Alma Mater Studiorum · Università di Bologna

SCHOOL OF SCIENCE

First Cycle Degree in Computer Science

Production Lot Detection through Synthetic Generated Data

Supervisor:
Chiar.mo Prof.
Andrea Asperti

Candidate:
Filippo Peterlini

Co-supervisor:
Alessandro Ferraiuolo

Session I
Academic Year 2021/2022

"Replicants are like any other machine. They're either a benefit or a hazard. If they're a benefit, it's not my problem."

Abstract

Correctness of information gathered in production environments is an essential part of quality assurance processes in many industries, this task is often performed by human resources who visually take annotations in various steps of the production flow. Depending on the performed task the correlation between where exactly the information is gathered and what it represents is more than often lost in the process. The lack of labeled data places a great boundary on the application of deep neural networks aimed at object detection tasks, moreover supervised training of deep models requires a great amount of data to be available. Reaching an adequate large collection of labeled images through classic techniques of data annotations is an exhausting and costly task to perform, not always suitable for every scenario. A possible solution is to generate synthetic data that replicates the real one and use it to fine-tune a deep neural network trained on one or more source domains to a different target domain. The purpose of this thesis is to show a real case scenario where the provided data were both in great scarcity and missing the required annotations. Sequentially a possible approach is presented where synthetic data has been generated to address those issues while standing as a training base of deep neural networks for object detection, capable of working on images taken in production-like environments. Lastly, it compares performance on different types of synthetic data and convolutional neural networks used as backbones for the model.

Sommario

Ottenere informazioni corrette in ambienti di produzione è parte fondamentale del processo di controllo qualità in molte realtà industriali, spesso questo compito è portato a termine da persone addette che visivamente prendono annotazioni durante le varie fasi del ciclo produttivo. A discapito di ció e dipendentemente dal tipo di controllo, la correlazione tra dove i dati vengono raccolti e cosa questi rappresentino viene nella maggior parte dei casi perduta durante questa fase. La mancanza di dati annotati rende di difficile applicazione l'utilizzo di modelli basati su reti neurali per object detection, oltre a questo il training supervisionato su modelli più profondi necessita di una grande quantità di dati. Costruire una comprensiva collezione di immagini annotate tramite tecniche manuali è un'operazione costosa ed estenuante, non sempre applicabile in ogni circostanza. Una possibile soluzione è generare dati che replicano quelli reali ed utilizzare questi per fare fine-tuning di reti neurali precedentemente allenate su diversi domini per adattarle al nostro. In questa tesi mostreremo un caso reale dove i dati a disposizione non raggiungevano un numero adeguato di esempi e non riportavano le necessarie annotazioni. Successivamente analizzeremo l'approccio utilizzato per generare dati sintetici per sostituire quelli reali. Questi serviranno come base su cui fare fine-tuning di un modello precedentemente allenato, così da ottenere un sitsema di object-detection in grado di lavorare su immagini prese da stabilimenti produttivi. Infine saranno analizzate le performance al cambiare dal tipo di dati generati e di modelli utilizzati.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Automation of previously human performed tasks has always been of great interest to industries and organizations for obvious efficiency/cost reasons, in recent years this phenomenon gathered even more traction also due to the application of modern machine learning models, in particular Deep Learning ones. Computer vision is one of the fields that more has benefited from it, thanks to its great range of possible applications in automation tasks, such as:

- **Image Classification**: in image classification tasks the goal is to correctly predict whether the presented scene contains one or more previously defined classes of objects.
- **Object Detection**: in Object Detection the aim is to know not only if the image contains the target object but also its position, marking and labeling the region where an instance of an object is contained.
- **Semantic Segmentation**: Semantic Segmentation is the task of assigning a class label (car, person, buildings, ...) to each point/pixel in the scene.
- **Instance Segmentation**: instance Segmentation means detecting and masking each distinct object of interest in a scene.
- **Panoptic Segmentation**: lastly Panoptic Segmentation is a combination of Instance Segmentation and Semantic Segmentation in which the result must be a

mask of all the visible area that any different object occupies, effectively classifying each pixel/point in the image for each visible instance.

For instance, a real case scenario can be the application of these techniques to improve the quality control section of a floor tile manufacturing company. To achieve that, pictures of the final product are taken and analyzed through a computer vision model. Image classification could sort each tile by its overall quality (1st choice, 2nd choice), object detection can find all the spots where chippings may have occurred discarding the ones with too many, and lastly image segmentation can be used to tell apart which points have shading's variation problems and give insight if there have been coloring issues during production.

However, the practical implementation of those techniques is slowed down by two main factors of deep learning. Firstly, training datasets and actual problem need to be very similar regarding their feature space and the data distribution in it. Furthermore, only those cases that are represented within the training data can be learned by the model. These results in training datasets need to be large and diverse enough to include rare events as well. In practice, such data collections are increasingly difficult to obtain as problems become more complex or if the examples are not easily obtainable. The second issue is that training a deep learning algorithm requires vast amounts of computational power and large data collections [17]. In the highly dynamic industrial automation environment, where e.g. one production line might switch products, tools, or processes regularly, this is impractical to sustain. A partial solution to both issues can be achieved through transfer learning [15], which consists of a set of approaches aimed to reduce the amount and quality of data needed, providing a way to build upon previously acquired knowledge as opposed to starting every learning process from zero.

Another viable solution to mitigate the absence of training data comes from the applications of data augmentation. Minor changes such as flips, translations or rotations of the original dataset's images have the double benefit of increasing the overall amount of data and creating a classifier with a strong invariance property that reduces the possibility of overfitting during training [3].

Chapter 2

Approaching the problem

2.1 Defining the Problem

The problem that will be discussed in this thesis originates from the need of automating annotation tasks that employees at Marcegaglia group perform to correctly keep track of the production stock. Marcegaglia is the Italian leading player and the largest independent global operator in the steel processing industry, one of its main goals is to continuously expand the corporate know-how to fully transform its worldwide production plants into 'smart factories'. In our specific scenario, the group production task force needs to extract the Production Lot Identifier (that from now on will also be abbreviated with 'PLI') seated on the newly produced steel coils and register it in a database. The production lot consists of a white handwritten marking of two stacked numbers, separated by a continuous line that traverses edge to edge the side of the coil. In fig. 2.1 an example is shown where the production lot identifier $\frac{4811}{94}$ sits in the top right position of the coil.

