$

with $k_x = 0, ..., W-1$ and $k_y = 0, ..., H-1$. To compare the spectra of real and synthetic images, we will apply this transform to all synthetic training images and real training images. We will then average the transforms of the images of each domain separately.

Before applying the DCT, all RGB images are first converted to grayscale using a weighted average of the channels[6]. Then, they are resized to 640×640 using bilinear interpolation to reflect the way YOLOv5 pre-processes input images. Their DCT's are computed using SciPy's implementation, and averaged element-wise. The mean magnitude of synthetic and real spectra are depicted in Figure 5.8, where the origin is located in the top-left corner of the plot, and the horizontal direction corresponds to frequencies in the horizontal direction of the image, and likewise for the vertical

[6]Y = 0.299R+0.587G+0.114B where Y is the grayscale value.

direction. The plots are log-scaled since low values would skew the colour map and lead to poor visual representation.

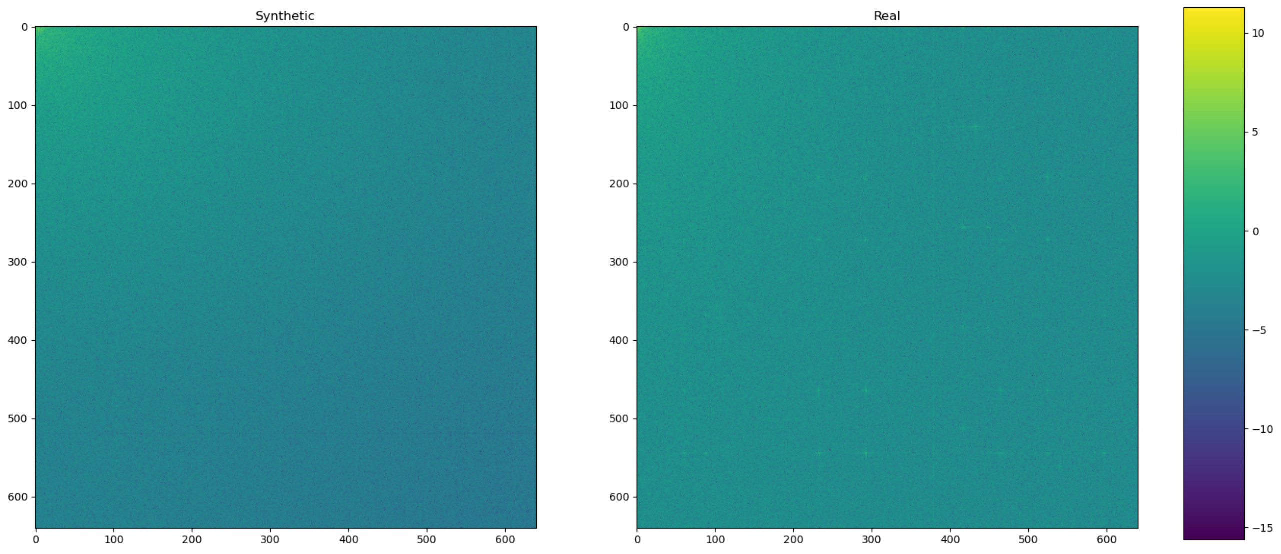


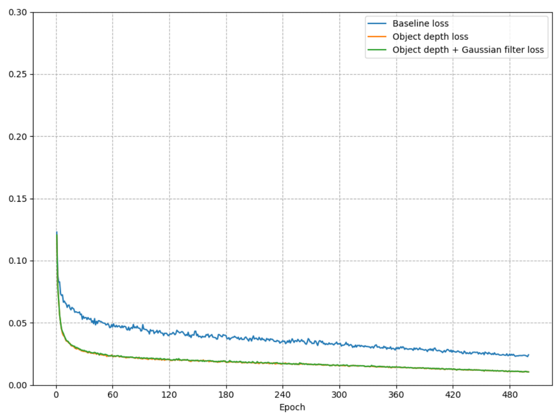
Figure 5.8: Synthetic and real spectra

The synthetic spectrum is quite concentrated towards low frequencies while the real spectrum appears to span low as well as higher frequencies, as indicated by the brightness being more evenly distributed. Unlike the synthetic spectrum, the real spectrum exhibits some high frequency components revealed by the bright dots which stand out from their neighbourhood. These observations entail that standard filters could hardly be used on the synthetic data to make their spectrum match the real spectrum, and in fact it is unlikely that any custom, carefully designed filter could reliably be applied. This being said, given the spectra observed in Figure 5.8, an interesting experiment could consist in training a model purely on synthetic data and apply a low pass filter to the real data at inference time to address the domain shift in the frequency domain. However, in this work we focus on enhancements that can be made to the synthetic training data, hence this remains future work.

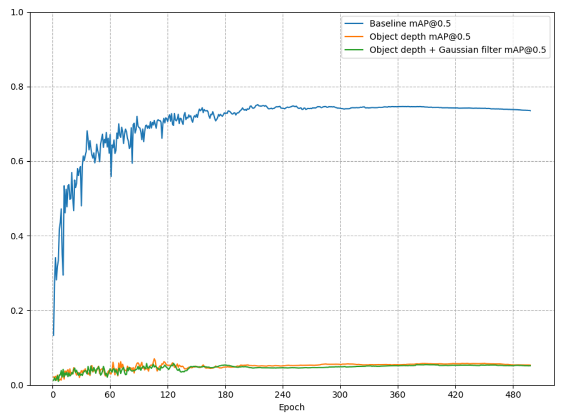
Then, inspired by Dwibedi et al. [50] who highlighted the existence of boundary artifacts when integrating foreground objects in their synthetic data and filtered their images to blend them in, we filter our synthetic images before using them as training data and observe the performance of the resulting model. For that purpose, we use a 3-by-3 Gaussian kernel

$$k = \begin{pmatrix} 0.0625 & 0.125 & 0.0625 \\ 0.125 & 0.25 & 0.125 \\ 0.0625 & 0.125 & 0.0625 \end{pmatrix}$$

and convolve it with the synthetic images. Admittedly, filtering an image with a Gaussian filter amounts to applying a low pass filter, and our observations in Figure 5.8 imply that this does not make the synthetic spectrum match the real one. However, we still elect to experiment it to see whether or not we observe improvements as Dwibedi *et al.* did, despite not making sense from a frequency perspective. Using the same training settings, we train a model with the filtered synthetic images. The results are presented in Figure 5.9 alongside the baseline and the unfiltered images. We observe that there are no noticeable differences or trends in the performance evolution of the filtered and unfiltered images, apart from the usual instability in the beginning of the training, and the validation $\text{mAP}_{0.5}$'s stabilise at very similar values.



(a) Total training loss for each configuration



(b) Validation $\text{mAP}_{0.5}$ for each configuration

Figure 5.9: Baseline, object depth and object depth followed by Gaussian filtering

Once again evaluating the best performing model on the test dataset, we present the results in Table 5.3. Gaussian filtering yields an ever so slightly higher $\text{mAP}_{0.5:0.95}$ metric, but a significant relative drop in $\text{mAP}_{0.5}$. Ultimately, this brings us to the conclusion that applying Gaussian filtering to our images does not improve performance, unlike Dwibedi *et al.* observed.

Training data	$\text{mAP}_{0.5}$	$\text{mAP}_{0.5:0.95}$
Real data baseline	72.55	56.88
Object depth	**8.09**	2.69
Object depth + Gaussian filtering	6.66	**2.82**

Table 5.3: Test metrics of the baseline, object depth and object depth followed by Gaussian filtering

5.4 Unsupervised image-to-image translation

Carrying on with our investigations, we slightly change the settings in which we assume the training of the object detection model to take place. Indeed, while previous experiments assumed access to synthetic data only, this section slightly adds onto this premise by granting access to unlabelled real data. To exploit unlabelled data, we will try to train a model to adaptively transform synthetic images to real-looking ones. Since the available synthetic and real images are unpaired, meaning that there are no direct correspondence between any two images where one would represent an object in the synthetic domain and the other would represent that same object in the real domain, we must select a model which does not required paired inputs to train. As mentioned in Chapter 3, such models include generative adversarial networks such as CycleGAN [60] and CUT [61], and we will attempt image-to-image translation with both of them. Both models are trained on all available synthetic and real images for 200 epochs using the default settings of the open-source implementations[7][8], which are also the ones used by the authors in their respective papers.

We visually present some representative samples of image adaptations using CUT in Figure 5.10. Indisputably, CUT fails to perform adequate domain adaptation. The foreground objects are either not found, displaced or deformed and modified beyond recognition, while the backgrounds do not exhibit any coherence with chaotic shapes and colours, as if patches had been somewhat randomly assembled. Moreover, CUT may adjust the size of objects to better match the training distribution, as illustrated in the leftmost image where the foreground object is tentatively scaled up. While a desirable property for some cases where domain adaptation entails changing the scale of objects, it is a significant drawback in our application as it invalidates bounding boxes. All in all, we find CUT to be unsuitable for image-to-image translation for object detection datasets.

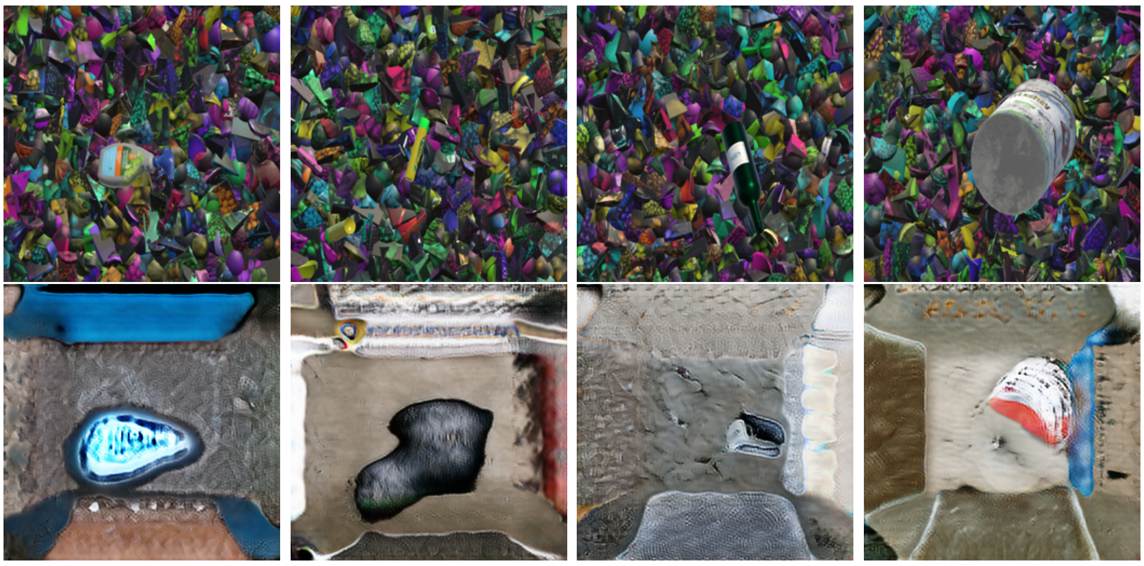


Figure 5.10: Samples of adaptations using CUT

We then present the failures, arguable successes and apparent successes of image-to-image translation using CycleGAN in Figures 5.11, 5.12 and 5.13 respectively. Additional samples can be found in Appendix G. The failed adaptations mostly include the foreground objects, albeit deformed or not prominently apparent, but surround them with incoherently mixed backgrounds. The arguable successes allow to distinguish quite distinctly the foreground objects, but these do not have clear-cut edges and seem to be leaking onto their surroundings. Finally, the apparent successes, produce quite coherent backgrounds and have well defined edges, although not perfect. While failures, arguable and apparent successes are illustrated here in equal proportions, it must be noted that failures actually severely outnumber successes when adapting the synthetic images. Hence, it is unlikely that the adapted images can serve as enhanced synthetic data and yield a better object detection

[7]https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix
[8]https://github.com/taesungp/contrastive-unpaired-translation

model than the raw synthetic data.

