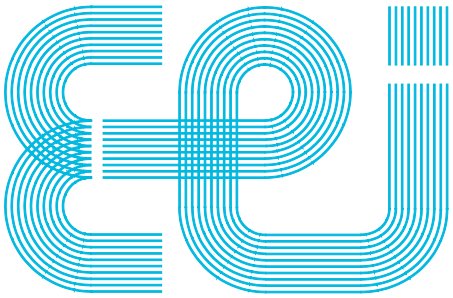


Escola de Enxeñería Industrial

TRABALLO FIN DE MESTRADO

Convolutional neural networks for object detection in subterranean environments

Master's Degree in Industrial Engineering

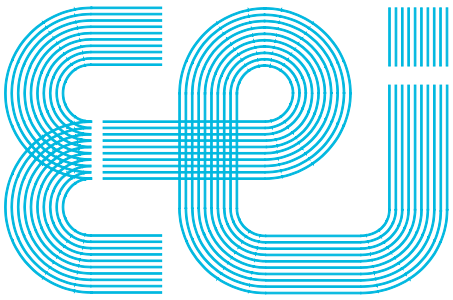
STUDENT: Pablo Santana González

SUPERVISORS: Enrique Paz Domonte

George Nikolakopoulos

Christoforos Kanellakis

Universidade Vigo



Escola de Enxeñería Industrial

TRABALLO FIN DE MESTRADO

Convolutional neural networks for object detection in subterranean environments

Master's Degree in Industrial Engineering

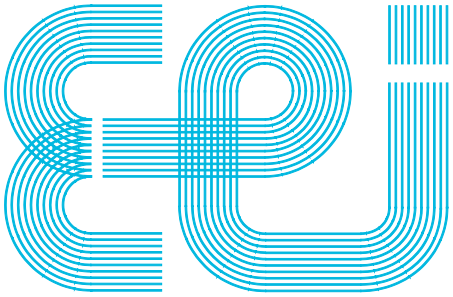
Document

GLOBAL INDEX

Universidade Vigo

GLOBAL INDEX

Additional resources hosted at `github.com/pabsan-0/sub-t`



Escola de Enxeñería Industrial

TRABALLO FIN DE MESTRADO

Convolutional neural networks for object detection in subterranean environments

Master's Degree in Industrial Engineering

Document

ABSTRACT

Universidade Vigo

ABSTRACT

This academical project aims to explore the field of applied machine vision by approaching a challenging computer vision problem, consisting on the detection and localization of items within a closed environment modelling subterranean settings.

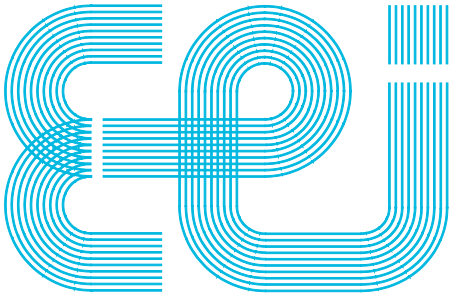
Among other risk factors, subterranean environments are characterized by lack of natural light, track roughness and isolation or restriction in space. These hazards vary drastically across domains and can degrade or change over time, often posing too high of a risk for having personnel enter such places. Hence, having unmanned vehicles quickly map, navigate and search underground settings has been a rising interest in the world of robotics. In a normal setting, this type of task is usually achieved by a robot or agent equipped with varied sensory, among which GPS is the standard for solving robot localization, which however cannot be used underground. While the combination of multiple sensors leads to the most robust results of simultaneous mapping, localization and item search, state-of-the-art computer vision techniques of today allow for the solution of all these tasks only on image input, even GPS-free localization.

First, several object detection models are researched in order to identify, from image or video data, instances of a certain set of artifacts inside an image. This set is extracted from the DARPA Subterranean Challenge (backpack, drill, helmet, rope, survivor and fire extinguisher), meant to represent objects or features of interest that could sensibly appear in subterranean environments during exploration or rescue tasks. From the proposed DARPA artifacts, those whose detection is not to be image-based (CO_2 leak, vents and smartphone) are discarded. For these artifacts, both synthetic and real data is produced in order to train and compare a set of state-of-the-art neural network-based object detectors (YOLO, EfficientDet) on a case-specific baseline dataset, by means of benchmarking their detection accuracy and inference speed.

Further work attempts to solve agent and item localization in the horizontal plane of a known, mapped environment. Localization is approached by a Kalman-filtered fiducial landmark system which provides, from live image feed, filtered estimates of the pose of the agent, experimentally represented by a human-driven camera. The same images are fed to a CNN-based object detector, which finds the position items of interest with respect to the camera. Camera coordinates of these detections are then converted to real-world coordinates, allowing for the construction of a top-view, two-dimensional likelihood map which marks the areas where certain items have been found to be. Finally, software to support these steps is designed and implemented in a multi-processing parallel pipeline that allows the localization plus item search task to be performed in real time with consumer technology.

Code and resources are available at `github.com/pabsan-0/sub-t`.

Keywords: Object Detection, YOLO, EfficientDet, Robot localization, Item search



Escola de Enxeñería Industrial

TRABALLO FIN DE MESTRADO

Convolutional neural networks for object detection in subterranean environments

Master's Degere in Industrial Engineering

Document

REPORT

Universidade Vigo

CONTENTS

FIGURE INDEX

TABLE INDEX

EQUATION INDEX

1 INTRODUCTION

1.1 Problem definition

1.1.1 Motivation

This academical project aims to explore the field of applied machine vision by approaching a challenging computer vision problem, consisting on the detection and localization of items within a closed environment.

Within the current state-of-the-art research in the field of robotics, autonomous navigation, mapping and item search in underground settings has been a rising interest. Subterranean environments are characterized by lack of natural light, track roughness and isolation or restriction in space, among others. These hazards vary drastically across domains and can degrade or change over time, often posing too high of a risk for having human personnel enter such places.

A robot is a goal-oriented machine that can sense, plan and act autonomously [1]. The design of a robot application must answer a series of questions: where the robot is, what is its surrounding environment, where should the robot go, etc. A classical robotic architecture is depicted on Figure 1, where the relationship between the different tasks a robot needs to perform is shown. The perception layer of a generic robot device performing autonomous navigation, mapping and item search tasks is usually composed of:

- **Item localization**: Usually based on vision, sound or radio signals.
- **Mapping**: Commonly performed with LiDAR.
- **Localization**: Generally achieved with global positioning systems (outdoors only).

While the proper combination of these techniques leads to the most robust results of simultaneous mapping, localization and item search, state-of-the-art computer vision techniques today allow for the solution of all these tasks only on image input.

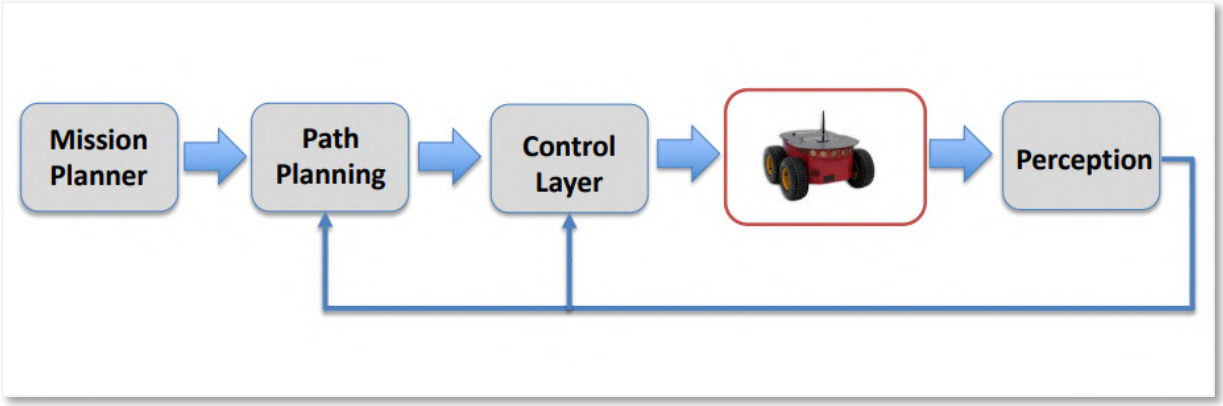


Figure 1: A classical robotic architecture
Source: [1]