Figure 2.1: A side view of a steel sheet coil

Keeping that picture as a reference it is useful to make some assumptions on possible issues that during the extraction of the PLI numbers can be faced:

1. The setting where those images come from is not a controlled environment, it cannot be assumed that every image that will be analyzed is in perfect condition, chromatic aberration and overexposed points are visible on the left side of the coil while some parts of the camera lens are covered in droplets of water, blurring significantly some serial numbers in the down right side of the image.

2. Except for these issues fig. 2.1 is a perfect candidate for camera positioning, front facing the coil which takes almost the whole height of the image, so the PLI is not subject to geometric distortions. That is not the same situation in fig. 2.2 where there is a gray-scale picture of a coil that has been shot from above, moreover some parts of it are covered behind a wall, luckily the production lot id is visible but in a worst-case the model should notify that a partial identifier or no identifier has been detected.

Figure 2.2: A gray-scale top view of the coil

3. In both above examples, the PLI is isolated from the remaining handwritten markings that appear on the side of the coil, in fig. 2.3 it is possible to see that is not always the case. In the top left section, the production lot identifier overlays with another serial number wherewith shares the same color. Those situations will for sure create problems when the PLI number will have to be recognized.

Figure 2.3: An example where the PLI overlay other markings

After the above findings the problem has been separated into two tasks to better address these issues individually:

1. detects the region of the image where PLI is positioned.
2. proceeds with digits recognition to extract the PLI number.

In this thesis, we'll discuss only the first part of the solution.

2.2 Data Analysis

Before proceeding with the analysis of all the possible approaches to solve the problem it is needed to make some consideration on the amount of data available, that's important to initially filter out non-applicable methods incompatible with a small dataset. All the images given are listed here separated by category.

```
images
├── examples
│   ├── 4 gray-scale images with a resolution of 660x450 pixels
│   └── 3 colored images with a resolution of 660x450 pixels
├── fisheye
│   └── 16 colored images with a resolution of 2592x1944 pixels
└── topview
    └── 14 colored images with a resolution of 2560x1440 pixels
```

In 'examples' are collected low-res images given by the company to get a first insight on the problem, 'fisheye' are pictures taken by a camera equipped with a fisheye lens front facing the coil and 'topview' are images shot from above the target object.

2.3 Possible Approaches

State of the art object-detection models such as YOLO [18] have been trained on PASCAL VOC 2007 [6] and 2012 [7], datasets that respectively count 5.011 and 11.540 images for both training and validation on 20 different classes. Even if each picture

in the dataset would be labeled it would still not reach an adequate size to train an accurate object detection system. Further examinations of the dataset show that only 65% images effectively present a marked coil with a lot identifier, reducing even more the available examples to use. After these considerations approaching the problem as a standard object-detection task through a convolutional neural network using real data would not be able to produce satisfying results. Consequently, there are two possible approaches, trying to retrieve the region of interest viewing the problem as a non-deep learning computer vision task, or building, through synthetic data, a collection of images to train a model on that new dataset.

2.4 Standard Computer Vision Approaches

Initially, classic methods of computer vision have been applied to test which kind of result could be obtainable from a non-deep learning approach. That would have lifted us from the burdensome task of training a deep neural model and would have solved the dataset size issue while providing a system with a lower predictive time against deep learning techniques.

To simplify the problem it has been divided into two steps. First detect the coil, so a round/ellipsoid type of object, then detect the line that divides the numbers in the production lot identifier. Jumping straight to line detection would leave too many similar shape objects in the scene providing false-positive results. For these initial testing, a subset of the images provided in the 'example' category has been used, concentrating only on the colored ones and leaving the gray-scale images for a later moment. Proving the following testings has been possible thanks to the famous open-source computer vision library **OpenCV**.

2.4.1 Detecting the Coil

Filtering out the region where the reel was located has been achieved by straightforward use of *Hough Circles Transform* [2] [11] function provided from opencv, with some prior image color space manipulation and parameters tweaking. After the coil's position

was detected and cropped further masking around his perimeter was performed to eliminate even more unnecessary features. A visual step representation of the procedure can be seen in fig. 2.4.