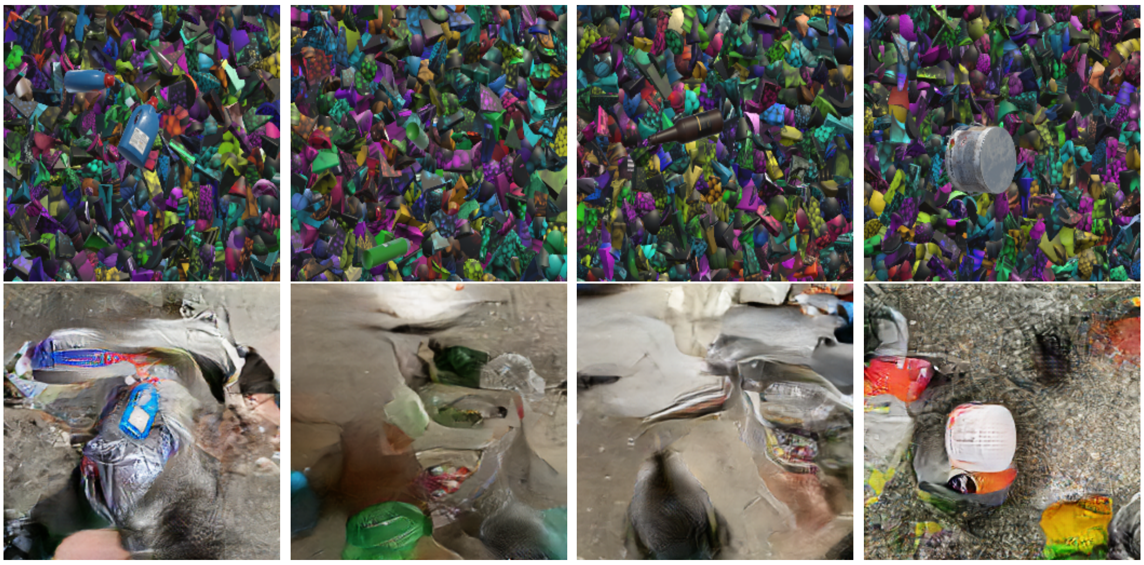


Figure 5.11: Samples of failed adaptations using CycleGAN

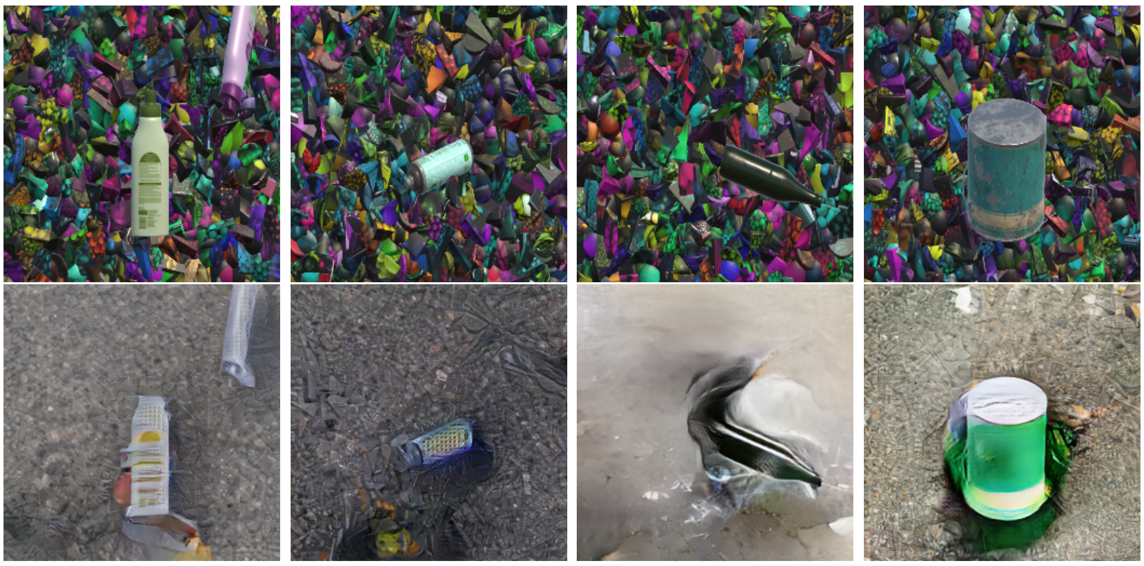


Figure 5.12: Samples of arguably successful adaptations using CycleGAN

Nevertheless, we train our object detection model using the CycleGAN-adapted synthetic data. The training progress is illustrated in Figure 5.14. As foretold, the model is harder to train and performs significantly worse than the model trained on raw synthetic data. Test results are presented in Table 5.4, and are unsurprisingly awful for CycleGAN-adapted synthetic data. The models trained with raw synthetic data with varying object depth outperforms models trained with tentatively enhanced synthetic data; hence, further experiments will be carried out using the former type of synthetic data.

Discussion

In this section, we have found generative adversarial networks to degrade the quality of our synthetic data rather than improve it, as they more frequently fail than succeed

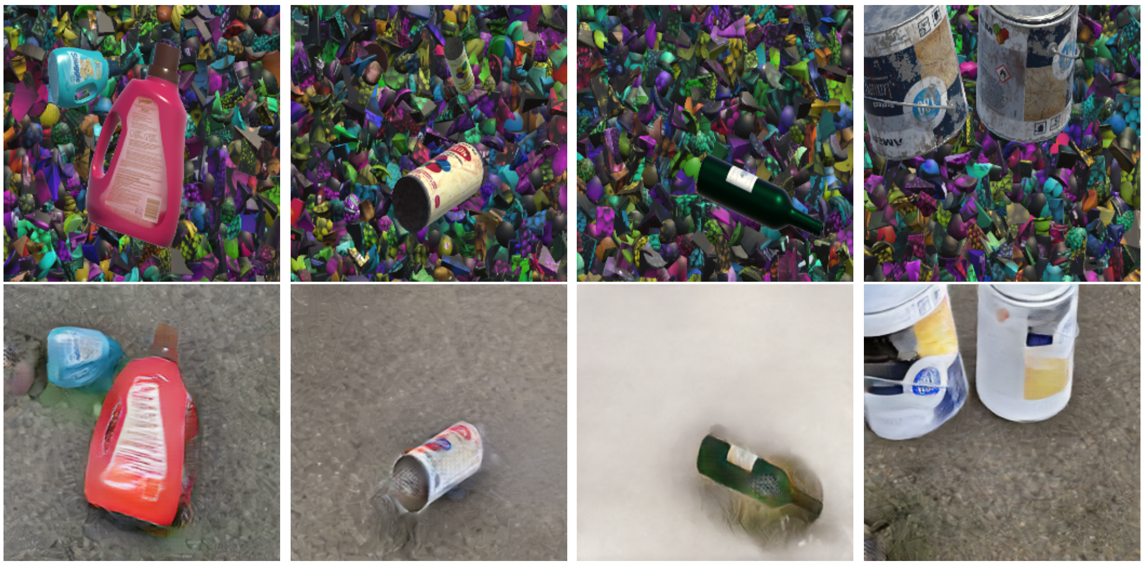
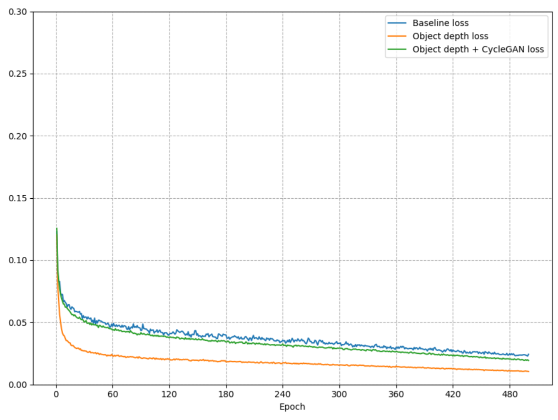
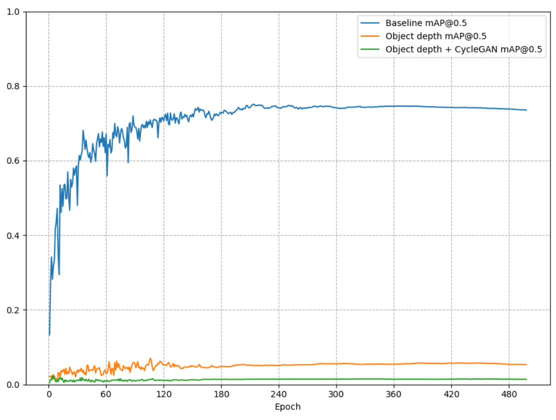


Figure 5.13: Samples of seemingly successful adaptations using CycleGAN



(a) Total training loss for each configuration



(b) Validation mAP$_{0.5}$ for each configuration

Figure 5.14: Baseline, object depth and object depth followed by CycleGAN adaptation

to perform domain adaptation. It arguably could have been tried to handpick what we consider to be successful domain adaptations, use these only as training data and observe the performance of the resulting object detection model. However, even if this endeavour did yield improved results compared to raw synthetic data, it would still imply that we cannot perform large-scale domain adaptation reliably since it would require timely human supervision. It is also worth noting that they would likely perform more consistently with photo-realistic synthetic data as the domain shift should be less extreme.

5.5 Training strategies

In the upcoming sections, we explore approaches to training object detection model in different settings. Section 5.5.1 assumes access to synthetic data only, while Sections 5.5.2 and 5.5.3 assume access to both synthetic and real, labelled data.

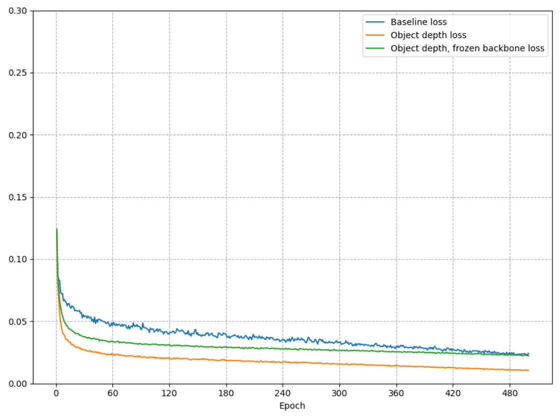
Training data	$\text{mAP}_{0.5}$	$\text{mAP}_{0.5:0.95}$
Real data baseline	72.55	56.88
Object depth	**8.09**	**2.69**
Object depth + CycleGAN	2.59	0.91

Table 5.4: Test metrics of the baseline, object depth and object depth followed by CycleGAN adaptation

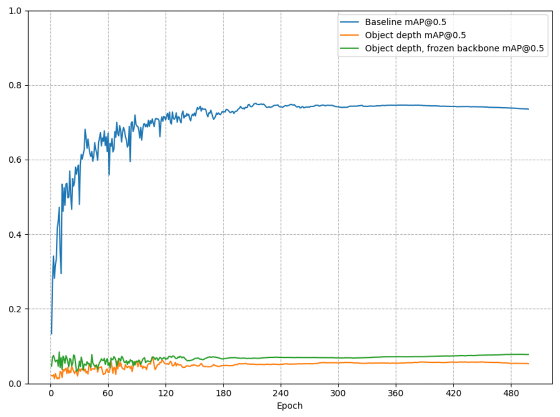
5.5.1 Freezing the backbone

A common practice in deep learning involves transferring knowledge by starting from weights which are pre-trained on another relevant task or dataset, and fine-tuning them on the downstream task and dataset. As mentioned in Section 4.2, this is the approach we take as we start from weights pre-trained on the COCO dataset before fine-tuning them. However, when training on synthetic data, we may wonder if fine-tuning the whole network is actually detrimental to the model's performance on real data. Indeed, weights pre-trained on large and varied real datasets such as COCO usually extract generic features which can be reused, and modifying these when training on synthetic data may perturb the features extractors and lead to decreased performance on real data. In the following experiment, we train YOLOv5 on synthetic data while freezing[9] its backbone and observe how the performance is affected compared to training the entire network.