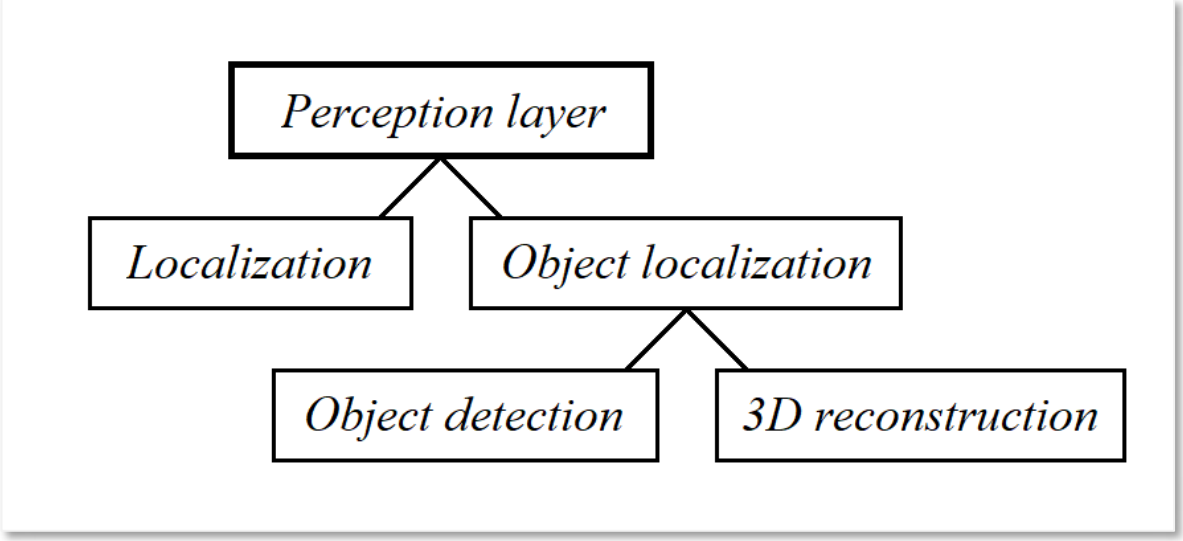


Figure 2: An example perception layer for an item search task
Source: Self-made

A pure computer-vision based approach for item search is of interest mostly in terms of research. Underground, vision localization provides a viable alternative to GPS, while it is true that one of its major pitfalls is the limitation of light. While some vision techniques, especially those based on pixel gradients, are robust even if light is scarce, a minimum amount is still required for vision techniques to perform properly.

An example perception layer for approaching the problem of item search in a mapped underground environment can be exploded, as shown in Figure 2, in the following tasks:

- **Localization**: The problem of identifying the robot's pose with respect to a reference coordinate frame. A common visual localization procedure is dead reckoning with fiducial markers as landmarks.
- **Object localization**: The problem of identifying the position of items. In a vision approach, this position is usually obtained with respect to the camera (mounted on the robot) and thus requires the localization problem to be solved to map items to absolute coordinates. Object localization can, again, be exploded in:
    - **Object detection**: Which is typically performed via object detection pipelines based on convolutional neural networks. Object detection consists on finding objects in 2D images and drawing square bounding boxes around them.
    - **3D reconstruction**: Which consists on linking the pixel-level information within an image to real world coordinates. In this case, mapping the bounding boxes obtained by object detection to real world locations.

While deep learning models for integrated visual object localization are just starting to appear [2], they share strong roots with the recent yet slightly older field of object detection. Convolutional object detection pipelines have led the state of the art of detecting items in images and video for the past years and suppose an advanced topic within the field of deep learning. It is found adequate for this academical work to explore these detection models instead of leaping for an integrated approach, which would also require the craft of novel highly complex data.

1.1.2 Object detection approach to the DARPA Subterranean Challenge

Towards approaching a real, domain-specific problem within object detection, this work addresses the research and comparison of various state-of-the-art object detection models for a relaxed realization of the DARPA Subterranean Challenge.

The DARPA Subterranean Challenge [3] aims for the localization of various artifacts in subterranean (subT) environments by unmanned vehicles. Towards this challenge, the term *artifact* references elements of interest that can reasonably be found in such places, and are divided in four different categories, depending on the nature of the subT environment they might appear: tunnel, cave, urban or common to all of them. Each of these artifacts has a specific interest in subT exploration, so that its finding would provide relevant information under high-risk situations in unpredictable and dangerous subT spaces.

The DARPA SubT Challenge is directed towards both simulation environments and real-life circuits. Official virtual circuits have been modelled for running in common robotics simulators, such as *Gazebo*, and are available online, same as virtual models for each of the artifacts. For real-life scenarios, the artifacts are defined to be very specific commercial products, on which the virtual models are based, so that the artifact appearance and characteristics are standard.

The artifacts to be detected towards the SubT Challenge are outlined in Table 1. As stated in [3], not all items are supposed to be detected visually, but by the usage of different sensory. By only using a camera, the detection of the following items shows specific difficulties, and thus their detection is removed from this work's scope:

- **CO_2 mist**: For which a gas detector is needed.
- **Vent ducts**: Expected to be different in any new deployment environment.
- **Cell phone**: Which is supposed to be playing audio while emitting Wi-Fi and Bluetooth signal. While its detection could be performed with only a camera, its small size and the arbitrary color pattern of its screen make its detection challenging.

The detection for the rest of the items is to be approached as neural network based object detection problem from image data, by using the very DARPA artifacts as training samples. The completion of this stage is of research interest, its desired a priori outcomes being:

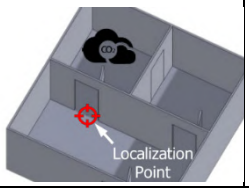
- A representative dataset for object detection with the artifacts of this challenge.
- A precision-speed benchmarking for a domain-specific 6-class detection problem.

1.1.3 3D reconstruction and object localization

From the detection outcomes of a suitable object detection model, to be defined after the detector benchmarking, it is desired to design the rest of the visual perception layer so that items can be located in a given, mapped environment. This pipeline is to be case and agent agnostic, meaning that it will comprise arbitrary object localization without regard to the rest of the robotic architecture (depicted in Figure 1) or the set of items to be detected. As stated, agent localization and 3D reconstruction will be component tasks within this layer.

A particular way to perform this localization, as well as its expected outcome, are not defined a priori but expected to be found by research and experimentation.

Table 1: DARPA SubT Artifact Detection Challenge artifacts

Name	Image	Description
Survivor		Represented by a 180cm high thermal mannekin wearing a high-visibility jacket.
Cell phone		Defined to be a Samsung Galaxy J8 smarthphone that will be playing full-screen video with audio during the scored run, screen visible pointing outwards. Meant to represent traces of human activity.
Backpack		Red Tape color JanSport Big Student Backpack, with red front and black back. Meant to represent traces of human activity.
Drill		Cordless Black&Decker orange drill with battery attached. Meant to represent sets of manual or powered hand tools that may be present in the subT environment.
Fire extinguisher		Hand-held red fire extinguisher with black hose. Meant to give hints about the location of general emergency gear in tunnels.
Vent duct		Typical supply of fresh air that may identify places where the air quality is superior and provide hints to possible escape routes to the surface.
Helmet		White caving helmet with attached headlamp. Meant to represent traces of human activity.
Rope		Coil of blue rope. Meant to represent traces of human activity and possible vertical passages.
Gas		A CO_2 emmitter device that simulates a range of hazardous air quality conditions.

Source: [3]

1.2 Scope & goals

The main goals of this project are stated as follows:

I. Approach the DARPA Subterranean challenge under a object detection approach:

- Build a representative baseline dataset for the obejcts to be detected by means of gathering existing/generating new data.
- Experimentally compare several state-of-the-art object detection models for the created baseline dataset and discuss their performance in this particular application.

II. Desing a perception layer built around an object detector for item localization:

- Study the localization problem of a vision device in an underground setting and achieve camera localization based on a fiducial marker system.
- Develop a perception pipeline for item localization in a mapped indoors real envionment for an agent receiving visual input from a single camera.

1.3 Methodology

This project starts with the well-defined objective of benchmarking object detection models towards the DARPA challenge simplification, later expanded in an open-end approach towards developing a visual perception layer as advanced as possible within the available time. The main initial planning towards reaching these objectives comprises the following milestones:

- Research on object detection to identify relevant models and methodologies
- Crafting data representative to the visual DARPA Challenge
- Train, evaluate and discuss results for different state-of-the-art detectors

Then progressively expanding through:

- Perform real-time object detection in a remote computing device
- Solve the localization problem of a camera device in a mapped environment
- Identify the position of items in a mapped environment

Figure 3 shows the tracking of the project through its development.