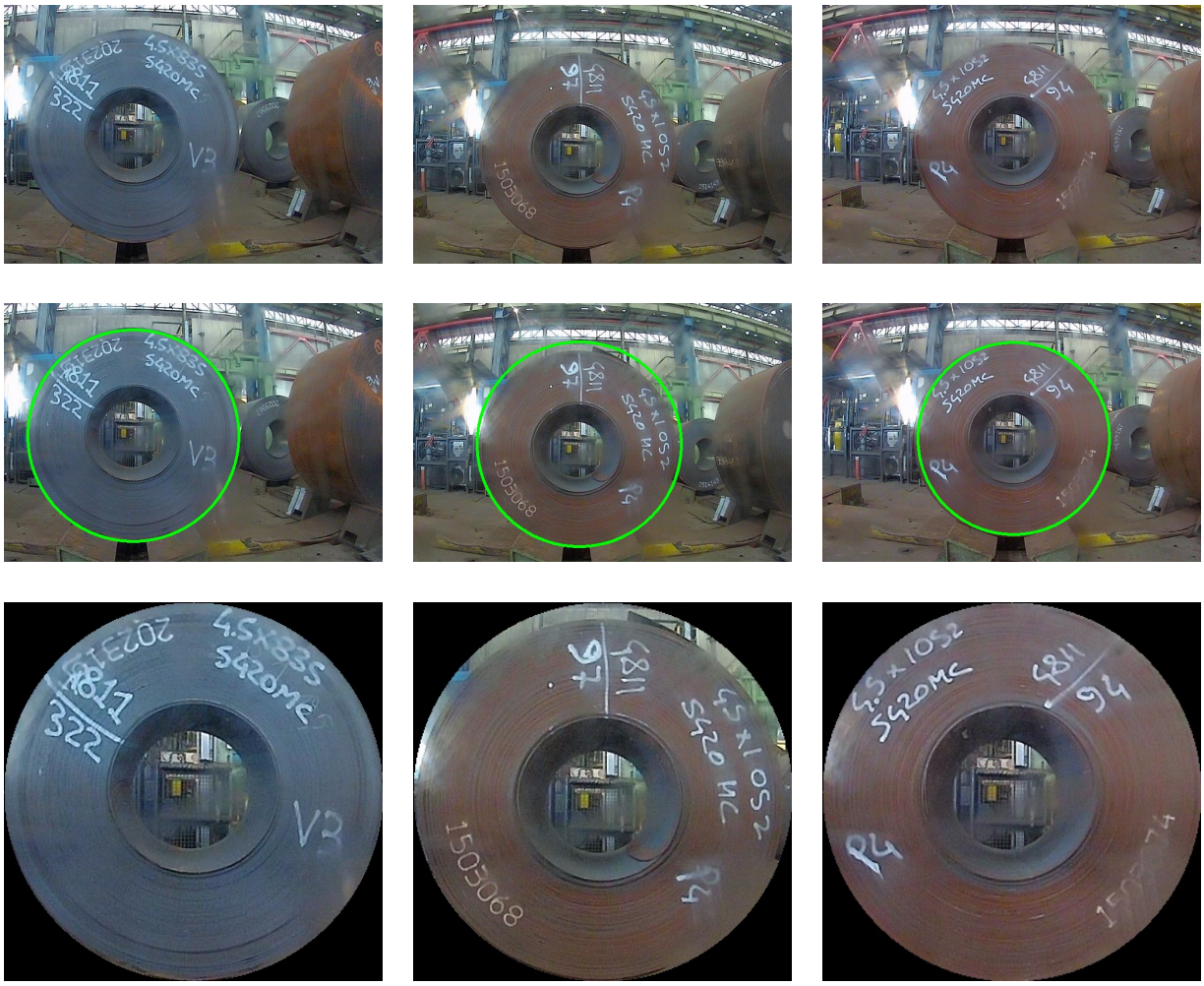


Figure 2.4: Visual representation of reel detection

2.4.2 Detecting the Line

Obtained a simpler image to work with the next step was to locate the production lot on the reel. Obviously trying to detect it by working on the digits would not be effective since, like it has been seen in the previous images, PLI shares its space on the side of the reel with other annotations and serial numbers, the only unique feature to extract shared by the production lot annotations was the line between the two numbers.
OpenCV provides another function that can be tweaked to detect lines in an image, *Probabilistic Hough Line Transform* [5] [11]. Before applying it some corrections on input images must be performed to better isolate the line from the numbers, cases like fig. 2.3 can impact the effectiveness of OpenCV line detection, providing false positives. To achieve that the picture has been upscaled and one pass of erosion has been applied to it, which removes enough perimeter pixels to separate the number from the line while leaving its shape intact, after that the original resolution is restored. Finally, *Probabilistic Hough Line Transform* can be applied with a tweaked set of parameters to retrieve all the detected lines in the image. If more than one line is detected only the closest to the image center is kept, like in fig. 2.5 central reference, where three possible matches have been detected. Lastly, the selected line's angle can be calculated with the following formula:

$$\measuredangle line = atan(\frac{Y_2 - Y_1}{X_2 - X_1})$$

so it can be applied as a rotation to present the image in the right horizontal configuration. Depending on the original PLI's position its orientation could be flipped, hence both horizontally mirrored variants must be provided as shown on the bottom row of fig. 2.5.

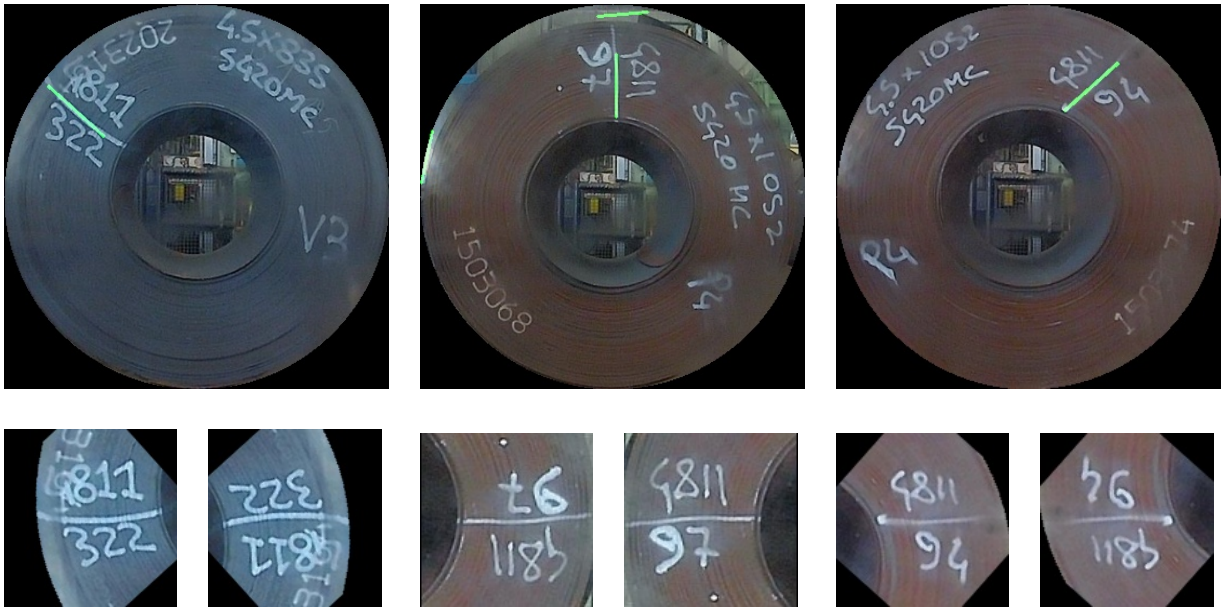


Figure 2.5: Visual representation of lines detection

2.4.3 Testing the Performance

For the last step, the algorithm needs to be tested on the remaining images taken from 'fisheye' and 'topview' categories. This method did not provide good results, the difference in image features between each category placed a higher bound on the algorithm's applicability. This is due also because metal materials cause light explosion in image processing and this affects finding contours badly. Changing the parameters of *Hough Circles Transform* and *Probabilistic Hough Line Transform* to accommodate better the different images provided more reasonable results, but was not compatible with our constraints of obtaining a general and robust enough solution to our problem. These motivations pushed us to the second approach, so the focus was moved to generating a robust dataset able to train a deep learning object detection model.

Chapter 3

Generating the Dataset

Inspired by recent results obtained by deep learning models trained over synthetic data [14] some possible ways to generate a new dataset have been studied to replace it with the existing one. Moreover an approach of this type is:

- Exempt of human errors
- Inexhaustible
- Pre-annotated
- Free from many ethical and practical concerns
- Cheaper than traditional methods

This chapter aims to show three different approaches, two of them completely automated and one hybrid.