The training progress is depicted in Figure 5.15. We can see that the training loss decreases less when freezing the backbone, indicating that the overall model is less fit to the synthetic data. The validation $\text{mAP}_{0.5}$ is slightly higher than its unfrozen counterpart, but still significantly lower than the baseline. The test metrics in Table 5.5 indicate that we cannot however conclude that freezing the backbone when training on synthetic data consistently yields better results, as the $\text{mAP}_{0.5:0.95}$ is slightly higher, but the $\text{mAP}_{0.5}$ is lower, making it difficult to draw any conclusion.



(a) Total training loss for each configuration



(b) Validation $\text{mAP}_{0.5}$ for each configuration

Figure 5.15: Baseline, object depth and object depth with frozen backbone

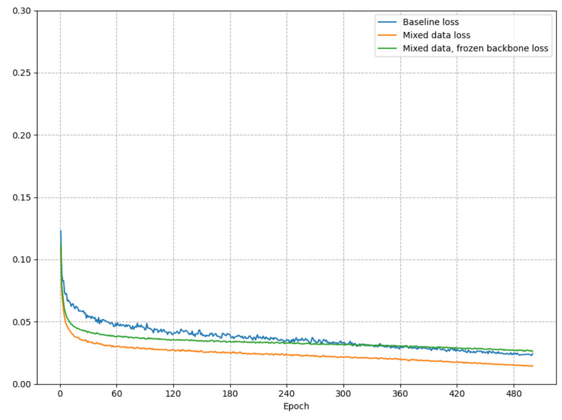
[9]Freezing parameters entails that their values remain unchanged, and only unfrozen parameters are updated during training

Training configuration	$mAP_{0.5}$	$mAP_{0.5:0.95}$
Real data baseline	72.55	56.88
Object depth	**8.09**	2.69
Object depth, frozen backbone	7.43	**2.96**

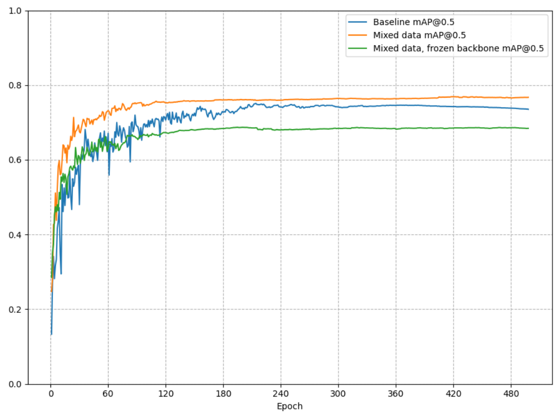
Table 5.5: Test metrics of the baseline, object depth and object depth with frozen backbone

5.5.2 Using mixed data

We now jointly use synthetic and real data during training. In the following experiments, we combine the synthetic and real training datasets together, such that the model is exposed to both types during training. We train one model using a frozen backbone, and another with an unfrozen one. The training results are presented in Figure 5.16, and indicate that using mixed data with a frozen backbone degrades performance, but allowing to fine-tune the whole network yields a higher validation $mAP_{0.5}$ compared to the baseline. We presume that mixing synthetic data adds variability and enhances the model's robustness.



(a) Total training loss for each configuration

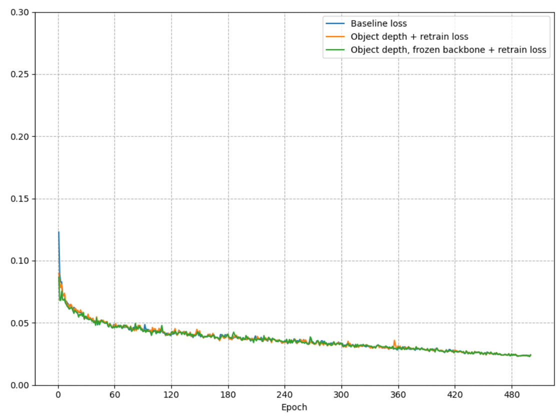


(b) Validation $mAP_{0.5}$ for each configuration

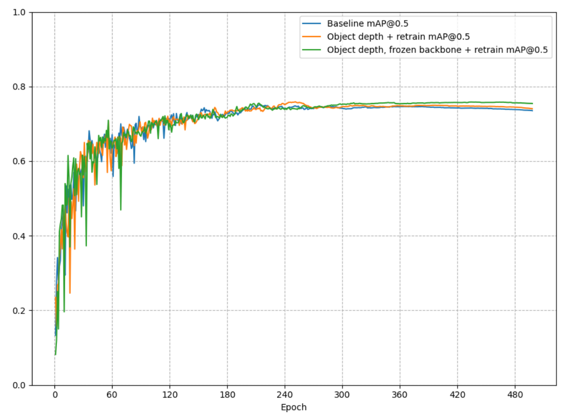
Figure 5.16: Baseline, mixed data and mixed data with frozen backbone

5.5.3 Pre-training using synthetic data

Rather than mixing both synthetic and real data during training, the following experiments start by training the model on the synthetic dataset only. The model which performed best on the validation set during training is then trained on the real dataset. We attempt this with one model having its backbone frozen, and another with its backbone unfrozen during synthetic data training. When re-training on real data, the backbone is always allowed to be fine-tuned. The training results are presented in Figure 5.17. The loss curves are nearly indistinguishable, and the validation $mAP_{0.5}$'s are too intertwined to reach any definite conclusion even though freezing the backbone on synthetic data does seem to bring slight improvements.



(a) Total training loss for each configuration



(b) Validation $mAP_{0.5}$ for each configuration

Figure 5.17: Baseline and re-training

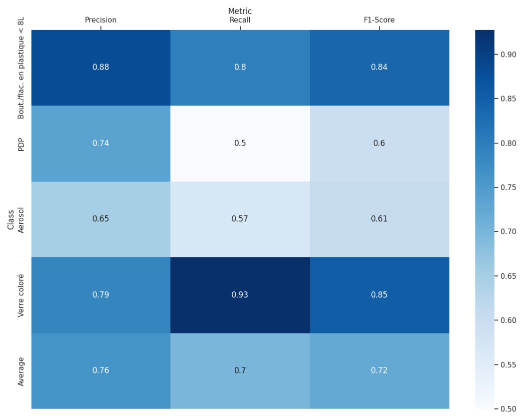
5.5.4 Best training strategy with synthetic and real data

We now summarise the test results in Table 5.6 to conclude as to which training configuration is best when both synthetic and real data are available.

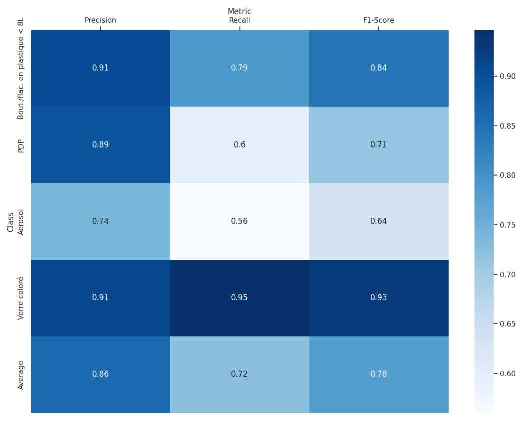
Training configuration	$mAP_{0.5}$	$mAP_{0.5:0.95}$
Real data baseline	72.55	56.88
Mixed data	**77.19**	**61.97**
Mixed data, frozen backbone	65.64	47.41
Object depth + re-train	73.83	58.21
Object depth, frozen backbone + re-train	75.71	60.09

Table 5.6: Test metrics of the baseline and different training configurations

It appears that exploiting both the synthetic and real data is almost always favourable compared to using the real data only, except if these are mixed and trained with a frozen backbone. Mixing them and allowing the whole model to be fine-tuned significantly boosts performance, leading to a relative increase of 6.49% in $mAP_{0.5}$ and 8.95% in $mAP_{0.5:0.95}$ compared to the baseline. We further detail the comparison by providing per-class and average precision, recall and F1-score in Figure 5.18. The average metrics are all improved, and it is especially noticeable for precision. This may be due to the hugely randomised backgrounds found in our synthetic data, which could bring robustness to the model by making it less likely to falsely detect background as foreground objects, hence decreasing the number of false positives.



(a) Baseline



(b) Mixed data

Figure 5.18: Baseline and mixed data detailed comparison

Chapter 6

Conclusion

In this work, we created synthetic data and investigated how it could be used on its own or alongside real data to train an object detection model. We created incrementally complex domain randomised synthetic data and found out that the configuration which varied the background's shape, texture and colour and the objects' depth was the one which transferred best when training on synthetic data and evaluating on real data. We compared the spectra of real and synthetic data and found that no obvious filtering could be done on the synthetic data to match the spectra. We then attempted to perform domain adaptation using generative adversarial networks, and found that these failed more often than not, making them unreliable for that purpose. Finally, we explored different training strategies : we found out that when using purely synthetic data, freezing the backbone did not reliably improve performance on real data, and that when both real and synthetic datasets are available, those should be combined and used to train the entire model without freezing any parameters. We acknowledge that these conclusions were reached for the specific dataset at hand and using our crafted synthetic data and that these may not necessarily always apply. However, we are fairly confident that the strategy consisting in mixing both types of data when available should hold true on other datasets.

Generalisation of the approach

The approach taken in this work in order to craft synthetic data was to scrape through marketplaces in order to find appropriate assets, which has a significant drawback : if the desired objects are very unusual or specific, they may very likely not be found in marketplaces, or at least not as free assets. In case these should be purchased, one should carefully assess whether the associated costs are worth it. Indeed, these could alternatively be invested to hire 3D artists to handcraft tailor-made assets, or into photogrammetry equipment if one has physical access to the objects which one aims to detect. Such a service is offered by Unity[1], although the pricing is not explicitly stated.