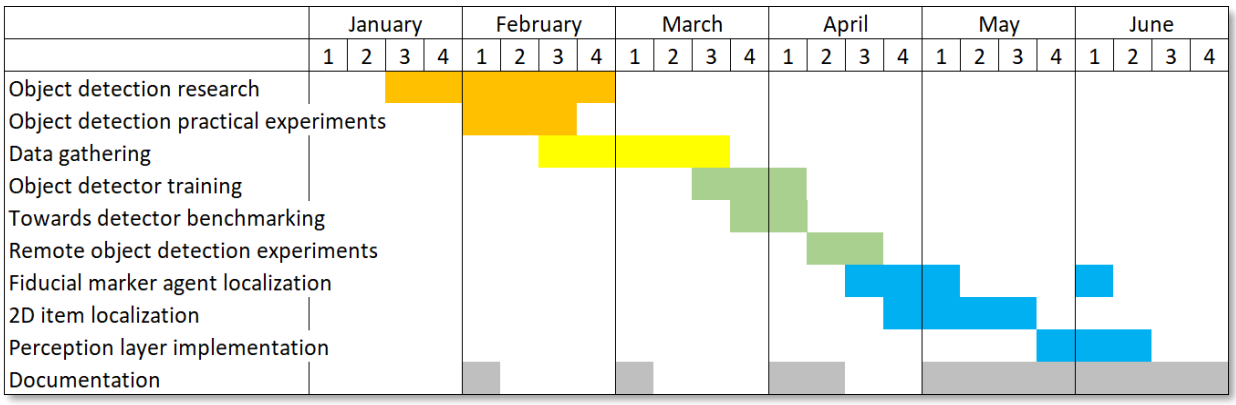


Figure 3: Project tracking
Source: Self-made

2 BACKGROUND

2.1 Artificial intelligence

2.1.1 Artificial intelligence

Artificial intelligence is the "effort to automate intellectual tasks normally performed by humans" [4], and is in itself a loose concept that does not particularly define a specific approach for solving real problems. Rather, it is a collection of solutions that try to have a machine make decisions similar to those that humans would. The author F. Chollet proposes what can be interpreted as a simple yet not exhaustive branching of artificial intelligence by defining the terms machine learning and symbolic AI in [4]:

- *Symbolic AI* is an approach based on sets of hand-crafted rules that a machine would verify for solving a given problem. A great example of this approach are chess bots, which decide on what move to make by simulating every possible choice and then actually making the one that produces the best simulated outcome. There is a programmer involved that has to write the rules that define these simulations: how can the pieces move, the rewards of killing each of the enemy pieces, etc. In the end, the machine is only able to replicate the scenarios that his designer has considered.
- *Machine learning* techniques do not rely on hand-crafted rules and instead try to infer models that fit to some example data. Compared to symbolic AI, they need little human input and are able to adapt and learn features from data. While they require less cognitive effort in their design, machine learning applications are heavily dependent on the quality and representativeness of the data they are fed. With the currently available computing power and accesibility to data sources, machine learning techniques have become predominant in the context of AI nowadays.

2.1.2 Machine Learning

Machine learning (ML) comprises a set of techniques or algorithms that attempt to solve specific tasks from sets of data. A typical machine learning solution is usually implemented in multi-stage data-processing pipelines sometimes combining different ML models or algorithms, in such a way that input data is succesively transformed into representations that are more representative to the problem at hand until these serve to provide an output. Typically, machine learning solutions live in two states: development and deployment.

Development comprises the design and implementation of the stages of data processing, model training and validation of a machine learning application. ML models can be trained, tested and validated yet within a development context. Deployment starts once the algorithm has been tested with an acceptable performance as outcome, and the machine learning solution is used to fulfill the practical application it was designed for, usually but not always remaining stationary over time.

Since machine learning consists on inferring data-processing rules to generate an algorithm from some example data, a single machine learning model is composed of three basic elements. For providing an example, let's imagine an image classification task:

- **Input data samples**: Which are to be representative of what the trained algorithm will receive as input once deployed, since a machine learning algorithm can only "learn" from the data that it is shown. An example of input data are pictures of either dogs or cats.
- **Output data samples**: Labels that correspond to the desired output for a given input and define the *ground-truth* (what these inputs actually represent), for example a numeric variable indicating whether a picture contains a dog or a cat.
- **A performance measure for evaluating the algorithm**: Usually called *loss function,* for which typically minimum values correspond to best performance of the algorithm or model. Based on the value of the *loss*, the algorithm adjusts its parameters so that it will hopefully improve its performance on the next iteration. For image classification, a performance measure could be the number of missclassifications for a set of 100 samples.

According to their dependence on data, machine learning models can generally be classified in the following categories [4]:

- ***Supervised learning***: Where the model is used to map input data to a known set of targets. This is, the data used for training the model typically consist of a set of input-output examples. Applications that fall in this category are by far the most common inside machine learning approaches. Some tasks that supervised learning deals with are:
  - ***Classification***: where a discrete category is to be assigned to a piece of data.
  - ***Regression***: where a continuous numerical value is to be assigned to a piece of data.
- ***Unsupervised learning***: Where the model is used to find useful transformations of the input data, without requiring examples for the desired model output.
  - ***Clustering***: which is used to group batches of samples showing a common behavior.
  - ***Dimensionality reduction***: which is used to compress data into newer, smaller representations that still retain useful information.
- ***Reinforcement learning***: where an agent is expected to perform an action and, based on its performance, receives feedback on how to adapt its behaviour.

2.2 Deep learning

2.2.1 The basics of Neural Networks

Recalling the concept that machine learning aims to learn meaningul representations of data, *deep learning* is a subfield within machine learning which emphasizes learning many successive layers (hence the name *deep*) of increasingly meaningful representations [4], typically learned by models called *artifical neural networks* (ANNs).

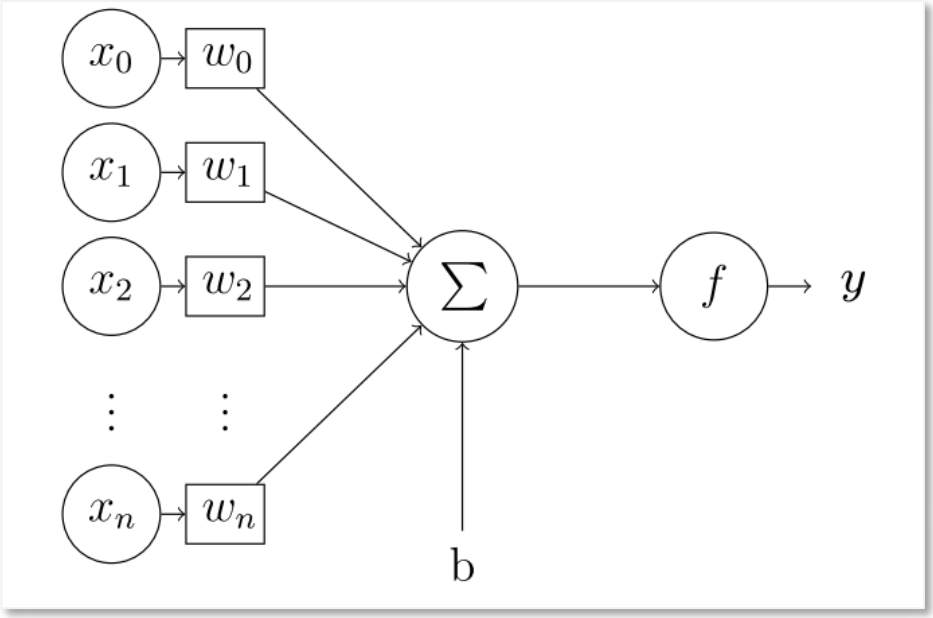


Figure 4: The perceptron
Source: [5]

Neural networks are generally comprised by stacks of neuron layers. The perceptron [6] is a simple mathematical model for the neurons in an ANN, a representation of which can be seen in Figure 4. When a whole layer can be modeled by a single perceptron, the mathematical expression that links this layer's input and output $\boldsymbol{x}$ and $\boldsymbol{y}$ is:

$$\boldsymbol{y} = f(\boldsymbol{x} * \boldsymbol{w} + \boldsymbol{b}) \quad (1)$$

Where we can find:

- $\boldsymbol{w}$: This layer's *weights* (tensor). To be inferred from $\boldsymbol{xy}$ examples.
- $\boldsymbol{b}$: This layer's *biases* (tensor). To be inferred from $\boldsymbol{xy}$ examples.
- f : This layer's *activation function* (non linear). Designer-defined.
- $\boldsymbol{x}, \boldsymbol{y}$: This layer's input and output, respectively. Provided as data.

In a supervised learning setting, the input and output vectors of this single-layer network $\boldsymbol{x}$ and $\boldsymbol{y}$ are known, and this layer's weights and biases would need to be inferred. This is done by comparing the predicted output of the network $\boldsymbol{y}$ with the example output data $\boldsymbol{y}_{\boldsymbol{training}}$ via a loss function $\boldsymbol{\ell}$, the output of which will be used to modify the values in $\boldsymbol{w}$ and $\boldsymbol{b}$. Within an ANN, a difference is made between the two following [4]:

- *Parameters* : Which are the *trainable* parameters of a network ($\boldsymbol{w}, \boldsymbol{b}$).
- *Hyperparameters* : Which are the *non-trainable* parameters of a network ($f, \boldsymbol{\ell}$).

A densely-connected layer is a layer in which all input values $\boldsymbol{x}$ are related to all output values $\boldsymbol{y}$. While more complex layer models exist, a simple densely-connected ANN layer can be entirely modelled by one single perceptron. In general, an ANN layer is simply a linear operation between the layer's input and its weights passed through a non-linear activation function.