3.1 Automated Generation

Fully automated generation does not require human intervention to create new items in the dataset, instead, it leverages only on the random generation implementation of its algorithms. The trade-off to this is that, depending on the features that must be

replicated from the original data, the underlying algorithm could become harder to implement or more computationally costly, hence it becomes important to choose the degree of accuracy that is desired. In the next sections two different approaches will be shown to reconstruct a synthetic representation of the PLI, the first easier to implement and the second produces a closer approximation to the real case. A base algorithm is shown here, each approach implements differently the *GeneratePLI* function.

Algorithm 1 Generic Scene Generation

for $i \in Range(0, target\_size)$ **do**
  $background \leftarrow Random.Choice(backgrounds)$
  $pli\_image \leftarrow GeneratePLI(source)$
  $scene, bounding\_box \leftarrow Overlay(pli\_image, background)$
  $return\ scene, bounding\_box$

3.1.1 Font-Source Generation

In this approach, the source parameter for *GeneratePLI* is a predefined font. To better simulate the real data it has been chosen one that mimics digits handwritten with a marker, fig. 3.1.

Figure 3.1: The complete font source

Sequentially the function chose one random upper number and one random lower number and convert those to images with different values for the alpha channel. A variable thickness line is drawn and then the final PLI image is combined, picking a random offset for the numbers-line distance. Before returning the resulting image *GeneratePLI* randomly scale - rotate - skew it and apply an erosion filter. Two examples of generated scenes are shown in fig. 3.2.

Figure 3.2: Image examples taken from font-source generation implementation

This approach has a main issue, and it comes directly from the source used by the *GeneratePLI* function. This implementation limits the maximum instances of different digits to ten as those are available in the font family fig. 3.1, with the consequence that in each image the same digits representation will be repeated. That can affect the model during training, introducing a side-effect where the learned task is not to detect the PLI position but find the specific font in the image. Using multiple fonts would be a partial solution but it can become harder and time consuming to search for ones with the handwritten-like characteristics that are needed.

3.1.2 Handwritten-Source Generation

Keeping as objective the implementation of a fully automated generation function of the PLI image, it's also become necessary to find an almost unbounded source of possible digits representation to use as seed. This section will show a possible substitution for the font source presented in section 3.1.1 where each instance is taken from the MNIST [4] dataset.

The MNIST dataset is a collection of handwritten digits largely used for classification tasks, it has a training set of 60.000 examples and a test set of 10.000 examples. All the single channel digits images have been centered and size-normalized to 28x28 pixels. In our implementation the digits images have been used to assemble the synthetic PLI representation, to achieve that each digit must be previously concatenated into numbers.

MNIST is made easily available by *keras* in the *tensorflow* framework, loading the train and test datasets consist of the following code:
`(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()`.
Obtained the images vector X and the label vector Y each digit has been stripped of the lateral black padding to close the gap when merged into a whole number. Finally, after generating two numbers those can be vertically assembled, separated by a horizontal line, into a PLI. In fig. 3.3 various examples have been generated by randomly picking digits from MNIST x_train dataset.

Figure 3.3: PLI generated taking as source the MNIST dataset

Expect for those changes the remaining scene generation task is the same as the font-source case, with the great advantage that now there are 60.000 possible choices for a different digit image.

Figure 3.4: Image examples taken from handwritten-source generation implementation

This approach further closes the gap between real and generated data, producing even more different examples for the target dataset. However, two key constraints of the production lot identifier are not replicated by this implementation:

1. The PLI cannot exists outside of the reel's side[1]

2. PLI's numbers divider goes from one edge to another of the coil

This can respectively be seen in the two examples of fig. 3.4. To achieve this kind of fidelity the coil side region should be detected for each background image. As shown in section 2.4.1 coil detection via 'standard' approaches of computer vision did not provide satisfying results. Another viable option would be to generate one more synthetic dataset to perform object detection only for the side of the coil, increasing the overall complexity of the problem. So to keep the solution simple and more flexible it has been decided to try a hybrid approach for data generation.

3.2 Hybrid Generation

This set of approaches consists in splitting the generation task into two steps. First, using support tools to fasten up the process, manually create fake objects over a pre-existing set of backgrounds, separating each object on a different layer, then for each layer automatically annotate the position of the object and overlay it on the background to assemble the final scene.

3.2.1 Drawing-Source Generation

To draw the fake PLIs the backgrounds have been selected from the existing dataset, precisely the ones not containing already a real PLI in the center coil. Those images have been loaded in a drawing tablet application called Procreate and for each background, ten transparent layers have been added to draw a fake PLI, following the guidelines stated at the end of section 3.1.2, on each one of them using different brushes and values thickness/opacity. Finally, everything has been exported to be processed. Procreate append to the name of the background image file a '-i' for each i-th exported layer, so the directory structure would be:

[1]*if more coils are visible the central one is the region of interest

```
export
├── scene_0001-1 (background)
├── scene_0001-2 (1st layer)
├── scene_0001-3 (2nd layer)
├── ⋮
└── scene_0001-11 (10th layer)
```

In OpenCV finding the bounding box for an object drawn on a transparent image is achieved in a single line of code using *cv2.boundingRect* on the alpha channel of the layer.

```
cv2.boundingRect(layer[..., 3])
```

For the last step, the final image is built overlaying each layer on its background, only after applying a randomized hue variation to it, avoiding the same repeated background in ten different images. A partial visual representation of the process can be seen in fig. 3.5.