Future work

Regarding the generation of synthetic data, a few things could be explored in future work. First, the meshes of the 3D models representing the foreground objects could be deformed in order to vary the objects' shapes and make them appear crumbled,

[1]`https://blog.unity.com/technology/supercharge-your-computer-vision-models-with-synthetic-datasets-built-by-unity`

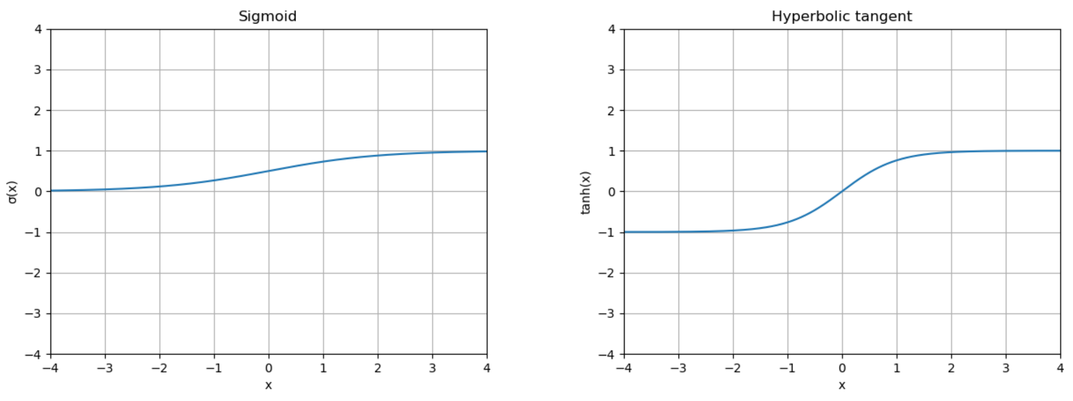
crushed or squashed. This would have been especially interesting for our waste detection dataset. However, performing these manually proved to be laborious and never quite gave satisfactory results, hence why they were not considered in this work. No mesh deformation plug-in could be found either, and the creation of such a plug-in would undoubtedly prove to be very challenging to make it work with any kind of object. Then, although there were some cases of partial occlusion in our synthetic data when two objects found themselves to be placed very close together, these remained rare, and crafting more of these would most likely help models detect objects surrounded by distractors or other objects. Finally, moving away from synthetic data crafted through 3D engines, we wonder if recent text-to-image diffusion models such as DALL-E[2], Midjourney[3] or Stable Diffusion[4], which have exhibited impressive visual results, could be used to create synthetic data. This is for instance what is done in the work of Ge *et al.* [71].

While we chose to generate domain randomised synthetic data, the creation of photorealistic synthetic data should also be explored when possible. In particular, photorealistic synthetic data may ease domain adaptation using generative adversarial networks, although failures would probably still occur. Future work could also explore other enhancements or domain adaptation techniques, as well as careful optimisation of training parameters. Indeed, we chose to set these and keep them unchanged throughout the different experiments, but it is entirely possible some tweaks to the hyper-parameters could yield better performing object detection models.

[2]`https://openai.com/dall-e-2/`
[3]`https://midjourney.com/home`
[4]`https://stability.ai/blog/stable-diffusion-public-release`

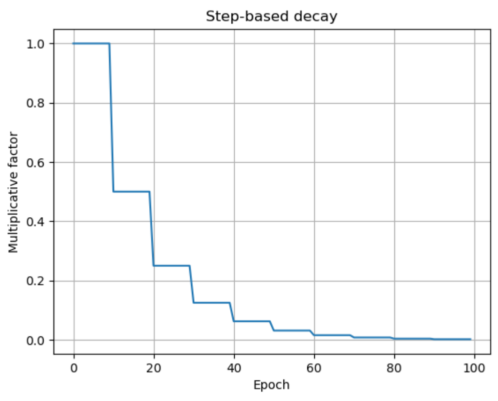
Appendix A

Background figures

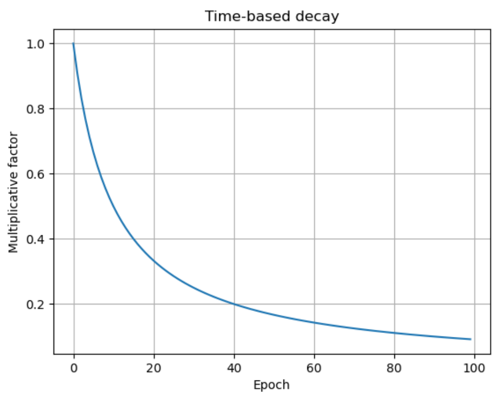


(a) $\sigma(x)$ **(b)** $\tanh(x)$

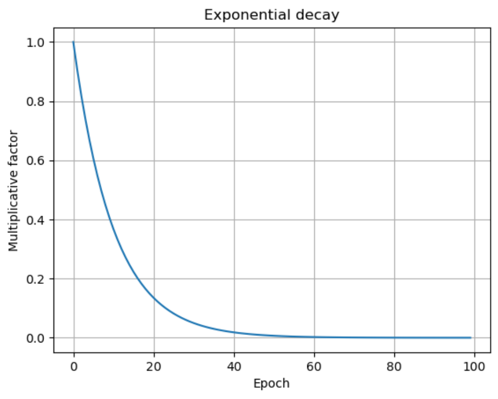
Figure A.1: Sigmoid and hyperbolic tangent activations



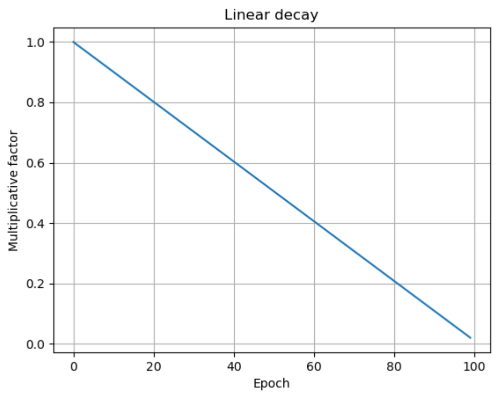
(a) Step-based decay with $k = 2$ and $n = 10$



(b) Time-based decay with $k = 0.1$



(c) Exponential decay with $k = 0.1$



(d) Linear decay with $l_f = 0.01$

Figure A.2: Common learning rate decay factors

Appendix B

Non maximum suppression

The predictions of an object detection model may detect an object multiple times through different bounding boxes, and as such be redundant. To mitigate this issue, models often rely on non-maximum suppression to avoid multiple detections of the same object. Non-maximum suppression merges detections which hold the same predicted class and have significant overlap. Greedy non maximum-suppression, which is the standard implementation found in most object detection models, proceeds in the following way for each class :

1. Let the set $\mathcal{B}$ hold all predicted bounding boxes of the class, and $\mathcal{S}$ the corresponding predicted confidences. Let the set $\mathcal{D}$, initially empty, hold the final detections.

2. The predicted bounding box with highest confidence is removed from $\mathcal{B}$ and added to $\mathcal{D}$. The IoU scores of that bounding box with all other bounding boxes left in $\mathcal{B}$ are computed. If an IoU score is greater than a threshold NMS_t, then the corresponding bounding box is removed altogether from $\mathcal{B}$.

3. Step 2 is repeated until $\mathcal{B}$ is empty.

Appendix C

Object detection model summary

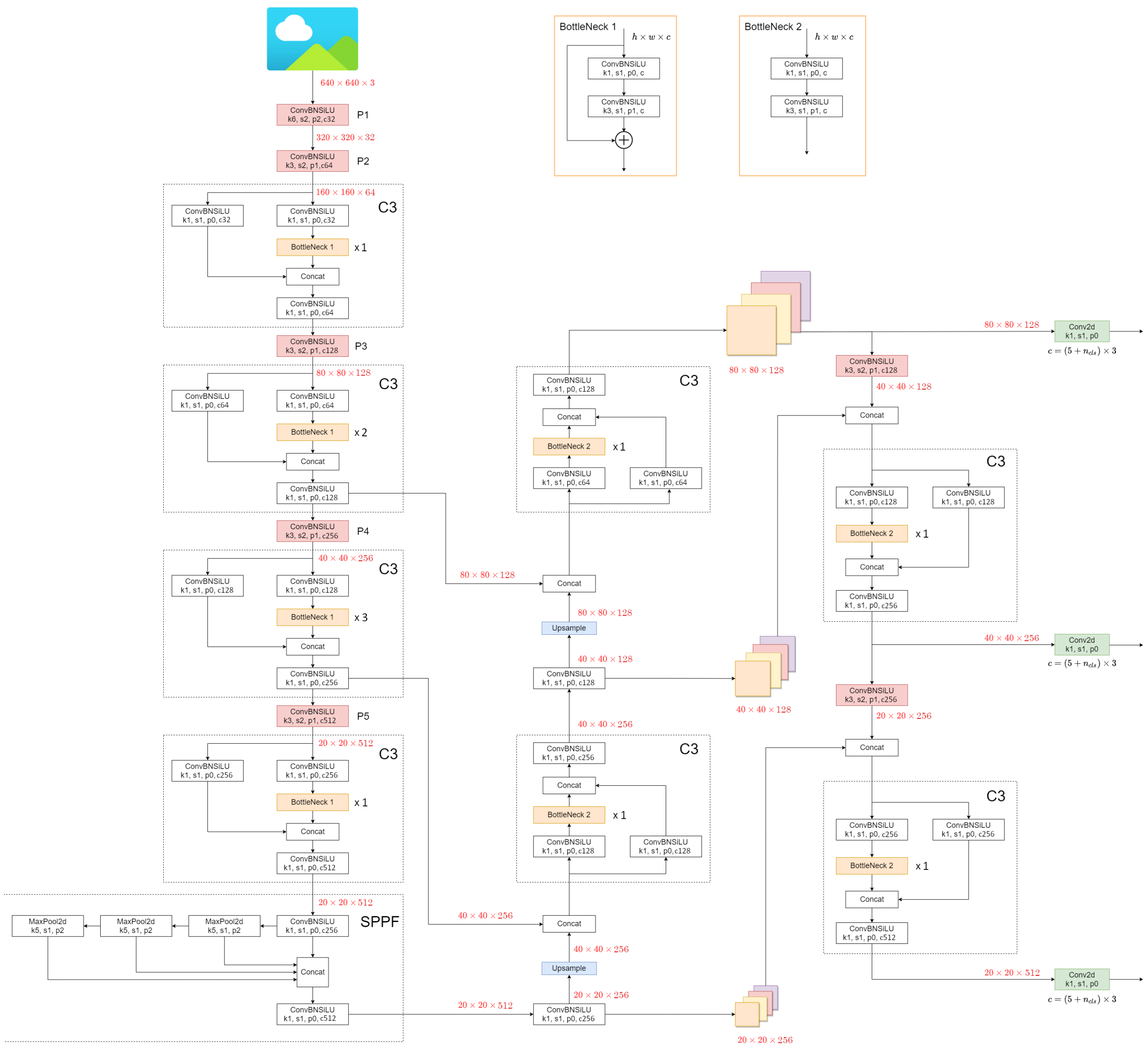


Figure C.1: Summary of YOLOv5s. Credits to Wu Zhe from which the illustration was adapted.[1]

YOLOv5 takes an input image and outputs detections at 3 resolutions which are respectively 8, 16 and 32 times smaller than the resolution of the input image. Figure C.1 illustrates the model's inner workings for a 640 × 640 input image, and hence outputs detections at resolutions $\mathcal{R} = \left\{80 \times 80, 40 \times 40, 20 \times 20\right\}$. In practice, inputs can have any resolution and are resized through bilinear interpolation before being fed to the model.

Convolutional layers are denoted by *ConvBNSiLU* to indicate that the convolutions are followed by batch normalisation and SiLU activation. The letters and numbers correspond to the parameters of the convolution layer. For instance, (k3, s2, p1, c64) indicates that the layer comprises 64 convolutional filters with 3 × 3 kernels, (2,2) stride and (1,1) padding.

The architecture's backbone alternates convolutions which progressively decrease the spatial dimension and increase the depth of the feature maps, and C3 blocks which are simplified CSP bottlenecks. While the idea of concatenating creating 2 paths for feature maps and concatenating the results at the end of these paths is certainly inspired by CSPNet, YOLOv5 does not actually split the feature map channels in 2, but rather duplicates them. C3 blocks include one or more residual bottleneck blocks BottleNeck 1 in series.