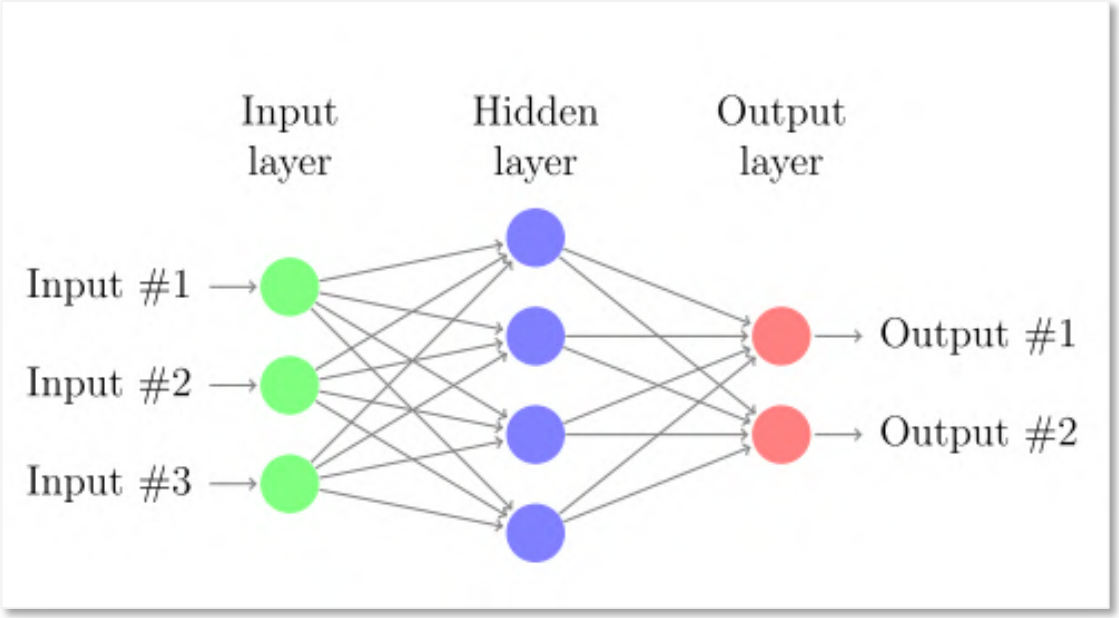


Figure 5: A very simple ANN architecture
Source: [5]

Figure 5 shows a densely-connected ANN architecture. Note that, in the perceptron example, the term layer was used as a reference for the operations happening in between each of the circular arrays in Figure 5, while now layer denotes a particular representation of the data within the network. In practice, the term layer is used indistinctively for both concepts (the preferred definition within this work being the former). The three layers, as pointed out in Figure 5, are:

- Input layer : Holds the input of the network $\boldsymbol{x}$.
- Hidden layer : Holds and intermediate representation of the data $\boldsymbol{h_1}$ (states).
- Ouput layer : Holds the output of the network $\boldsymbol{y}$.

Under this definition, every neural network has at least one input and one output layers, with an arbitrary number of hidden layers in between [5]. By adopting the perceptron model for a densely-connected layer, the ANN in Figure 5 can be mathematically expressed as:

$$\begin{aligned} \boldsymbol{h_1} &= f_1(\boldsymbol{x} * \boldsymbol{w_1} + \boldsymbol{b_1}) \\ \boldsymbol{y} &= f_2(\boldsymbol{h_1} * \boldsymbol{w_2} + \boldsymbol{b_2}) \end{aligned} \tag{2}$$

Where the output of each layer is directly fed into the next one, each with its own parameters and activation function. Note that, because any arbitary combination of succesive linear operations can be expressed as one single linear operation, to actually benefit from training multiple layers it is necessary that their activation functions are non-linear.

Thus, a simple ANNs is a particular architecture for a prediction model that aims to predict an output $\widehat{\boldsymbol{y_0}}$ from a given input $\boldsymbol{x_0}$, by applying a series of non-linear transformations to $\boldsymbol{x_0}$. These transformations are defined by the network's parameters, which are inferred in a non-deterministic manner by using computational algorithms from a set of examples $\boldsymbol{x_{training}}$, $\boldsymbol{y_{training}}$ in a so-called training stage.

2.2.2 Training a Neural Network. Stochastic gradient descent

The parameters of a neural network have been said to be inferred from data via computational algorithms. These algorithms used to train the network by updating the weights and biases in its layers are called *optimizers*. The working principle of a neural network is to minimize a loss function $\boldsymbol{\ell}$, so that the smaller $\boldsymbol{\ell}$ is, the closer the prediction $\hat{\boldsymbol{y}}$ will be to its expected ground-truth value $\boldsymbol{y}$ for an input $\boldsymbol{x}$. Thus, optimizers update the parameters of a network so that the loss $\boldsymbol{\ell}(\hat{\boldsymbol{y}}, \boldsymbol{y})$ is minimized [4].

During training, a dataset D composed by $\boldsymbol{xy}$ examples $\boldsymbol{x_{train}}, \boldsymbol{y_{train}}$ is fed into the network. Since the implementation of ANNs is usually done in high-dimensional matrices, multiple examples can be computed at once by adding an extra dimension to the input $\boldsymbol{x}$. While training, it is common to feed the network with batches of data taken from D, while inference is typically performed with single samples. When training, the action of feeding the network with a piece or batch of training data from D is called an *iteration*. Generally, a group of iterations that entirely go over D are what we call an *epoch*. A single iteration is composed of the following steps, parameters being usually randomized at the very beginning of training:

- *Forward pass*:
  - Feed the network with $\boldsymbol{x}$ and obtain a prediction $\hat{\boldsymbol{y}}$.
  - Compare the predicted and expected output via the loss $\boldsymbol{\ell}(\hat{\boldsymbol{y}}, \boldsymbol{y})$.
- *Backward pass*:
  - Compute the gradient of the loss with respect to each layer's weights $\frac{d\boldsymbol{\ell}(\hat{\boldsymbol{y}}, \boldsymbol{y})}{d\boldsymbol{w_i}}$.
  - Update the weights of each layer in the decreasing direction of the loss:

$$\boldsymbol{w_{i,new}} = \boldsymbol{w_{i,old}} - \gamma * \frac{d}{d\boldsymbol{w_i}} \boldsymbol{\ell}(\hat{\boldsymbol{y}}, \boldsymbol{y}) \tag{3}$$

Where γ is the *learning rate* of the network, a defined hyperparameter. The higher the learning rate the faster the network will converge to a set of final values for $\boldsymbol{w}$ (note the non-uniqueness of this solution). The method above described is the optimization technique known as *Stochastic Gradient Descent* or SGD [4].

On the choice of the learning rate, the following aspects need to be considered:

- If γ is too high: the network will converge quickly, but its solution will be unstable and significant wobbling will happen around the loss minimum, leading to a suboptimal solution.
- If γ is too low: the network will take longer to train and there is a chance that the loss $\boldsymbol{\ell}$ gets stuck inside local minima, again leading to a suboptimal solution.

Towards dealing with these problems, among others, other optimizers exist. These are commonly derived from SGD. Its most inmediate adaptation is *SGD with momentum*, which adds a factor accounting for the rate of change of the loss derivative: its intuition is to provide some kind of inertia to SGD, addressing the problem of $\boldsymbol{\ell}$ getting stuck in local minima [4]. Examples of other opmitizers are *Adagrad, RMSProp*, etc.

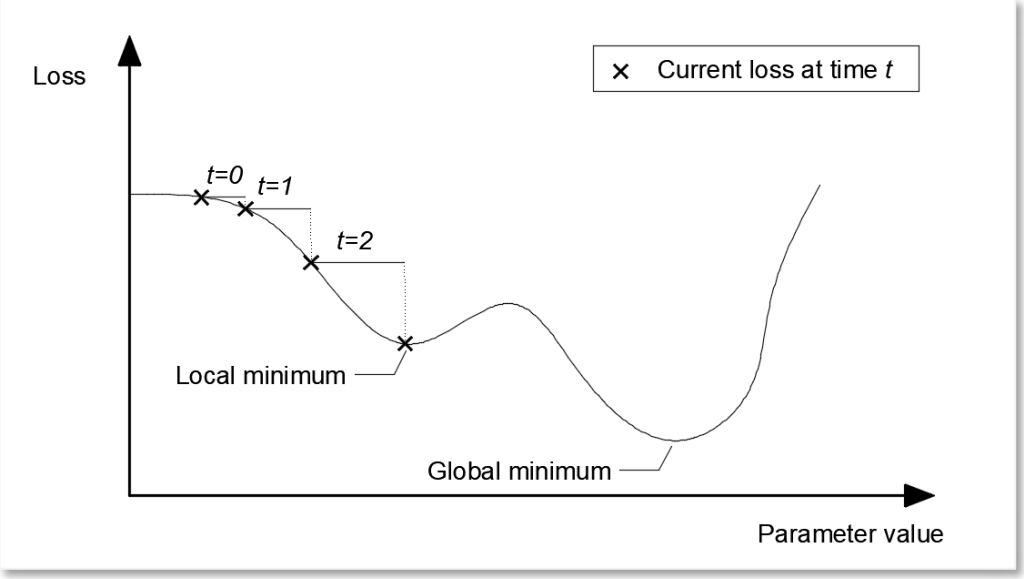


Figure 6: Single-dimensional loss during training
Source: Self-made

Figure 6 shows a simplified case in which a 1D loss is plotted, the markers indicating actual values that the loss has adopted over the training iterations. The step in between each of these is proportional to the learning rate γ and the loss at the current instant $\boldsymbol{\ell}$. The issues previously noted can easily be seen in this example, where the loss will be very prone to getting stuck in the local minima due to γ being too small and $\boldsymbol{\ell}$ being very flat in that particular area.