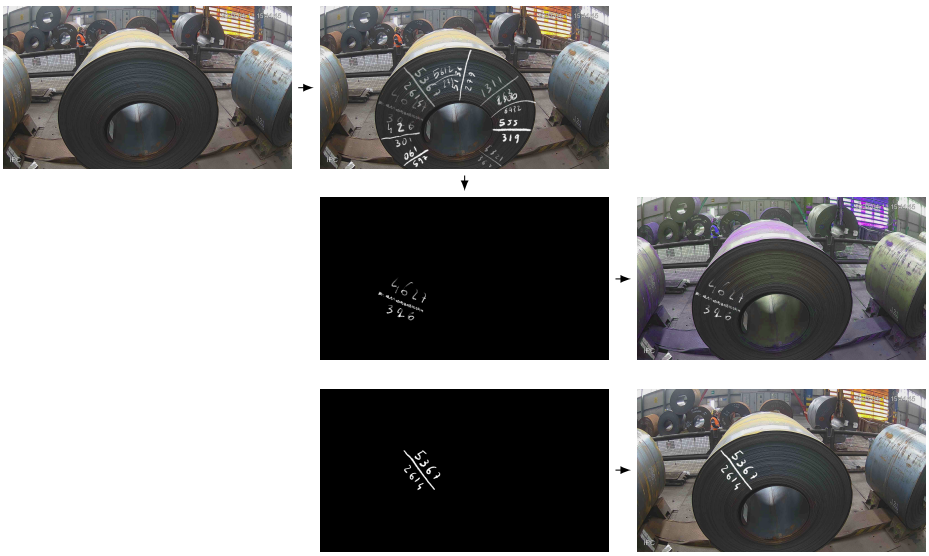


Figure 3.5: Hybrid generation visualized

Chapter 4

Obtaining the Model

The application of complex deep learning models is getting easier by the day. Fundamental has been the great abstraction level delivered by modern machine learning frameworks like Pytorch [16] and Tensorflow [1]. On top of that multiple projects have been born to automate the training, testing, performance measurement and delivery of machine learning models in production. Providing also well know pre-trained neural networks for transfer learning applications, tools for data augmentation and dataset management. This chapter will introduce Detectron2, a Pytorch based platform for object-detection, and how it has been tweaked to provide an accurate model for PLI localization.

4.1 Detectron2

First published in 2018 Detectron [9] is a Facebook AI Research project initially designed to support rapid implementation and evaluation of novel research for object-detection, successively extended to production applications with the release of Detectron2 [21], moving its underlying framework from Caffe2 to PyTorch. It includes many capabilities such as panoptic segmentation, Densepose, Cascade R-CNN, rotated bounding boxes, PointRend becoming a flexible library for different solutions and implementations for many object-detection algorithms like Fast R-CNN [8], Faster R-CNN [19], Mask R-CNN [10], RetinaNet [12] and many more. In the next sections, it will be shown

how Detectron has been used to fine-tune a trained model to adapt it for PLI detection. Before that, a brief introduction of how to solve object-detection in a deep learning paradigm is examined.

4.2 Object-Detection

4.2.1 Object-Localization

At the start of chapter 1 a definition for object-detection has been given, the goal is to correctly classify and get all the regions of interest (ROI) for multiple object instances in a picture. A simpler version is object-localization in which a single ROI classification is required as result, so taking as input an image and for output a class and its bounding box. The prediction can be represented by a one-dimensional vector in this configuration.

$$y = \begin{bmatrix} P_c \\ bbox_x \\ bbox_y \\ bbox_h \\ bbox_w \\ C_1 \\ C_2 \\ \vdots \\ C_n \end{bmatrix} \tag{4.1}$$

P_c is alternatively 0 or 1 if the image contains an object of the given classes, while $bbox\_x, bbox\_y, bbox\_h, bbox\_w$ are the coordinate for the region of interest, $C_1, \ldots, C_n$ are each of the possible n class labels. The base deep learning model to solve object-localization would take the input image and pass it to a convolutional network, the resulting vector features then goes through a softmax layer that outputs the predicted class and bounding box as vector 4.1.

4.2.2 Convolutional sliding windows

A computationally inefficient way to achieve object-detection via sliding-windows would be to compute the entire pipeline for each window of the input one at a time. Basically training a classification model with closeup crops of the object that needs to be detected, then takes multiples crops of the input image and it performs object classification on those, if no object is detected different values for stride and crop size are used. To keep down the computational cost it is possible to exploit ConvNets inherent efficiency when applied in a sliding fashion, thanks to their intrinsic features of sharing computations on overlapping regions [20]. In fig. 4.1 this process is shown for a ConvNets trained on an input image of size 14x14[1], if at test time a 16x16 input image is presented it is possible to run the same ConvNets, keeping the convolutional 5x5 filters and 2x2 pooling filters, to obtain a 2x2 output layer that represents a mapping of values equal to running the model four times for each portion of the input image using the inefficient implementation of sliding window.

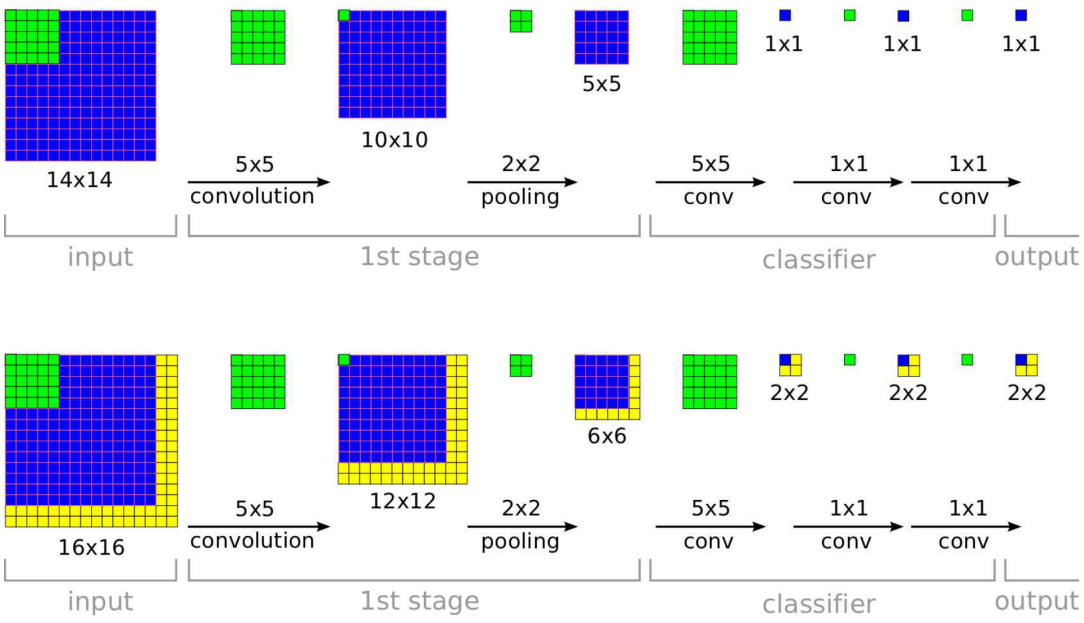


Figure 4.1: The efficiency of ConvNets for detection

[1]feature dimension omitted for simplicity.