The neck's SPPF block is mathematically equivalent to an SPP block, but its implementation is optimised to use less floating point operations and is consequently much faster. The PANet-like passthrough connections allow to concatenate feature maps from earlier layers in the backbone with feature maps from the neck by upsampling the latter through nearest-neighbour interpolation.

Finally, the detection head uses feature maps at different stages of the neck to output detections at 3 different resolutions. The 3 detection layers use convolutional layers with 1×1 kernels, unit stride and no padding, and $3 \times (5 + C)$ filters as there are 3 anchor boxes per grid cell, followed by sigmoid activations.

YOLOv5 comes in multiple sizes, which differ by the number of filters used in convolutional layers, and the number of BottleNeck blocks in C3 blocks.

[1]`https://github.com/ultralytics/yolov5/issues/6998`

Appendix D

Object detection metrics

In a binary classification task, one usually computes the amount of true positives *TP*, true negatives *TN*, false positives *FP* and false negatives *FN* to evaluate a model. These 4 values together form what is referred to as the confusion matrix.

		Predicted	
		Positive	Negative
Actual	Positive	TP	FN
	Negative	FP	TN

Table D.1: Confusion matrix in binary classification

Extending to multi-class classification, the same table can be computed for each class separately. In object detection, the same principle can be used, but determining TP, FP, TN and FN is slightly more involved.

D.1 TP, FP, TN, and FN for bounding boxes

These different quantities are computed per class. For each class, any detection such that the predicted probability is the highest for that class, and such that the associated confidence is larger than a chosen threshold conf_t, is considered.

- TP : A correct detection, i.e. such that the IoU between the ground truth and the predicted bounding box is $\geq \text{IoU}_t$
- FP : An incorrect detection, i.e. such that the IoU between the ground truth and the predicted bounding box is $< \text{IoU}_t$
- FN : A ground truth for which no predicted bounding box has an IoU $\geq IoU_t$
- TN : Not applicable in object detection, because it would represent the background regions that were correctly not detected as objects, and one could define an infinite number of bounding boxes around background regions

D.2 Precision, recall and F1-score

Different metrics can be defined to capture different aspects of the model's performance by combining the quantities previously defined.

Precision

Precision portrays the ability of the model to detect relevant objects, and those only. As such, it is given by

$$\text{Precision} = \frac{TP}{TP + FP} \tag{D.1}$$

Recall

On the other hand, recall portrays the ability of the model to detect all relevant objects and is therefore given by

$$\text{Recall} = \frac{TP}{TP + FN} \tag{D.2}$$

F1-score

The F1-score is the harmonic mean between the precision and recall :

$$F1 = 2\frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{D.3}$$

It represents the trade-off implicitly decided by setting the confidence threshold to its value, since precision and recall are inversely affected by changes in this threshold, *i.e.* a high confidence threshold favours high precision and low recall, while a low confidence threshold favours low precision and high recall.

D.3 Average precision

Average precision, denoted AP, is a metric based on the precision-recall curve. A concise summary of the curve can be obtained by computing the area under it. This area is the average precision, and allows clear comparison of the general performance of different object detection models.

The precision recall curve is obtained by sorting every detection of the model in decreasing order of confidence. Precision and recall can then be computed for each detection in decreasing order of confidence, by accumulating the previous TP and FP. The average precision is then computed by interpolating the precision at a predefined, discrete number of equally-spaced recall levels. In YOLOv5, the average precision is computed under the COCO guidelines, by interpolating the precision p_{interp} at 101 recall levels r, such that

$$AP = \frac{1}{101} \sum_{r \in \{0, 0.01, \ldots, 1\}} p_{interp}(r) \tag{D.4}$$

with

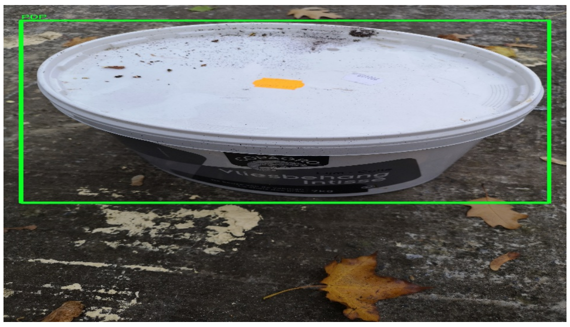
$$p_{interp}(r) = \max_{\tilde{r} \geq r} p(r) \tag{D.5}$$

and where $p(r)$ denotes the precision at recall r.

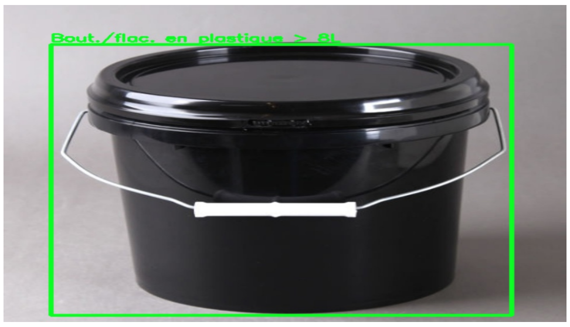
The average precision can be computed for each class. Averaging over all classes yields the mean average precision, denoted mAP. This metric is the most widely used in object detection, but several versions of it coexist. Indeed, the mean average precision depends on the threshold IoU_t used to discriminate correct from incorrect detections. The most straightforward one is the $\text{mAP}_{0.5}$, which uses $\text{IoU}_t = 0.5$. Another popular version is the $\text{mAP}_{0.5:0.95}$, which is computed by taking the mean average precision at 10 equally spaced thresholds $\text{IoU}_t = [0.5, 0.55, ..., 0.95]$, and averaging these.

Appendix E

Dataset



(a) PDP



(b) Bout./flac. en plastique > 8L

Figure E.1: Similarity between PDP and Bout./flac. en plastique > 8L

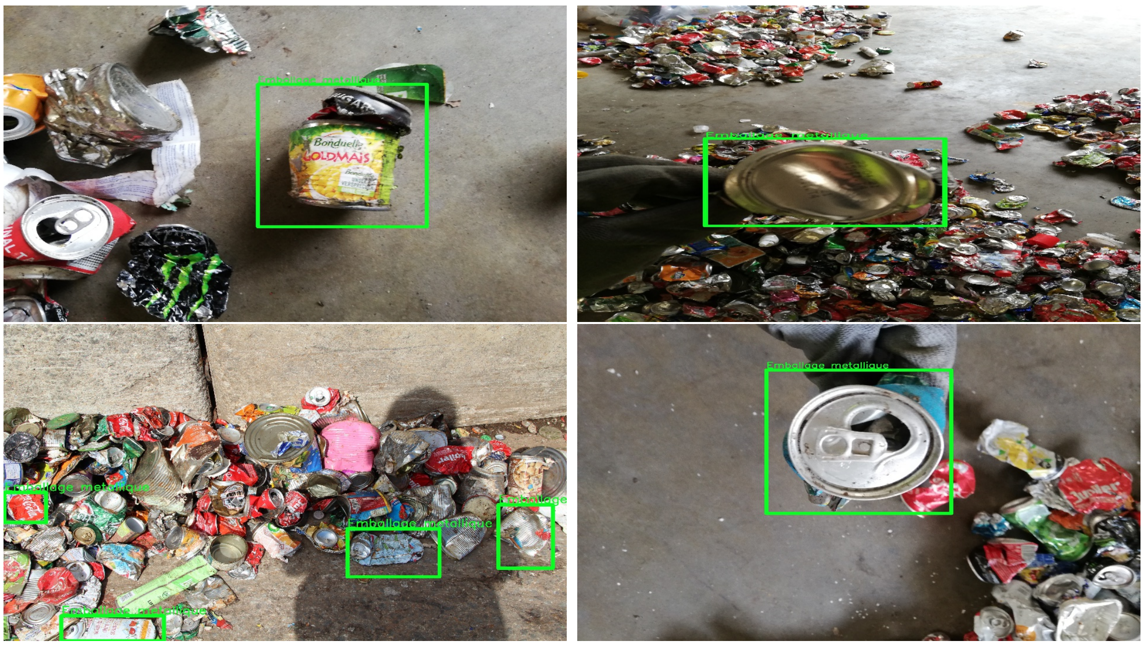
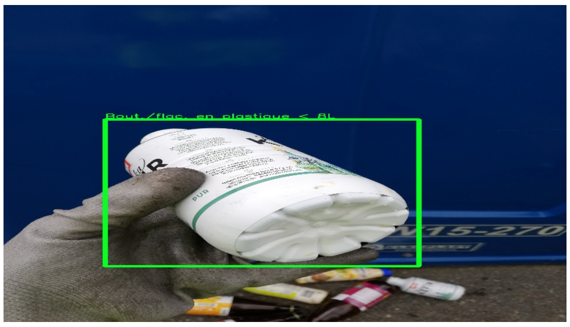
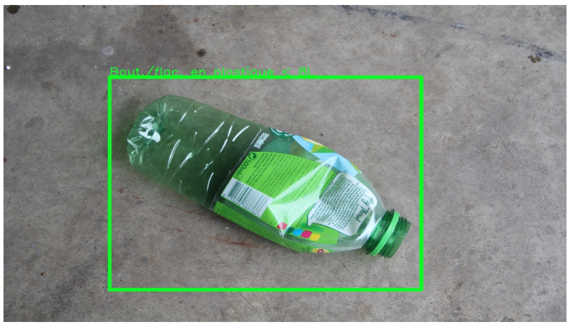


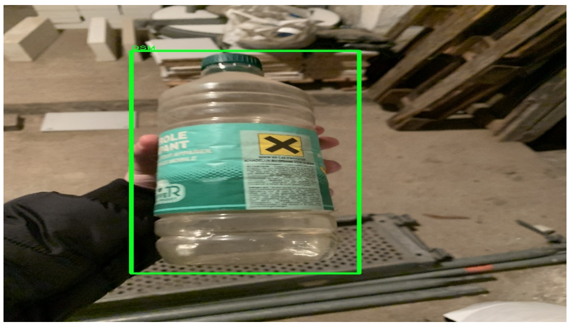
Figure E.2: Arbitrary labels for Emballage métallique