To propagate the loss backwards through the network, the *backpropagation algorithm* is typically used. Computing the derivatives of the loss with respect to each of the weights in a neural network is a very computationally expensive process. Backpropagation allows for an efficient way to perform this operation by optimizing the number of computations that need to be performed via storing some intermediate results in memory [4]. Figure 7 shows how this algorithm can compute the loss gradient with respect to each layer on a generic ANN.

2.2.3 Intuitions on activation functions

There are many possibities among the choice of activation functions. Some of the most common activation functions are shown in Table 2. As pointed in the example in Figure 5, activation functions are, in general, required to provide nonlinearities and expand the output range of the network. In general, a difference can be made depending on where an activation funtion is placed:

- Activations in hidden layers: This is, the activation functions of all the layers within the network but the last one. These activations, as previously stated, provide nonlinearities so that stacking layers is actually beneficial for increasing the prediction power. Proper choice of intermediate activation is crucial towards the trainability of the network.
- Activations on the output layer: Which define the shape of the network's output. Furthermore, the loss function must be chosen in accordance to this activation.

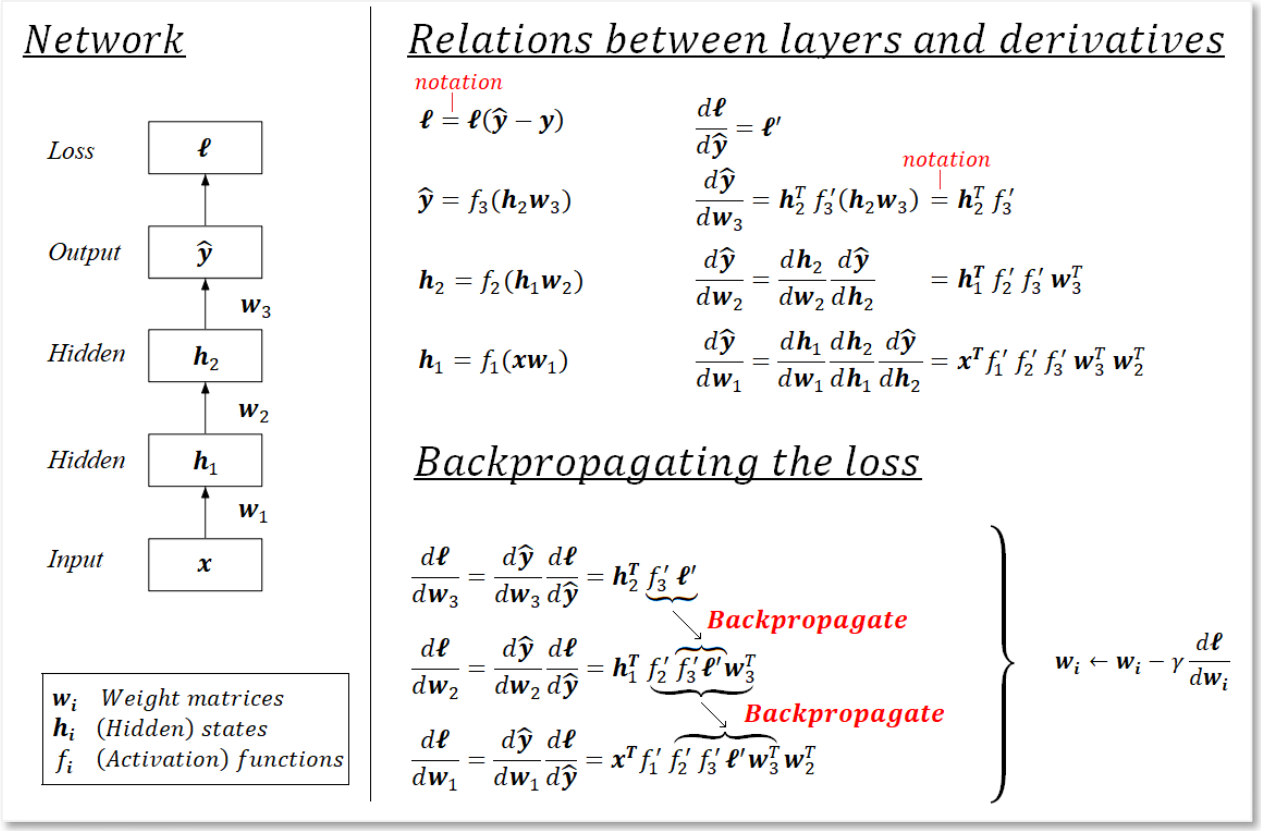
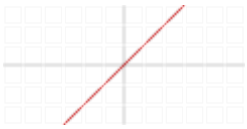
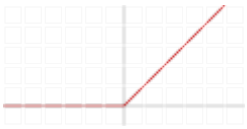
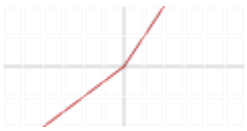
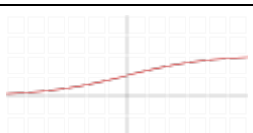
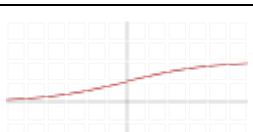


Figure 7: Backpropagation illustrated
Source: Self-made

Table 2: Common activation functions

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases}$
Leaky ReLU (*LReLU*)		$f(x) = \begin{cases} ax & x < 0 \\ x & x \geq 0 \end{cases}$	$f'(x) = \begin{cases} a & x < 0 \\ 1 & x \geq 0 \end{cases}$
Sigmoid		$f(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)[1 - f(x)]$
Softmax		$f_i(\boldsymbol{x}) = \frac{e^{x_i}}{\sum_{j=1}^{J} e^{x_j}} / \, i = 1, \ldots, J$	$f_i'(\boldsymbol{x}) = f_i(\boldsymbol{x})(\delta_{ij} - f_j(\boldsymbol{x}))$

Source: [7]

As stated, the activation functions in the hidden layers of a network play a crucial role on its trainability. First, lets introduce the concept of:

- Early layers : The closer a layer is to the input of the network, the earlier it is.
- Late layer : The closer a layer is to the output of the network, the later it is.

In a densely-connected ANN, when applyting the backpropagation algorithm to propagate the loss and update the weights, it generally holds that:

$$\frac{d\boldsymbol{\ell}}{d\boldsymbol{w_i}} \propto \prod_{\boldsymbol{L=i}}^{\boldsymbol{N}} f_L{}'(*)$$

- i : denoting a given layer of the network, $i = 1$ being the input layer and
- N : being the total number of layers.
- $f_L(*)$: denoting the activation function of the layer L,
- $*$: being its corresponding input in the forward pass and
- $f_L{}'(*)$: being its derivative at that point.

Meaning the derivative of the loss with respect to a given layer is proportional to the product of the outputs of the derivatives of the activations of later layers. This can be an issue if:

- $f_L'(*) \approx \mathbf{0}$: If activation functions tend to have a flat region, their derivative might become very close to zero, leading to the gradient progressively decreasing in earlier layers. This way, the earlier a layer is, the slower it will train. It is also possible that these layers stall and stop training. This is known as the *vanishing gradient problem.*
- $f_L{}'(*) \to \infty$: In the same way, if activations tend to have very steep slopes, their derivative might become very high, and the same problem arises only that this time the loss gradient will tend to a bigger value than the computations can handle. This is known as the *exploding gradient problem.*

Overall, this whole issue is known as the *unstable gradient problem*, a more detailed explanation of which can be found at [8]. While nowadays this problem has been addressed with modern optimizers and activation functions, training networks deeper than 10 layers used to be a big challenge that limited the potential of deep learning for many years [4]. Towards this aspect, good choices of intermediate activation functions are, for example: *rectified linear unit* (ReLU), *leaky ReLU,* etc [9].

Regarding the choice of last-layer activations, imagine a ternary classification problem in which we take a series of values $\boldsymbol{x}$ to output a possible class $\boldsymbol{y}$. Typically, we would encode the network predictions $\hat{\boldsymbol{y}}$ as a vector $\hat{\boldsymbol{y}} = (y_1, y_2, y_3)$. If, for instance, a *softmax* function is used at output layer, the outputs will verify $y_1 + y_2 + y_3 = 1$, and thus this output can be understood as the predicted "probability" that a sample $\boldsymbol{x}$ belongs to each of the classes y_1, y_2, y_3. Indeed, the very same problem could be formulated as a regression or even a discrete classification problem, but in some cases (specially for n-class classification, $n > 2$) the probability intuition can be more interesting. It is important to note that the chosen representation of the output strongly conditions the loss function to be used.