4.2.3 Refine bounding box prediction, YOLO algorithm

In this section, the YOLO algorithm [18] is briefly presented to obtain more refined bounding boxes around the target objects. The way it is achieved goes through a splitting of the input image in a $S \times S$ grid (in the original paper a 7x7 grid was used for evaluation), then running the classification and localization algorithm shown in section 4.2.1 on each grid cell. In training each grid cell needs a label like vector 4.1 in the dataset, that produce a prediction output tensor of size $S \times S \times (5 \times B + C)$, where B is the number of possible bounding box per grid cell. This implementation allows one and only one class associated with a grid cell, as shown in fig. 4.2 Classprobability map.

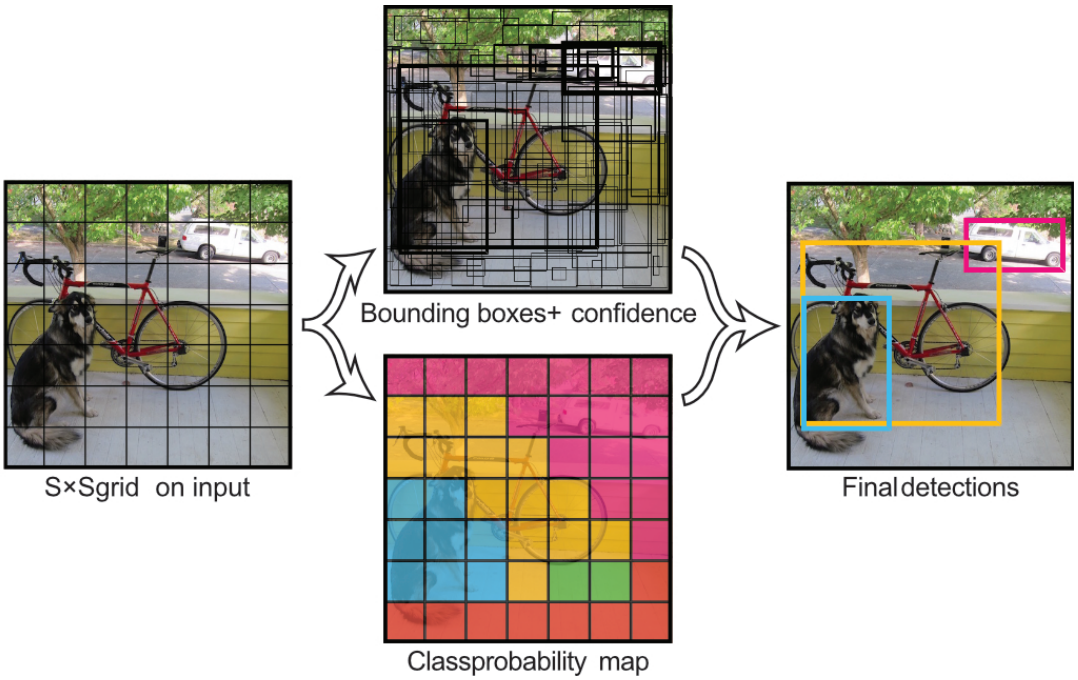


Figure 4.2: Yolo grid division for classification and bounding box prediction

ConvNets implementation of this algorithm does that by forcing the last layer to be a volume with the upper defined size, avoiding so to run classification and localization for each cell in the grid. In fig. 4.3 the last layer volume is $7 \times 7 \times 30$ because YOLO has been trained over 20 classes with 2 possible bounding boxes per grid cell.

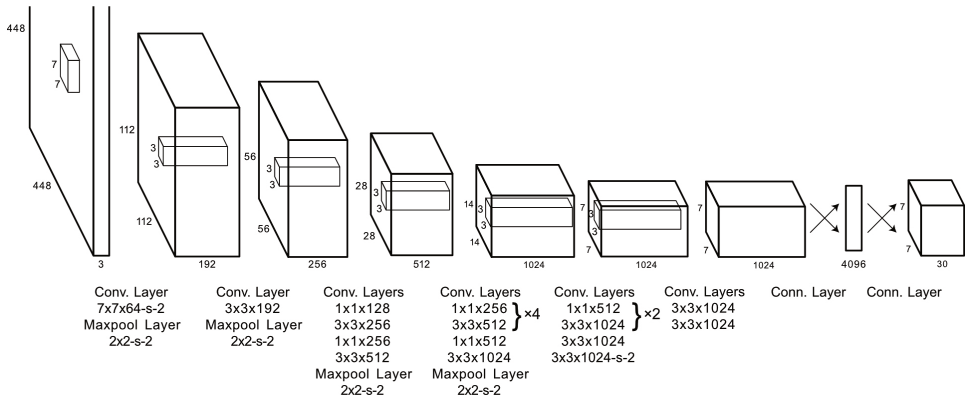


Figure 4.3: Layers schema of YOLO's ConvNets

4.2.4 Evaluating object-detection, IoU

In order to compare the object-detection model predictions to ground-truth bounding boxes, the *Intersection over Union, IoU* function is required. Given two region B_1, B_2 the Iou can be calculated as follow:

$$IoU = \frac{B_1 \cap B_2}{B_1 \cup B_2}$$

By convention, an IoU greater than 0.5 is considered a valid bounding box prediction. This evaluation can be used to implement a non-max suppression algorithm to eliminate possible multiple bounding box predictions over the same object, keeping only the one with the highest IoU.

4.3 COCO Format and Datasets distribution

There are certain formats in which data can be fed to a model such as a YOLO format, PASCAL VOC, COCO [13], etc. Detectron2 accepts datasets formatted using COCO notation. It consists of a JSON file that includes general information about the dataset and all the details for the images such as width, height, filename, and id. It contains also the data for the annotations for every single image as bounding box coordinates and its category. While generating the synthetic data with some basic python scripting we populated one file for training and one for validation. An example taken from the 'train.json' file can be seen below:

```
"images": [
    {
        "file_name": "0005-1.0005-2.png",
        "width": 1920,
        "height": 1080,
        "license": "1",
        "id": 2
    }, ...
],
"annotations": [
    {
        "id": 2,
        "image_id": 2,
        "category_id": 0,
        "bbox": [648, 799, 193, 257],
        "area": 49601,
        "iscrowd": 0
    }, ...
]
```

We used a splitting of 80/20 respectively for train and validation, giving us the following distribution on the three datasets.