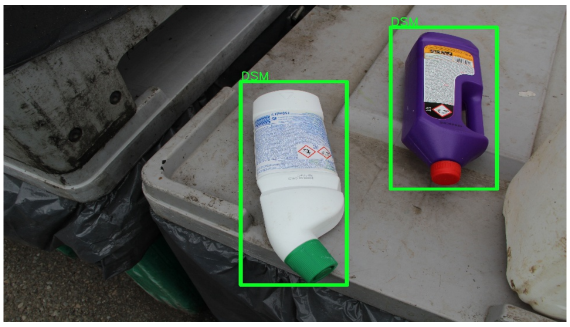
(a) Bout./flac. en plastique < 8L



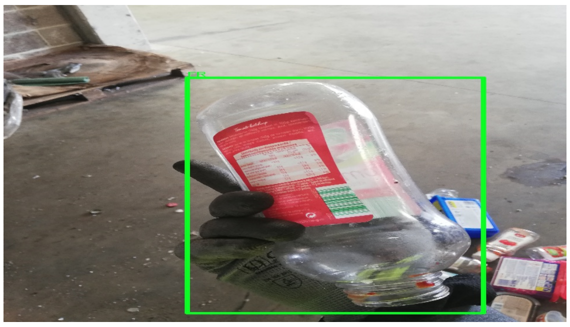
(b) Bout./flac. en plastique < 8L



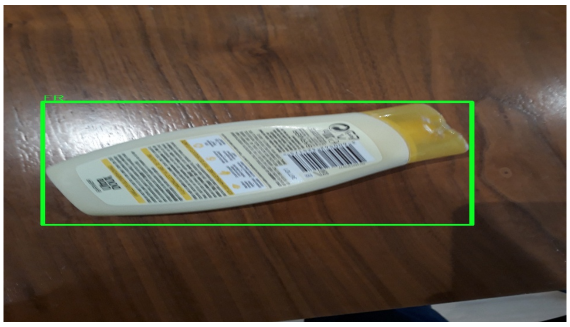
(c) DSM



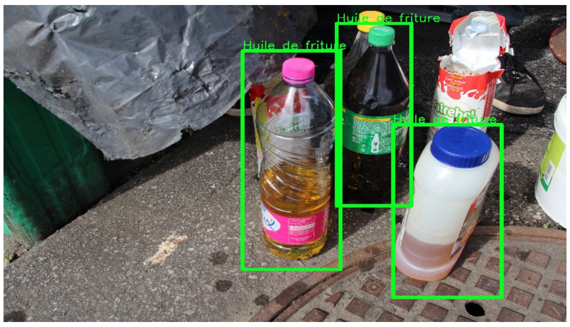
(d) DSM



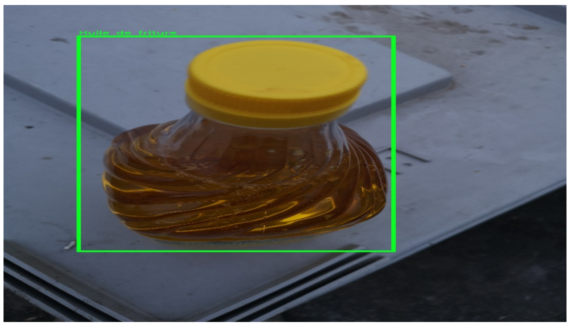
(e) FR



(f) FR



(g) Huile de friture



(h) Huile de friture

Figure E.3: Similarity of some classes with Bout./flac. en plastique < 8L

Appendix F

3D models of foreground objects

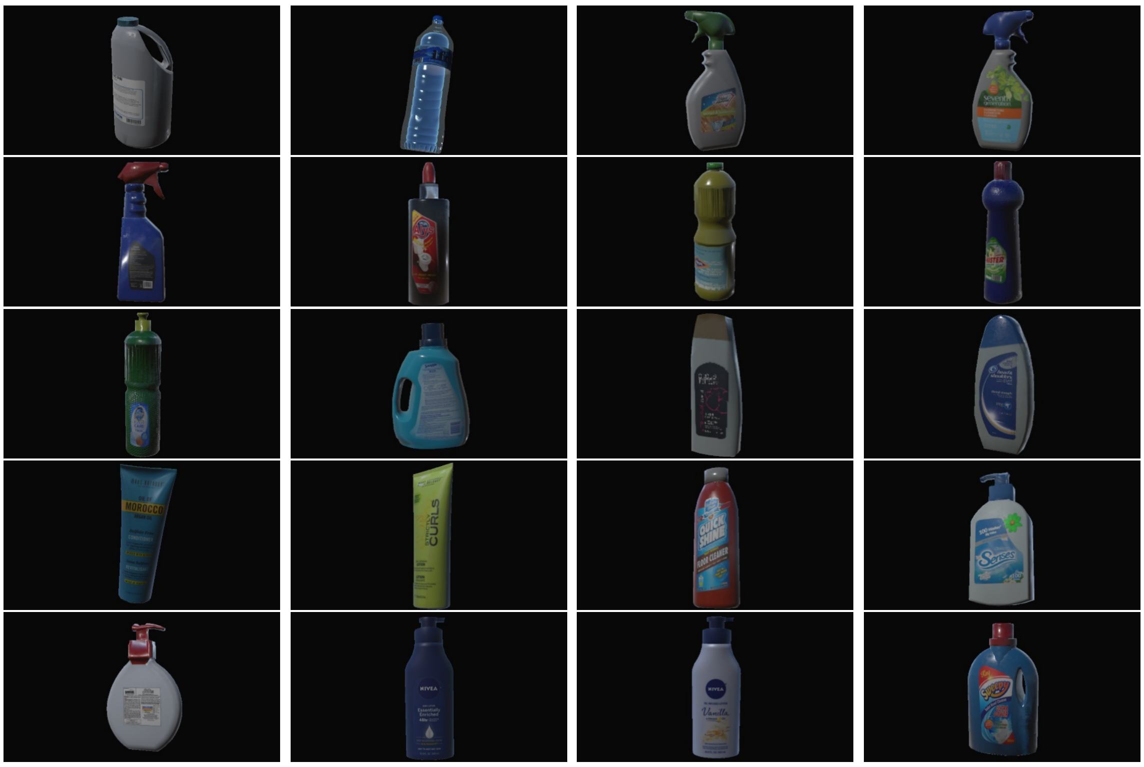


Figure F.1: 3D models for Bout./flac. en plastique < 8L

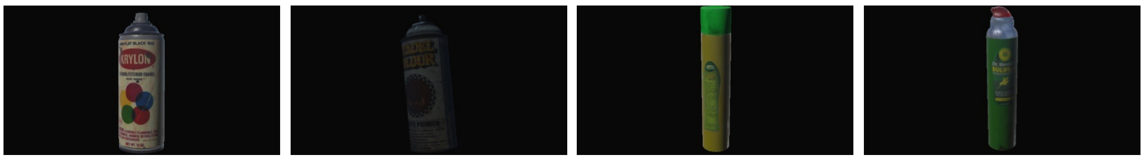


Figure F.2: 3D models for Aerosol

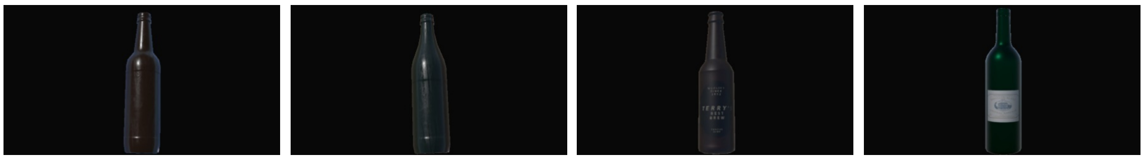


Figure F.3: 3D models for Verre coloré

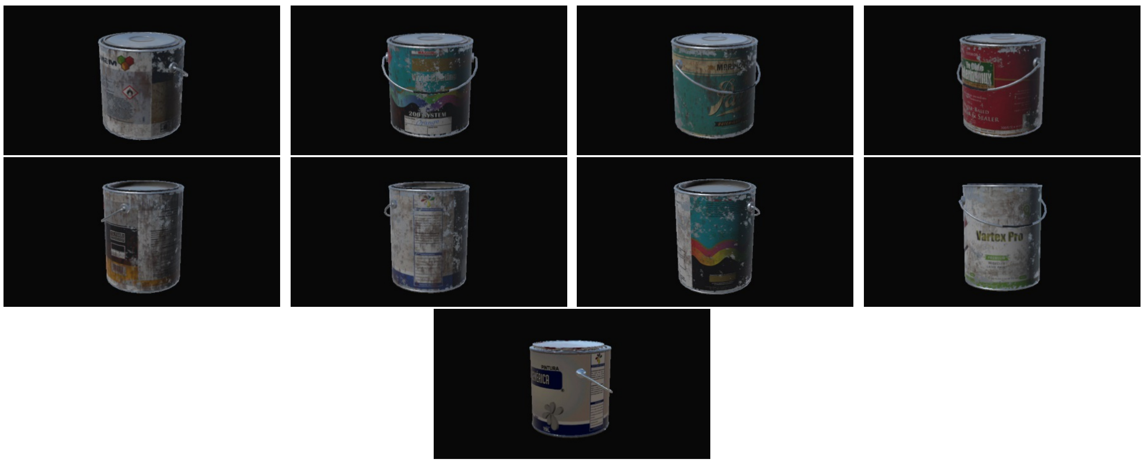


Figure F.4: 3D models for PDP

Appendix G

Additional samples of adaptation using CycleGAN

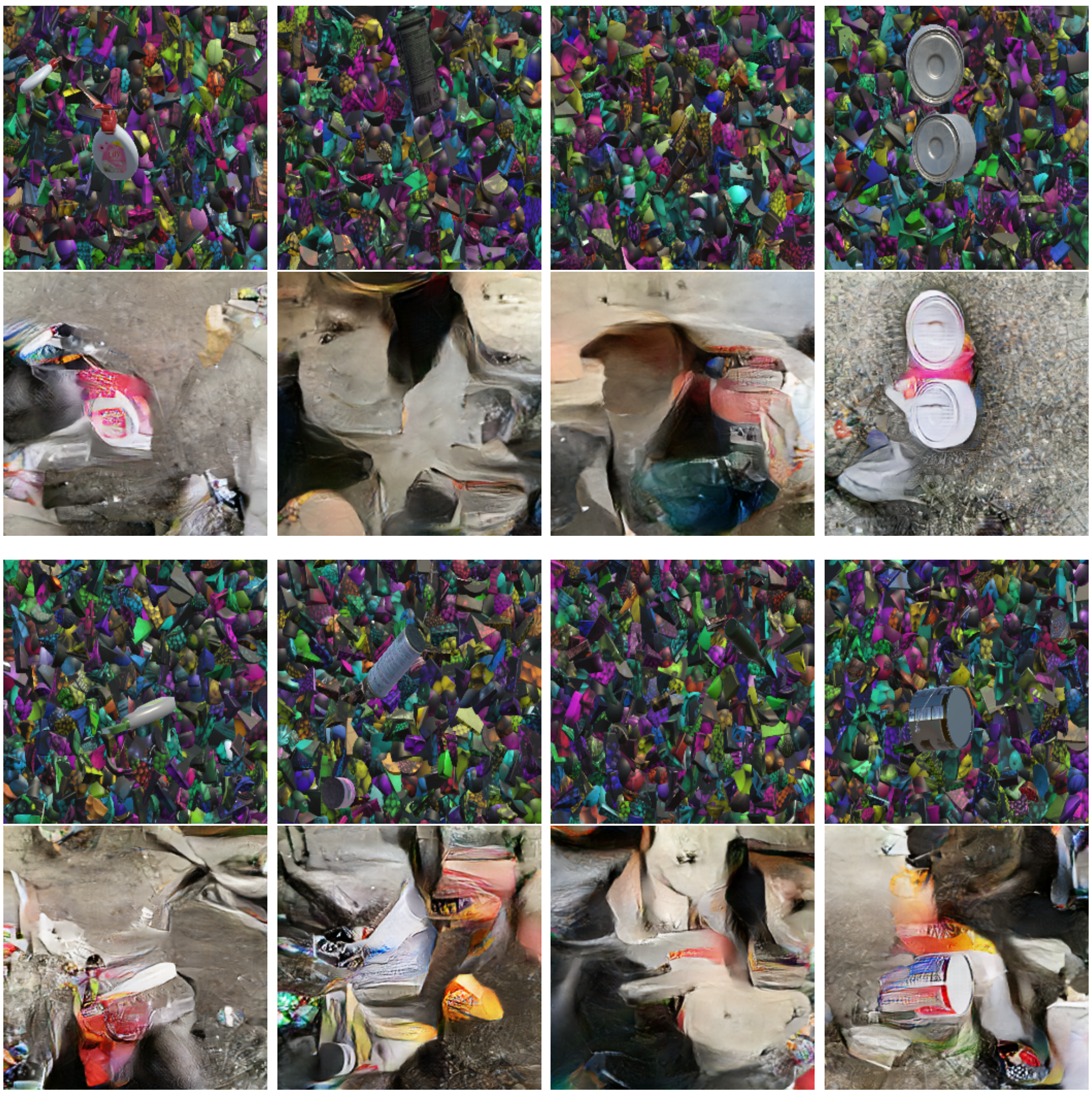


Figure G.1: Additional samples of failed adaptations using CycleGAN

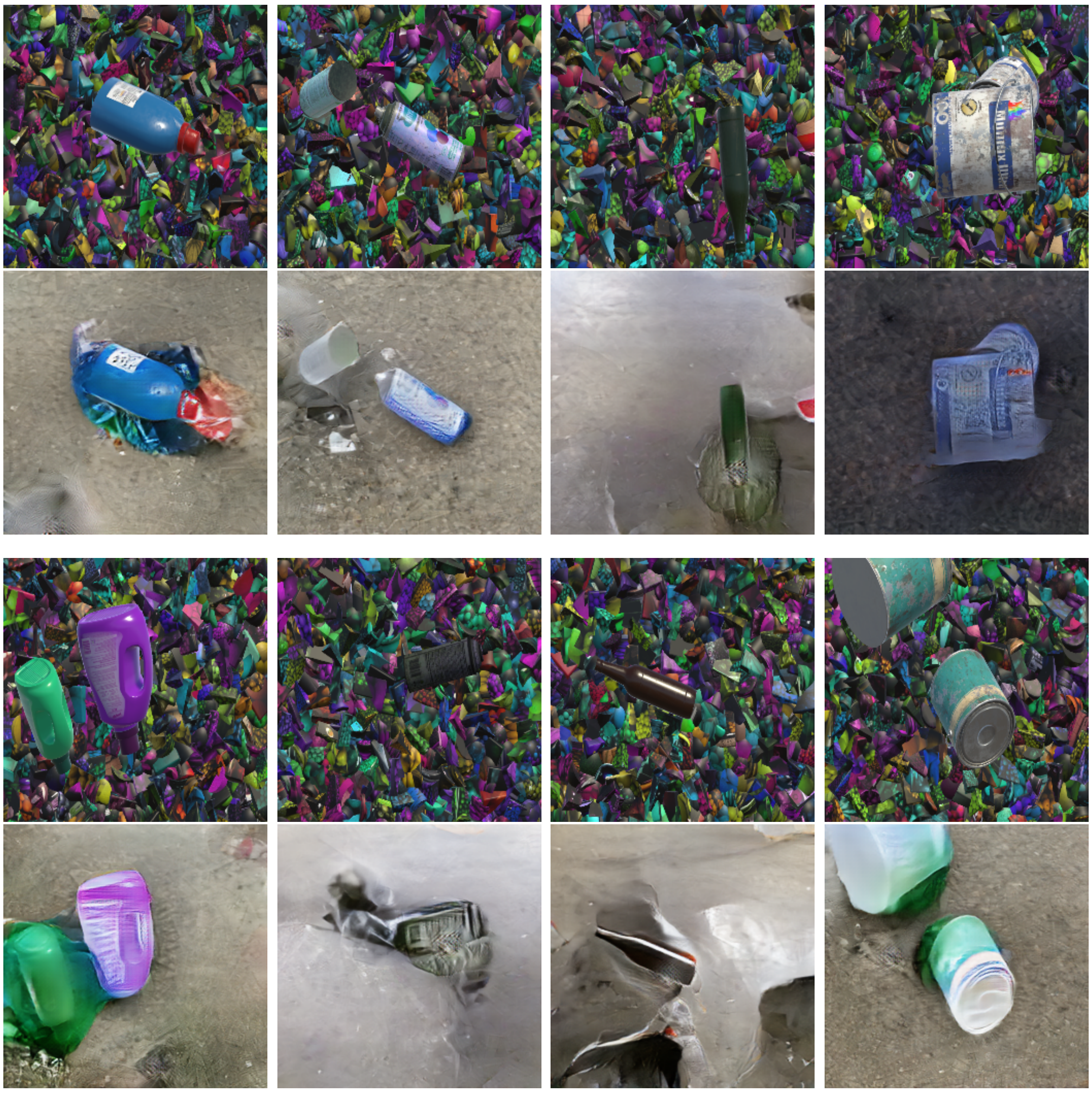


Figure G.2: Additional samples of arguably successful adaptations using CycleGAN

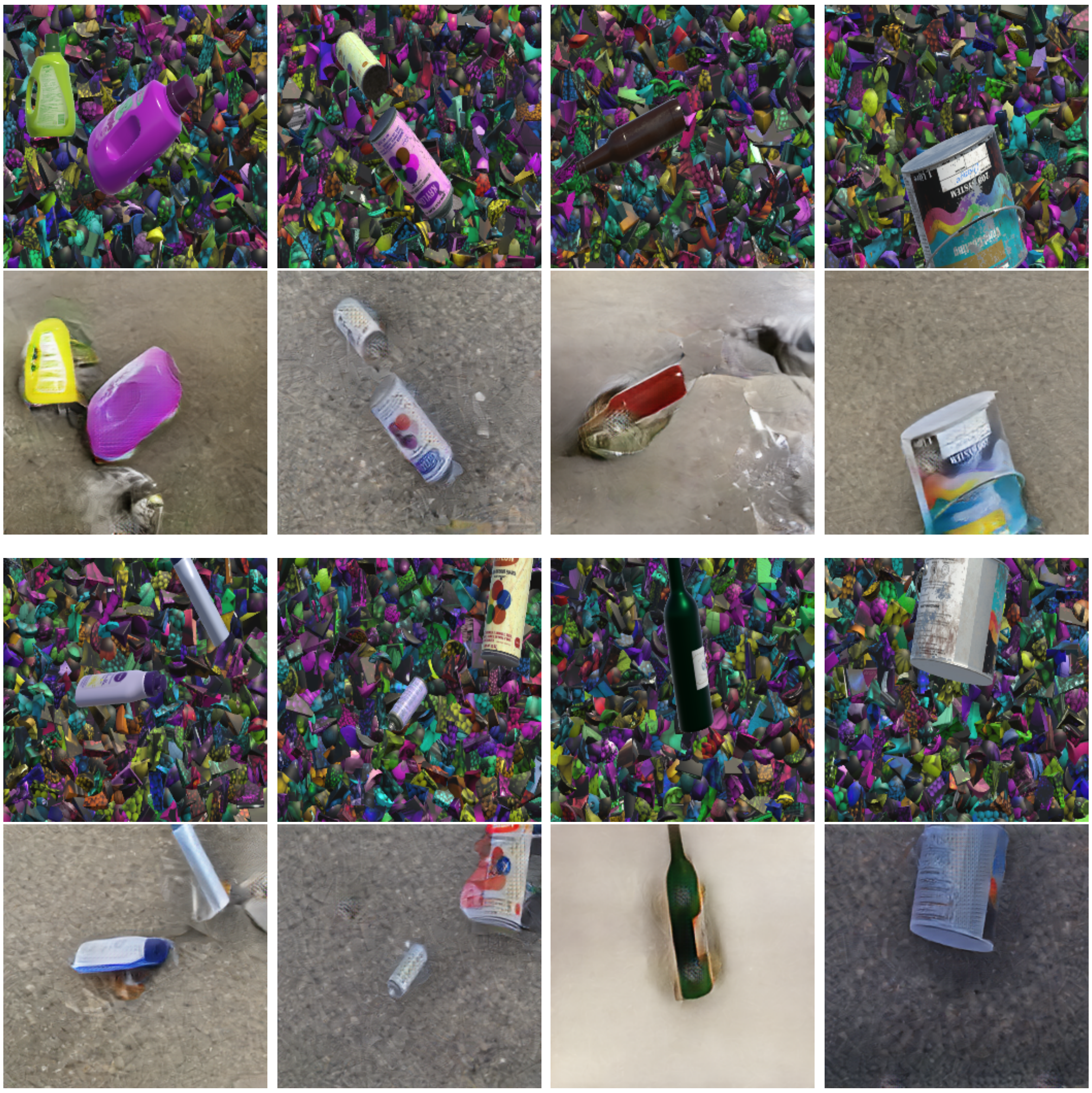


Figure G.3: Additional samples of seemingly successful adaptations using CycleGAN

References

[1] I. Sobel, “An isotropic 3x3 image gradient operator,” *Presentation at Stanford A.I. Project*, 1968.

[2] C. Harris and M. Stephens, “A combined corner and edge detector,” in *In Proc. of Fourth Alvey Vision Conference*, pp. 147–151, 1988.

[3] J. Matas, O. Chum, M. Urban, and T. Pajdla, “Robust Wide Baseline Stereo from Maximally Stable Extremal Regions,” *Image and Vision Computing*, vol. 22, pp. 761–767, 09 2004.

[4] E. Rosten and T. Drummond, “Machine Learning for High-Speed Corner Detection,” in *Computer Vision – ECCV 2006* (A. Leonardis, H. Bischof, and A. Pinz, eds.), (Berlin, Heidelberg), pp. 430–443, Springer Berlin Heidelberg, 2006.