2.2.4 Optimization and generalization. Regularization

As it has been stated, neural networks are prediction architectures that infer their prediction rules from a series of data D_{train}, composed of example pairs of input-output x_{train}, y_{train}. These prediction rules are computed by applying a penalty via a loss function that relates the ground-truth examples $\boldsymbol{y_{train}}$ to their predictions $\boldsymbol{\hat{y}_{train}}$, so that the network can only "learn" from this example data D_{train}.

Even with outstanding predictions on D_{train}, it is not guaranteed that the learned parameters will offer a good performance when approaching new data. This issue is known as *overfitting*. Let's define the following characteristics of a neural network [4]:

- ***Optimization***: the optimization power of a neural network is its ability to fit and effectively learn from a set of data. Imagine a multi-dimensional regression task: one small network might not have enough parameters to hold all of the features that are representative towards this complex regression. Optimization is the capability of a machine learning model, in this case a neural network, to hold representative information and thus learn efficiently from a set of data D_{train}, in such a way that the loss $\boldsymbol{\ell}(\boldsymbol{y_{train}}, \boldsymbol{\hat{y}_{train}})$ is minimized significantly.
- ***Generalization***: the generalization power of a (trained) neural network is its ability to have a good performance when facing new, unseen data.

Indeed, the most important quality of a neural network model is its generalization power. In general, neural networks are prediction tools meant for deployment on given tasks, in which input stimuli is very unlikely to be exactly equal to the training data D_{train} used for tuning the model. A neural network that has high optimization power (i.e., good performance on training data D_{train}) but poor generalization power (i.e., poor performance once deployed) is said to suffer from *overfitting*.

Note that optimization is also a desired and important quality, since we need the model to be able to learn all of the representative features in D_{train}. In early deployment stages, overfitting is desired to happen at first on training data: this means that the network has enough optimization power, which is the ability to "learn from" or "memorize" D_{train} . If a model can't overfit on a training dataset D_{train}, either:

- $\boldsymbol{D_{train}}$ **is not rich enough**: Meaning that the provided inputs $\boldsymbol{x_{train}}$ do not contain enough information to define the expected outcomes $\boldsymbol{y_{train}}$.
- **The network architecture is not adequate**: The number of parameters might be too low or the overall architecture design might not be suitable for the problem at hand.

These should be checked via training a model and testing it using a meaninful performance measure. Specially when working with high-level neural network implementations, loss functions are sometimes too detached from intuitive metrics. In case of image classification, *cross-entropy* [4] might be a great loss towards training, but the percentage of missclassifications is a more human-readable metric on which decisions can be made. "Is 80% accuracy enough for our purposes?" is a much more reasonable thought than "Is a loss of $1.9542 * 10^{-6}$ good enough for our purposes?".

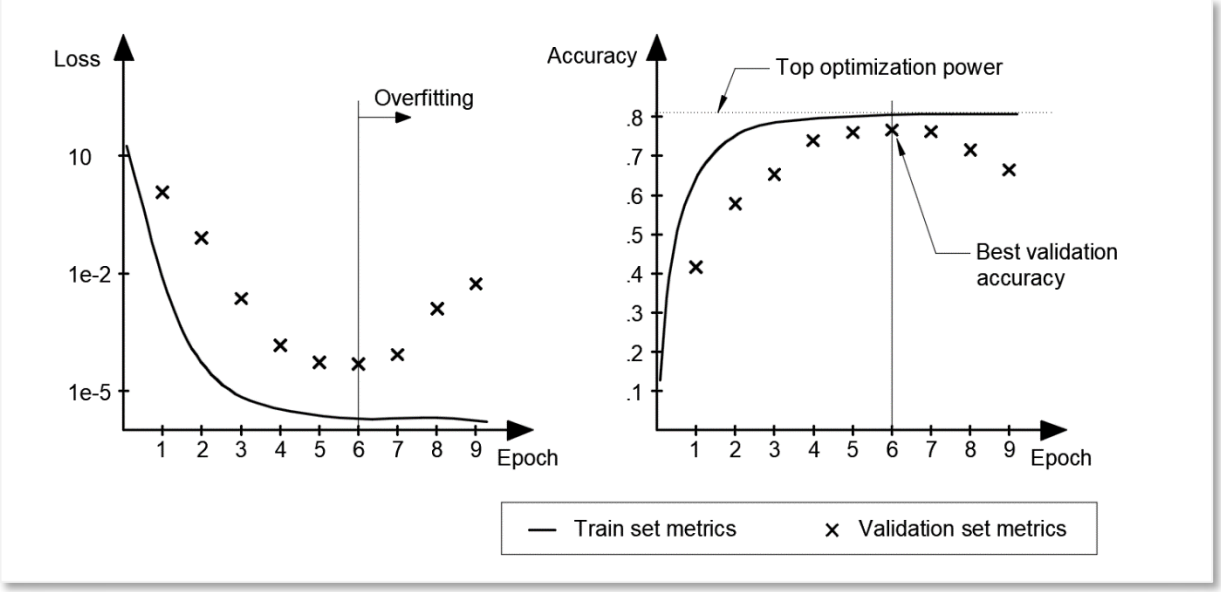


Figure 8: Metrics of interest during the training of a neural network
Source: Self-made

Since the most important feature of a neural network is its generalization power, its overall performance cannot be measured by using the same data used for training. Furthermore, because we can train a neural network for as long as we wish (repeating D_{train}), if the optimization power is enough any network is guaranteed to overfit after a number of training epochs, hence being critical for the quality of a model that training is stopped at the right moment. To adress these issues, it is common to divide the available set of data D in the following splits [4]:

- ***Training set* (D_{train})**: Data destined to train the network.
- ***Test set* (D_{test})**: Data destined to evaluate a trained model. It is critical for the network to be completely oblivious of these data, so the testing stage is performed on completely new, unseen data. Thus, the test results are representative of the generalization power of the model, which is its most important quality.
- ***Validation set* (D_{valid})**: Data used to track a network's performance during training. While the network will not infer any rule from the validation set, it is important to make a difference between test and validation. The validation set is used as a tool for decision-making about the model under development, and thus some *information leak* happens. The validation set spares some extra data so the testing split is completey unknown to the model.

Two plots of a hypothetical training run for an image classification problem are shown in Figure 8, representing an arbirary loss function and accuracy for the data in both D_{train} and D_{valid}. From these, we can diagnose our trained model by paying attention to the following:

- Is the top optimization power good enough?
- Is the model learning from D_{train} so that it can improve on D_{test}?
- How long does it take for my model to overfit?

While possible, it is highly unlikely that unseen data reaches the overfitted accuracy obtained for D_{train}, so that the top training accuracy generally represents an upper bound of the model performance. This allows diagnosing the chosen architecture and training data. A significant increase in validation accuracy proves that D_{train} is actually representative of D_{valid} and verifies the health of the data split. Eventually, the training accuracy is expected to stall while the validation accuracy is expected to drop. This denotes the point of overfitting where the network is starting to learn noisy, undesired features exclusive to the split D_{train}. Sometimes, the validation accuracy can stall as well instead of dropping, which could be an indicator that D_{train} and D_{valid} are excessively similar, specially if their top accuracies are close.

After these checks are made, training should be repeated for the right number of epochs so that the overfitting region is avoided. Once the model is properly trained, its performance on a set of new data D_{test} is to be computed and, from it, the quality of the model assessed.

The minimum accuracy for a model to be deemed adequate depends on the application, but overall we want our models to posses *statistical power*, meaning its results are better than a dumb baseline [4]. For instance, imagine a binary classification task for which our model has an accuracy of 65%. While this accuracy might not be good enough for applications related to human safety, the model still outperforms the 50% accuracy of making a choice at random.

Thus, a rule-of-thumb procedure to approach the training of a neural network could be:

- Fetch the required data D and split in D_{train}, D_{valid}, D_{test}.
- Define a network architecture (layers, optimizer, loss, hyperparameters...).
- Train the network on D_{train} measuring its performance on D_{valid} every epoch.
- Take a look at the training results and check that the optimization power is enough.
- Retrain the network, interrupting the training before overfitting on D_{train} takes place.
- Test the network's performance on D_{test} and assess the validity of the model.

Even following these steps, early stopping a neural network's training does not guarantee the complete dissapearance of overfitting, which will reflect on the performance gap between the training and validation accuracy. The term *regularization* comprises the set of techniques used to fight overfitting. Some examples of these are [4]:

- **Getting more training data**: A model trained on more data will naturally generalize better. However, it is not always possible to easily produce more training data.
- **Reducing the size of the network**: The smaller the number of learning parameters, the lesser the optimization power of the network, i.e. the "less likely a network is to just memorize the training data". Thus, smaller networks are less prone to overfitting.
- ***Weight regularization***: *L*N-*regularization* techniques add an additional cost proportional to the weights to the power of N, promoting a smaller weight domain within the network. For example, *L2-regularization* (also called *weight decay*), adds a factor proportional to the square of the value of the weight coefficients.
- ***Dropout***: which consists on randomly setting some of the weights within a layer to zero, during training. This way, individual weights are forced to be more representative themselves instead of "relying" on other weights in the same layer.