Dataset	Train	Validation	Total
Font	400	100	500
Handwritten	400	100	500
Hybrid	124	30	154

Table 4.1: Number of examples for train and validation in each dataset

As stated before with a complete automated generated dataset there is no limit on the number of examples we can generate. For fine-tune purposes even a moderate size dataset can be used, so we chose to stop at 500 examples. For the hybrid generation, we stopped at 154 because we had 14 usable backgrounds on which 10 synthetic PLI has been added. Empty backgrounds with no labels have been added to provide examples where no object should be detected by the model.

4.4 Training

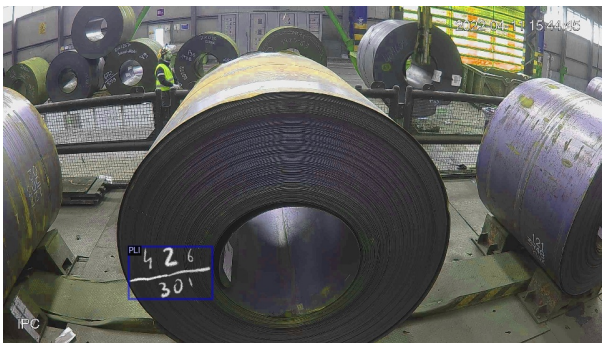
For each dataset the coco-formatted train and validation collections need to be registered so Detectron can correctly load them.

```
train = 'pli_train'
validation = 'pli_validation'
register_coco_instances(train, {}, './dataset/hybrid/train.json',
    './dataset/hybrid/imgs')
register_coco_instances(validation, {}, './dataset/hybrid/validation.json',
    './dataset/hybrid/imgs')

def plot_samples(dset_name, n=1):
    dset = DatasetCatalog.get(dset_name)
    dset_metadata = MetadataCatalog.get(dset_name)
```

```
    for sample in random.sample(dset, n):
        image = cv2.imread(sample['file_name'])
        viz = Visualizer(image[:,:,::-1], metadata=dset_metadata, scale=.5)
        viz = viz.draw_dataset_dict(sample)
        plt.figure(figsize=(15,20))
        plt.imshow(viz.get_image())
        plt.show()
```

Using `plot_samples` we can double-check if an image and its relative annotation have been correctly loaded.



Detectron needs to be configured to train our dataset. To do so we got the basic configuration from the library and merge it with the chosen model structure that we want to fine-tune. All the models present in the model zoo of the Detectron library are pre-trained on COCO dataset. Lastly, the weights of the chosen model need to be loaded into the configuration.

```
def get_train_cfg(config_file, checkpoint_url, train, validation, classes,
    output):
    cfg = get_cfg()
    cfg.merge_from_file(model_zoo.get_config_file(config_file))
    cfg.MODEL.WEIGHTS = model_zoo.get_checkpoint_url(checkpoint_url)

    cfg.DATASETS.TRAIN = (train, )
    cfg.DATASETS.TEST = (validation, )
```

```
    cfg.DATALOADER.NUM_WORKERS = 4
    cfg.DATALOADER.FILTER_EMPTY_ANNOTATIONS = False
    cfg.SOLVER.IMS_PER_BATCH = 4
    cfg.SOLVER.BASE_LR = .0025
    cfg.SOLVER.MAX_ITER = 1000

    cfg.MODEL.ROI_HEADS.NUM_CLASSES = classes
    cfg.MODEL.DEVICE = 'cuda'
    cfg.OUTPUT_DIR = output

    return cfg
```

Next, the train dataset that has been previously registered is passed over the configuration, in those some of the examples do not contain any label so we flagged false the `FILTER_EMPTY_ANNOTATIONS`, to signal Detctron to evaluate also these cases. The batch size, learning rate, max iterations number are respectively assigned using `IMS_PER_BATCH`, `BASE_LR` and `MAX_ITER`. Finally, we can specify the number of classes, in our case only one, and force Detectron to use GPU computation to achieve faster training time via 'cuda' directive.

Lastly, we loaded the configuration to Detectron *DefaultTrainer* and started the training on Faster RCNN model for 1000 iterations with a starting learning rate of .0025 and 4 images per batch. The whole training had an approximate duration of 12 minutes for each dataset on a 12GB NVIDIA Ampere architecture gpu.

Chapter 5

Results

Before showing the results on the validation and test datasets we should introduce the different metrics defined by COCO for object detection.

5.1 Notation

Precision is defined as the number of true positives divided by the sum of true positives and false positives:

$$precision = \frac{TP}{TP + FP}$$

- **Average Precision (AP)**: Average Precision is averaged over all categories. Traditionally, this is called "mean average precision" (mAP). COCO makes no distinction between AP and mAP
- **AP50**: Average Precision with a fixed IoU of .50
- **AP75**: Average Precision with a fixed IoU of .75
- **APs**: Average Precision for small objects: $area < 32^2$
- **APm**: Average Precision for medium objects: $32^2 < area < 96^2$
- **APl**: Average Precision for large objects: $96^2 < area$

5.2 Validation

After each training, a cross evaluation on the validation dataset is performed to check if the model correctly detects data from the same distribution, so if it is accurate on synthetically generated images. To do that the configuration that detectron saves at the end of the training phase needs to be loaded and the fine-tuned model's weights need to be merged into the configuration. After that, it can be passed to the Detectron *DefaultPredictor* so it can make predictions on the dataset. As evaluator we have chosen the *COCOEvaluator* also provided by Detectron.

AP	AP50	AP75	APs	APm	APl
63.868	98.990	70.191	nan	nan	63.868

Table 5.1: Evaluation of the Font-source validation dataset

AP	AP50	AP75	APs	APm	APl
60.572	90.260	72.459	nan	nan	60.572

Table 5.2: Evaluation of the Handwritten-source validation dataset

AP	AP50	AP75	APs	APm	APl
78.419	100.000	93.388	nan	81.238	77.380

Table 5.3: Evaluation of the Hybrid validation dataset

We achieved an average precision (AP50) for an IoU of .5 above 90% for all three models and an average precision (AP75) above 70% for an IoU of .75, which means that our models correctly detect our synthetic generated PLI.

5.3 Model inference on test dataset

Finally, the three models can be evaluated on the real data. To do so we imported the 23 pictures taken from the original dataset introduced in section 2.2 that visually presented a PLI into *LabelME* to manually draw a bounding box around them. LabelME generates a JSON file for each imported image, to aggregate each one of these into the test dataset we used a library called *labelme2coco*. As we have done for the validation and the training dataset also this test dataset has been registered in Detectron for use. Differently from the validation results we will present only the values for AP50 and AP75.