[5] D. G. Lowe, “Distinctive Images Features from Scale-Invariant Keypoints,” in *International Journal of Computer Vision*, pp. 91–110, 2004.

[6] H. Bay, T. Tuytelaars, and L. Van Gool, “SURF: Speeded Up Robust Features,” in *Computer Vision – ECCV 2006* (A. Leonardis, H. Bischof, and A. Pinz, eds.), (Berlin, Heidelberg), pp. 404–417, Springer Berlin Heidelberg, 2006.

[7] W. S. McCulloch and W. Pitts, “A logical calculus of the ideas immanent in nervous activity,” *The bulletin of mathematical biophysics*, vol. 5, pp. 115–133, 1943.

[8] F. Rosenblatt, “The perceptron - a perceiving and recognizing automaton,” Tech. Rep. 85-460-1, Cornell Aeronautical Laboratory, Ithaca, New York, January 1957.

[9] C. Sun, A. Shrivastava, S. Singh, and A. Gupta, “Revisiting unreasonable effectiveness of data in deep learning era,” in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.

[10] K. P. Murphy, *Machine Learning: A Probabilistic Perspective.* The MIT Press, 2012.

[11] G. Louppe, “Fundamentals of machine learning.” `https://gloupe.github.io/info8010-deep-learning/pdf/lec1.pdf`, 2021.

[12] P. Geurts, “An introduction to machine learning.” `https://people.montefiore.uliege.be/lwh/AIA/intro-ml-2017-light.pdf`, 2020.

[13] X. Glorot, A. Bordes, and Y. Bengio, “Deep sparse rectifier neural networks,” in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics* (G. Gordon, D. Dunson, and M. Dudík, eds.), vol. 15 of *Proceedings of Machine Learning Research*, (Fort Lauderdale, FL, USA), pp. 315–323, PMLR, 11–13 Apr 2011.

[14] S. Elfwing, E. Uchibe, and K. Doya, “Sigmoid-weighted linear units for neural network function approximation in reinforcement learning,” *Neural networks : the official journal of the International Neural Network Society*, vol. 107, pp. 3–11, 2018.

[15] P. Ramachandran, B. Zoph, and Q. Le, “Swish: a self-gated activation function,” 2017.