2.2.5 Deep learning for computer vision

So far, all examples have used densely-connected (or *dense*) ANN layers as models of choice. Dense layers are used to learn global patterns in their input feature space and, though they can be used for solving basic computer vision problems, they are not the optimal tool. A much more powerful layer type towards the field of machine vision are the *convolutional layers*. An artificial neural network built from convolutional layers is called *convolutional neural network* or CNN. CNNs are interesting to computer vision because of two main reasons [4]:

- **They learn translation-invariant patterns**: A learned pattern (*kernel*) can be found anywhere within a picture. Because the real world is fundamentally translation-invariant, this is a critical property towards efficient learning.
- **They learn spatial hierarchies of patterns**: In a CNN, early layers learn very simple patterns in pictures such as edges or simple corner shapes. Deeper layers can build richer combinations from these and learn increasingly complex visual concepts such as faces or animal shapes.

Digital images are represented by a three-dimensional array with axes $[height, width, channel]$. Conventional RGB images are composed of 3 channels, encoding the intensity of colors red, green and blue at every position $[height, width]$ within the picture. Figure 9 shows the three-channel decomposition of an example picture. Note that, for a conventional RGB image, each single channel can be seen as a single-channel (grayscale) image itself.

Convolutions operate over 3D tensors called *feature maps*, which conceptually are the same as images: the same way a common image is composed of three RGB channels, feature maps have a channel axis that contains different layers with information for the very same $[height, width]$ coordinate. Again, each channel in a feature map can be imagined as a a single-channel image (i.e., a grayscale picture) that encodes some useful information. If a conventional image is a graphical representation of the visual world, a feature map is the same concept only not so explicitly human-readable. In practice, the concepts of image, feature map and tensor (or array) are the same thing.

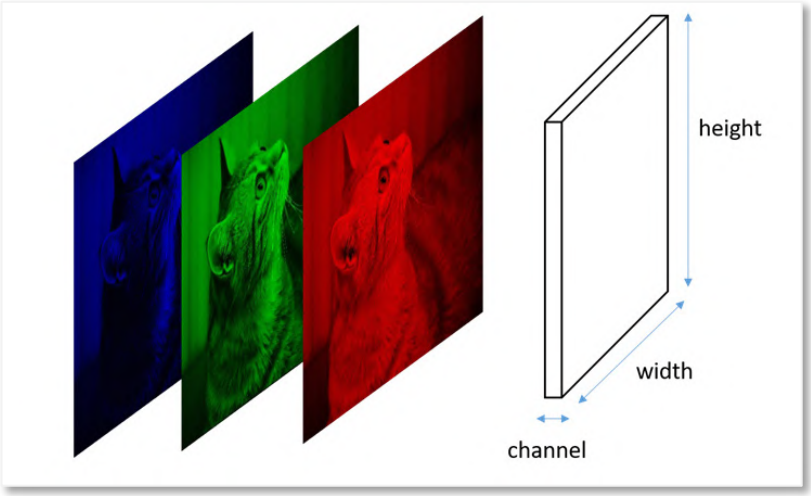


Figure 9: RGB channel decomposition of an image
Source: https://stats.stackexchange.com/questions/427680/

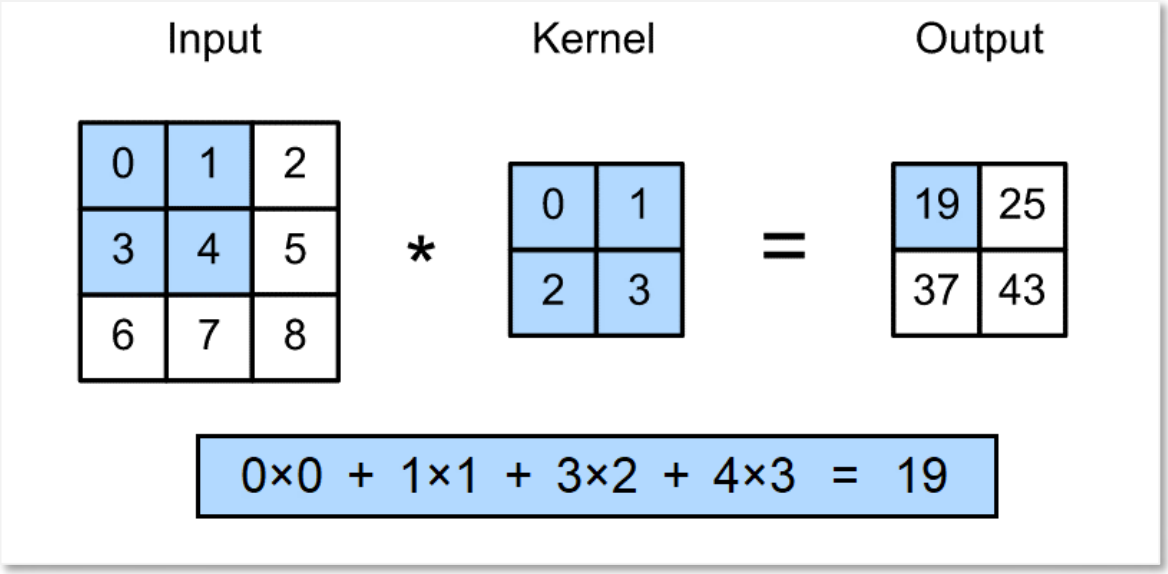


Figure 10: A simple numerical example of 2D convolution
Source: https://d2l.ai/chapter_convolutional-neural-networks/conv-layer.html

The convolutional layers of a CNN apply a convolution operation to their input to produce output feature maps. A 2D convolution operation over a single-channel image, by using a single-channel kernel, can be formally described as:

$$\boldsymbol{y}[i,j] = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} \boldsymbol{h}[m,n] * \boldsymbol{x}[i-m, j-n] \tag{4}$$

- $\boldsymbol{x}$: Being the input image undergoing the convolution operation.
- $\boldsymbol{y}$: Being the output image or feature map after said convolution.
- i,j : Denoting the coordinates of a pixel inside an arbitrary channel of x or y.
- $\boldsymbol{h}$: Convolution kernel, $h[0,0]$ denoting its center pixel.
- m,n : Helper variables to apply each point in the kernel h over the input image.

A simple example being shown in Figure 10. Put in words, 2D convolutions create an output feature map by fillling each output pixel with the *sum of the element-wise product between the kernel and a slice of the input image*. Convolutional layers admit inputs of arbitrary dimension, though the output size depends on that of the input.

In practice, convolution operations quantify the presence of a pattern, encoded in its kernel, all over the feature map. This concept is easily seen in Figure 11, which holds an example of a $3x3x1$ size kernel convolution, showing input picture and the output feature map, both single-channel matrices. It is common that convolutional layers apply 3D convolutions (i.e., kernel having various channels in the channel axis). The shape of the kernel defines whether an input feature map is to be broadcasted or shrinked in any of its axes $[height,\ width,\ channels]$.

The kernel size of a layer is defined when designing the network architecture, but its values are inferred, so that the output feature map becomes representative to the problem at hand through training. Thus, convolutional layers learn patterns of interest in a feature map.

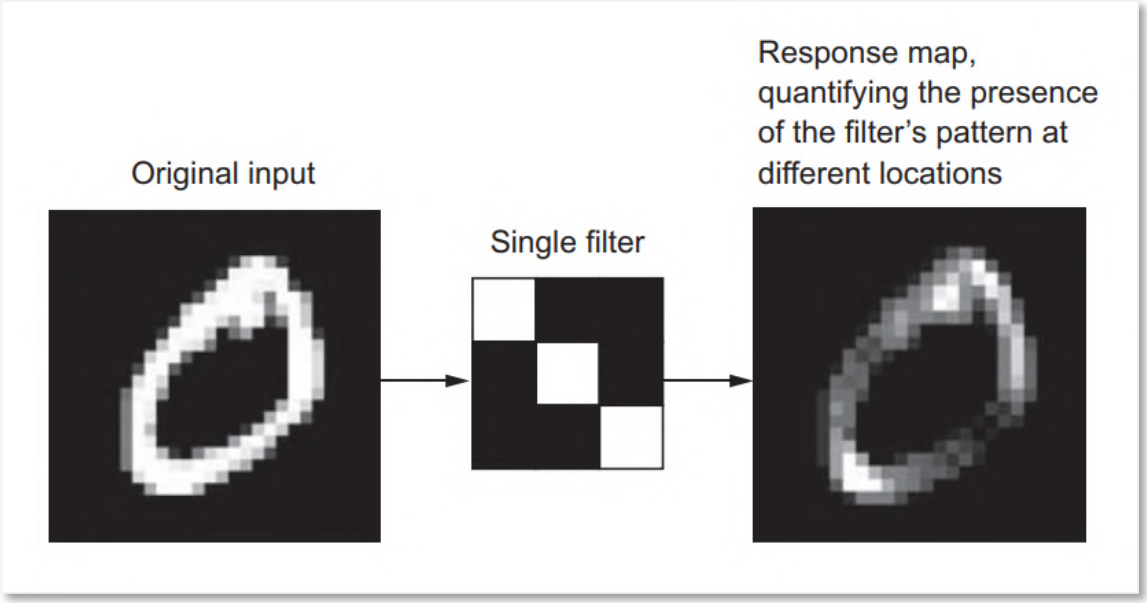


Figure 11: A simple visual example of 2D convolution
Source: [4]

Other relevant operation within CNNs is the *max-pooling* operation. Max-pooling layers are very similar to convolutional layers, only that instead of applying an element-wise product kernel, they keep the maximum value within a group of pixels [4]. When applying this operation, images are substantially downscaled.