Model	AP50	AP75
font-source	36.600	25.800
handwritten-source	25.600	19.200
hybrid	84.00	69.900

Table 5.4: Performance on real data for each model

Results are clear, font-source scores better on the handwritten-source dataset but fails on providing viable precision, on the other side hybrid dataset stands out with its 84% on AP50 metrics. To better visualize the performances of the hybrid trained model we will show some crops produced by predictions on real data. For each image in the test dataset we will show only the detection with the highest prediction confidence.

Figure 5.1: Predictions of the hybrid trained model on real data

On 23 images contained in the test dataset the model detect a possible PLI on 22 of them, of those 18 are correctly cropped PLI so 78% of the overall images.

Chapter 6

Ablation studies

The scope of this chapter is to study how the model performance change by varying Detectron's configuration parameters. We will perform these testing only for the model trained on the hybrid dataset since it has the best scores of all three. To be precise we will study its behavior changing the ConvNets backbone, learning rate and the number of iterations. For brevity reasons, only the results on the real data are shown, without showing also the validation step for each different model.

6.1 Varying the backbone

For this test we have left the iterations and learning rate unchanged respectively at 1000 iterations and .0025 for the base learning rate.

ConvNets Backbone	Learning Schedule	Depth	AP	AP50	AP75	APm	APl
Faster R-CNN FPN	3x	50	53.542	84.004	69.948	40.153	60.988
Faster R-CNN FPN	3x	101	48.426	75.372	61.437	53.313	49.050
RetinaNet	3x	50	47.051	69.424	64.177	39.612	50.394
RetinaNet	3x	101	54.098	74.279	62.052	34.964	60.704

Table 6.1: Scores on the hybrid dataset changing the network backbone

6.2 Varying the iteration number

Varying the iterations leaving fixed the base learning rate at .0025 and the ConvNets backbone is a Faster R-CNN FPN (50 layers) with a learning schedule 3x.

Iterations Number	AP	AP50	AP75	APm	APl
500	48.739	79.178	63.048	38.036	61.885
1000	53.542	84.004	69.948	40.153	60.988
1500	60.117	88.606	77.008	44.961	68.102
2000	57.991	83.922	74.637	48.980	63.861

Table 6.2: Scores on the hybrid dataset changing the number of iterations

6.3 Varying the base learning rate

Varying the learning rate leaving fixed the iterations at 1000 and the ConvNets backbone is a Faster R-CNN FPN (50 layers) with a learning schedule 3x.

Learning Rate	AP	AP50	AP75	APm	APl
.025	47.895	68.277	65.192	35.535	53.113
.0025	53.542	84.004	69.948	40.153	60.988
.001	55.170	82.800	69.367	42.847	63.880
.00025	26.951	54.483	19.939	14.441	38.830

Table 6.3: Scores on the hybrid dataset changing the learning rate

FPN: Use a ResNet+FPN backbone with standard conv and FC heads for mask and box prediction, respectively.

Chapter 7

Conclusions and future work

In this work, we first defined the problem, its requirements and the limitations of the possible approaches. We provided proof of work on a limited portion of the dataset using non-deep learning computer vision methodology and showed how this approach failed when scaled on the whole dataset, which gave us better insights on the overall challenges bounded to the detection of a well-defined handwritten production lot. Next due to the absence of a robust collection of real annotated data we provided multiple approaches to generate custom datasets. Dividing the generative task into 2 categories, *Automated Generation* and *Hybrid Generation* and implementing 3 different solutions to generate datasets to be later used to fine-tune a convolutional neural network. We showcased Detectron2 framework which hosted our training and experiments. After a brief introduction on how to approach object-detection in a deep learning way, we trained on the previously generated datasets. After the validation phase, the obtained models have been tested on real data. Based on the performance achieved by each dataset the hybrid generated one has been the choice to go for the last section. We performed ablation studies changing the underlying ConvNets used, the iteration steps, and the base learning rate, providing even more insights on the overall work. Future horizontal expansion of this work could the creation of a newly generated dataset that combines the fidelity of hybrid generation with the scalability of automated generation, this could be achieved using 3d scenery generation software like Blender, first creating a torus object, that mimics the steel coil, and then apply a generated PLI as a texture to it. To vertically

expand the project digits recognition on the cropped PLI is the next obligatory step, it could be also approached using Detectron separating each digit in a single class, and then recreating the PLI number having the right writing orientation.

Bibliography

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

[2] D. H. Ballard. *Generalizing the Hough Transform to Detect Arbitrary Shapes*, page 714â725. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987.

[3] A. Botev, M. Bauer, and S. De. Regularising for invariance to data augmentation improves supervised learning, 2022.

[4] L. Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[5] R. O. Duda and P. E. Hart. Use of the hough transformation to detect lines and curves in pictures. *Commun. ACM*, 15(1):11â15, jan 1972.

[6] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The PASCAL Visual Object Classes Challenge 2007 (VOC2007) Results. http://www.pascal-network.org/challenges/VOC/voc2007/workshop/index.html.

[7] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Results. http://www.pascal-network.org/challenges/VOC/voc2012/workshop/index.html.

[8] R. Girshick. Fast r-cnn, 2015.

[9] R. Girshick, I. Radosavovic, G. Gkioxari, P. Dollár, and K. He. Detectron. `https://github.com/facebookresearch/detectron`, 2018.

[10] K. He, G. Gkioxari, P. DollÃ¡r, and R. Girshick. Mask r-cnn, 2017.

[11] P. V. Hough. Method and means for recognizing complex patterns. 12 1962.

[12] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. DollÃ¡r. Focal loss for dense object detection, 2017.

[13] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, and P. DollÃ¡r. Microsoft coco: Common objects in context, 2014.

[14] S. I. Nikolenko. Synthetic data for deep learning. *CoRR*, abs/1909.11512, 2019.

[15] S. J. Pan and Q. Yang. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, 2010.

[16] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[17] A. Ratner, C. De Sa, S. Wu, D. Selsam, and C. RÃ©. Data programming: Creating large training sets, quickly. 2016.

[18] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection, 2015.

[19] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks, 2015.

[20] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks, 2013.

[21] Y. Wu, A. Kirillov, F. Massa, W.-Y. Lo, and R. Girshick. Detectron2. `https://github.com/facebookresearch/detectron2`, 2019.