[16] G. Louppe, “Multi-layer perceptron.” `https://glouppe.github.io/info8010-deep-learning/pdf/lec2.pdf`, 2021.

[17] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[18] S. Russell and P. Norvig, *Artificial intelligence : a modern approach*. Pearson, 4^{th} ed., 2020.

[19] P. Goyal, P. Dollár, R. B. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, “Accurate, large minibatch SGD: training ImageNet in 1 Hour,” *ArXiv*, vol. abs/1706.02677, 2017.

[20] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, “On the importance of initialization and momentum in deep learning,” in *Proceedings of the 30th International Conference on Machine Learning* (S. Dasgupta and D. McAllester, eds.), vol. 28 of *Proceedings of Machine Learning Research*, (Atlanta, Georgia, USA), pp. 1139–1147, PMLR, 17–19 Jun 2013.

[21] G. Hinton, “Overview of mini-batch gradient descent.” `https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`, 2014.

[22] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings* (Y. Bengio and Y. LeCun, eds.), 2015.

[23] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” in *Proceedings of the 32nd International Conference on Machine Learning*, vol. 37, p. 448–456, 2015.

[24] G. Louppe, “Convolutional networks.” `https://glouppe.github.io/info8010-deep-learning/pdf/lec5.pdf`, 2021.

[25] V. Dumoulin and F. Visin, “A guide to convolution arithmetic for deep learning,” *ArXiv*, vol. abs/1603.07285, 2016.

[26] H. Lin, Z. Shi, and Z. Zou, “Maritime Semantic Labeling of Optical Remote Sensing Images with Multi-Scale Fully Convolutional Network,” *Remote. Sens.*, vol. 9, p. 480, 2017.

[27] H. Rezatofighi, N. Tsoi, J. Gwak, A. Sadeghian, I. Reid, and S. Savarese, "Generalized Intersection Over Union: A Metric and a Loss for Bounding Box Regression," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[28] Z. Zheng, P. Wang, W. Liu, J. Li, R. Ye, and D. Ren, "Distance-IoU Loss: Faster and Better Learning for Bounding Box Regression," in *The AAAI Conference on Artificial Intelligence (AAAI)*, 2020.

[29] Z. Zheng, P. Wang, D. Ren, W. Liu, R. Ye, Q. Hu, and W. Zuo, "Enhancing Geometric Factors in Model Learning and Inference for Object Detection and Instance Segmentation," *IEEE Transactions on Cybernetics*, vol. 52, no. 8, pp. 8574–8586, 2022.

[30] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 580–587, 2014.

[31] R. Girshick, "Fast R-CNN," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, December 2015.

[32] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," in *Advances in Neural Information Processing Systems* (C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, eds.), vol. 28, Curran Associates, Inc., 2015.

[33] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "SSD: Single Shot MultiBox Detector," in *Computer Vision – ECCV 2016* (B. Leibe, J. Matas, N. Sebe, and M. Welling, eds.), (Cham), pp. 21–37, Springer International Publishing, 2016.

[34] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 779–788, 2016.

[35] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The Pascal Visual Object Classes (VOC) Challenge," *International Journal of Computer Vision*, vol. 88, pp. 303–338, June 2010.

[36] J. Redmon and A. Farhadi, "YOLO9000: Better, Faster, Stronger," *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 6517–6525, 2017.

[37] J. Redmon and A. Farhadi, "YOLOv3: An Incremental Improvement," *ArXiv*, vol. abs/1804.02767, 2018.

[38] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *Proceedings of the IEEE / CVF Computer Vision and Pattern Recognition Conference (CVPR)*, pp. 770–778, 2016.

[39] G. Jocher, A. Chaurasia, A. Stoken, J. Borovec, NanoCode012, Y. Kwon, TaoXie, J. Fang, imyhxy, K. Michael, Lorna, A. V, D. Montes, J. Nadar, Laughing, tkianai, yxNONG, P. Skalski, Z. Wang, A. Hogan, C. Fati, L. Mammana,

AlexWang1900, D. Patel, D. Yiwei, F. You, J. Hajek, L. Diaconu, and M. T. Minh, "ultralytics/yolov5: v6.1 - TensorRT, TensorFlow Edge TPU and OpenVINO Export and Inference." `https://doi.org/10.5281/zenodo.6222936`, February 2022.

[40] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "YOLOv4: Optimal Speed and Accuracy of Object Detection," *ArXiv*, vol. abs/2004.10934, 2020.

[41] C.-Y. Wang, H.-Y. M. Liao, Y.-H. Wu, P.-Y. Chen, J.-W. Hsieh, and I.-H. Yeh, "CSPNet: A New Backbone That Can Enhance Learning Capability of CNN," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2020.

[42] S. Liu, L. Qi, H. Qin, J. Shi, and J. Jia, "Path aggregation network for instance segmentation," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 8759–8768, 2018.

[43] K. He, X. Zhang, S. Ren, and J. Sun, "Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition," in *Computer Vision – ECCV 2014* (D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, eds.), (Cham), pp. 346–361, Springer International Publishing, 2014.

[44] Z. Huang, J. Wang, X. Fu, T. Yu, Y. Guo, and R. Wang, "DC-SPP-YOLO: Dense connection and spatial pyramid pooling based YOLO for object detection," *Information Sciences*, vol. 522, pp. 241–258, 2020.

[45] S. R. Richter, V. Vineet, S. Roth, and V. Koltun, "Playing for data: Ground truth from computer games," in *Computer Vision – ECCV 2016* (B. Leibe, J. Matas, N. Sebe, and M. Welling, eds.), (Cham), pp. 102–118, Springer International Publishing, 2016.

[46] S. R. Richter, Z. Hayder, and V. Koltun, "Playing for benchmarks," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.

[47] M. Wrenninge and J. Unger, "Synscapes: A photorealistic synthetic dataset for street scene parsing," *CoRR*, vol. abs/1810.08705, 2018.

[48] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An open urban driving simulator," in *Proceedings of the 1st Annual Conference on Robot Learning* (S. Levine, V. Vanhoucke, and K. Goldberg, eds.), vol. 78 of *Proceedings of Machine Learning Research*, pp. 1–16, PMLR, 13–15 Nov 2017.

[49] G. Georgakis, A. Mousavian, A. Berg, and J. Kosecka, "Synthesizing training data for object detection in indoor scenes," in *Proceedings of Robotics: Science and Systems*, (Cambridge, Massachusetts), July 2017.

[50] D. Dwibedi, I. Misra, and M. Hebert, "Cut, paste and learn: Surprisingly easy synthesis for instance detection," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.

[51] M. Z. Wong, K. Kunii, M. Baylis, W. H. Ong, P. Kroupa, and S. Koller, "Synthetic dataset generation for object-to-model deep learning in industrial applications," *PeerJ Computer Science*, vol. 5, 2019.

[52] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, "Domain randomization for transferring deep neural networks from simulation to the real world," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 23–30, 2017.

[53] J. Tremblay, A. Prakash, D. Acuna, M. Brophy, V. Jampani, C. Anil, T. To, E. Cameracci, S. Boochoon, and S. Birchfield, "Training deep networks with synthetic data: Bridging the reality gap by domain randomization," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2018.

[54] S. Hinterstoisser, O. Pauly, H. Heibel, M. Martina, and M. Bokeloh, "An annotation saved is an annotation earned: Using fully synthetic training for object detection," in *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*, pp. 2787–2796, 2019.

[55] A. Prakash, S. Boochoon, M. Brophy, D. Acuna, E. Cameracci, G. State, O. Shapira, and S. Birchfield, "Structured domain randomization: Bridging the reality gap by context-aware synthetic data," *2019 International Conference on Robotics and Automation (ICRA)*, pp. 7249–7255, 2019.

[56] P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros, "Image-to-image translation with conditional adversarial networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[57] K. Bousmalis, N. Silberman, D. Dohan, D. Erhan, and D. Krishnan, "Unsupervised pixel-level domain adaptation with generative adversarial networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[58] A. Shrivastava, T. Pfister, O. Tuzel, J. Susskind, W. Wang, and R. Webb, "Learning from simulated and unsupervised images through adversarial training," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[59] J. Hoffman, E. Tzeng, T. Park, J.-Y. Zhu, P. Isola, K. Saenko, A. Efros, and T. Darrell, "CyCADA: Cycle-consistent adversarial domain adaptation," in *Proceedings of the 35th International Conference on Machine Learning* (J. Dy and A. Krause, eds.), vol. 80 of *Proceedings of Machine Learning Research*, pp. 1989–1998, PMLR, 10–15 Jul 2018.

[60] J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros, "Unpaired image-to-image translation using cycle-consistent adversarial networks," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.

[61] T. Park, A. A. Efros, R. Zhang, and J.-Y. Zhu, "Contrastive learning for unpaired image-to-image translation," in *Computer Vision – ECCV 2020* (A. Vedaldi, H. Bischof, T. Brox, and J.-M. Frahm, eds.), (Cham), pp. 319–345, Springer International Publishing, 2020.

[62] S. Hinterstoisser, V. Lepetit, P. Wohlhart, and K. Konolige, "On pre-trained image features and synthetic images for deep learning," in *Computer Vision –*

ECCV 2018 Workshops (L. Leal-Taixé and S. Roth, eds.), (Cham), pp. 682–697, Springer International Publishing, 2019.

[63] F. E. Nowruzi, P. Kapoor, D. Kolhatkar, F. A. Hassanat, R. Laganière, and J. Rebut, "How much real data do we actually need: Analyzing object detection performance using synthetic and real data," in *ICML Workshop on AI for Autonomous Driving*, June 2019.

[64] K. Bousmalis, G. Trigeorgis, N. Silberman, D. Krishnan, and D. Erhan, "Domain separation networks," in *Advances in Neural Information Processing Systems* (D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, eds.), vol. 29, Curran Associates, Inc., 2016.

[65] Y. Ganin, E. Ustinova, H. Ajakan, P. Germain, H. Larochelle, F. Laviolette, M. March, and V. Lempitsky, "Domain-adversarial training of neural networks," *Journal of Machine Learning Research*, vol. 17, no. 59, pp. 1–35, 2016.

[66] E. Tzeng, J. Hoffman, T. Darrell, and K. Saenko, "Simultaneous deep transfer across domains and tasks," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, December 2015.

[67] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: Common objects in context," in *Computer Vision – ECCV 2014* (D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, eds.), (Cham), pp. 740–755, Springer International Publishing, 2014.

[68] Unity Technologies, "Unity Perception Package." `https://github.com/Unity-Technologies/com.unity.perception`, 2020.

[69] Y.-C. Jhang, A. Palmar, B. Li, S. Dhakad, S. K. Vishwakarma, J. Hogins, A. Crespi, C. Kerr, S. Chockalingam, C. Romero, A. Thaman, and S. Ganguly, "Training a performant object detection ML model on synthetic data using Unity Perception tools." `https://blogs.unity3d.com/2020/09/17/training-a-performant-object-detection-ml-model-on-synthetic-data-using-unity-computer-vision-tools/`, Sep 2020.

[70] J. Frank, T. Eisenhofer, L. Schönherr, A. Fischer, D. Kolossa, and T. Holz, "Leveraging frequency analysis for deep fake image recognition," in *Proceedings of the 37th International Conference on Machine Learning*, ICML'20, JMLR.org, 2020.

[71] Y. Ge, J. Xu, B. N. Zhao, N. Joshi, L. Itti, and V. Vineet, "DALL-E for detection: Language-driven compositional image synthesis for object detection," 2022.