Convolutional layers can also slightly reduce the size of an image. When applying strict convolution, a number of pixels on the borders of the input feature map are removed since there are no values to convolve there. This issue is usually addressed by *padding*: adding a frame of zeroes around an image so that the convolution operation will output a feature map with the same pixel size as the input. Convolution operations can also be used for image downsampling by adjusting the *stride*: the distance in pixels between two windows when sweeping an input feature map with a convolutional kernel. Default convolution is *unstrided*, or $stride = 1$, meaning that these windows are strictly contiguous. While padding and stride are concepts typically visualized in the $[height, width]$ axes, the same concepts can apply to the $[channels]$ dimension.

In any of these axes, the relationship between the number of features (pixels in $[height, width]$ or single-channel feature maps for $[channels]$) in the input and output of a convolution operation can be expressed as:

$$n_y = \left(\frac{n_x + 2p - k}{s}\right) + 1 \tag{5}$$

- n_y, n_x : Number of features in output and input feature map, respectively.
- p : Padding size (or number of zeroes added around the feature map).
- s : Stride size (or distance between two consecutive kernel windows).
- k : Kernel size, measured in the same direction as the feature of interest.

2.2.6 Data augmentation

Overfitting is caused by having few examples to train a model on, so the model has trouble generalizing to new data. A common method to greatly reduce overfitting on image data is to artificially enlarge the data using random label-preserving transformations that yield believable-looking images [4]. This technique is known as *data augmentation.* Data augmentation is generally performed in-line when training a network, meaning that transformed images are not stored in disk. This way, a neural network never trains on two equal imaegs, its generalization power greatly increasing. Some image transformations comonly used for data augmentation are:

- Mirroring : Either horizontally or vertically.
- Rotation : Clockwise or counterclockwise.
- Blurring : Applying either *gaussian noise* or directional blurring effects.
- Brigthness : Altering brightness and contrast to account for lighting variations.
- Hue : To learn from items of different item colours.
- Warping : Geometrical transformations to account for different shapes of items.

Note that not all of the previous transformations will always yield a positive effect on training: mirroring might not be suitable for trying to detect asymmetrical items and hue can be pointless if working with grayscale images. Figure 12 shows an example of some augmentations.

Figure 12: Examples of image augmentation by warping and mirroring
Source: [4]

2.2.7 Transfer learning. Feature extraction and fine-tuning

Transfer learning is the practice of using a neural network that has been pretrained on a large amount of data to solve a task T_1, replace one or multiple layers of it, and then retrain it on a different set of data to solve task T_2 [5]. Towards training CNNs, this approach is very powerful because representations learned in early layers tend to be very low-level and universal: these kernels typically hold information on edge detection, color detection and lighting changes, and usually require gigantic amounts of data and computing power for a proper tuning.

In transfer learning, a pretrained model M_1, typically trained on a rich dataset, is used to initialize the weights of a second model M_2, that is then trained on an usually smaller, more limited data. The transferred layers generally have the same architecture, though the rest of the network might be different, though for the sake of simplicity let's assume that the architectures of M_1 and M_2 are equal. There exist two main methodologies to apply transfer learning [10]:

- *Feature extraction:*
  - M_2 is initialized with the weights of M_1, which is pretrained on a dataset D_1.
  - Early layers of M_2 are frozen (*frozen parameters* are constrained as constant).
  - M_2 is trained on a set of data D_2, so only later layers are learning.
- *Fine-tuning:*
  - M_2 is initialized with the weights of M_1, which is pretrained on a dataset D_1.
  - M_2 is trained on a set of data D_2, so all layers are learning.

The difference between the two being the fact that some layers are frozen. In general, it is always favorable to use transfer learning for initializing neural networks, especially complex one. Unless for some reason this is not possible, transfer learning should always be performed if a suitable pretrained model is available, as it will always save training time.

- When to use feature extraction?
  - When the problem M_1 and M_2 solve is similar and D_2 has some traits that can lead to convolutional layers overfitting (small data, class imbalance... etc.).
  - When we want a faster training.
- When to use fine-tuning?
  - When the problem M_1 and M_2 solve is reasonably different.
  - When a high accuracy is desired no matter how slow the training.

As an additional note, the name *feature extraction* is also used to define the action of processing a feature map to extract features of interest via CNNs [11].

2.2.8 Alexnet. Image classification

Alexnet [9] is a CNN architecture designed for a 1000-class image classification task. At the time of publication (2012), Alexnet achieved a new state-of-the-art on image classification by the use of a very deep network architecture, the training of which required innovative solutions such as the use of non-saturating neurons to minimize vanishing gradient and a double-GPU implementation. The main result of this work was that very deep CNN architectures are key towards improving feature extraction from images [9].

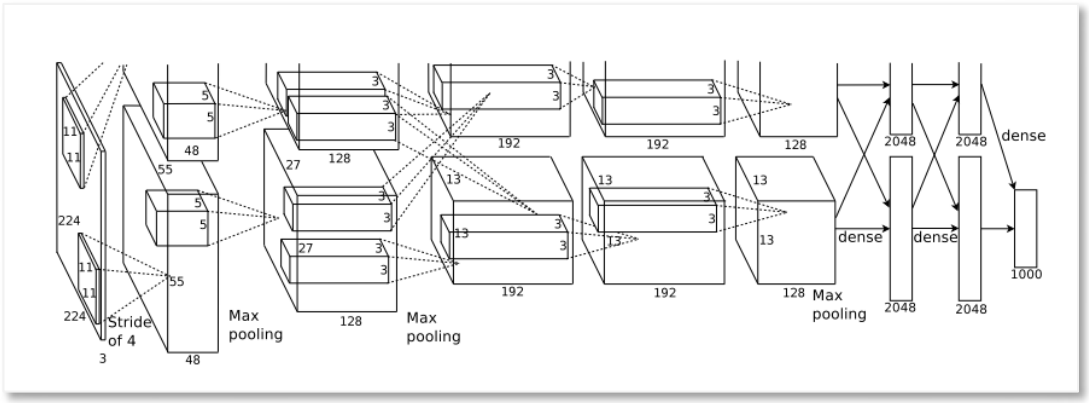


Figure 13: Architecture of the image classifier Alexnet
Source: [9]

The architecture of Alexnet is shown in Figure 13. From start to finish, we can see two parallel branches, which represent the computations happening in two separate GPUs. This architecture defines a very basic structure for the problem of image classification. A generic image classification network is composed of two blocks:

- **A convolutional set of layers**: Which acts as a feature extractor by processing an input image and obtaining relevant features from it.
- **A densely-connected set of layers**: Which acts as a classifier, assigning a class to the feature vector extracted in the convolutional stage via a stack of dense layers with a 1000-way softmax last-layer activation.

The parallelization power of GPUs makes them very suitable for optimizing efficient implementations of the convolution operation, higly overperforming CPUs [4], also for neural network fields outside of computer vision. By the time Alexnet was published, GPUs were not as powerful as they are now, hence the reason the original paper proposed a double-GPU setup for faster training. Training and inference on GPU are estimated to be in the order of 10 times faster than on CPU. Newer, AI-dedicated computing units known as TPUs also exist, though it is more common for a regular consumer to have access to a GPU on any high-end laptop.

2.3 Object detection

2.3.1 Introduction

Object detection is a complex and advanced task in image processing and computer vision that seeks to identify the location of specific items in images by defining a *bounding box* (*bbox*) around them and assign the corresponding labels to the objects in terms of the object category, i.e., human, tree, or car. An example of a detected image is shown in Figure 14. Some applications of object detection lay in the fields of automated visual inspection, robot vision, autonoumous driving, video surveillance, etc [12]. Video-based object detection pipelines also exist, though they are overall less popular than their image counterpart, as they have only emerged more recently [13]. Compared to a frame-by-frame inference of a video with image detectors, video-based detectors are more efficient by taking temporal correlation into account and more robust to phenomena like motion blur, defocus, occlussion, etc.

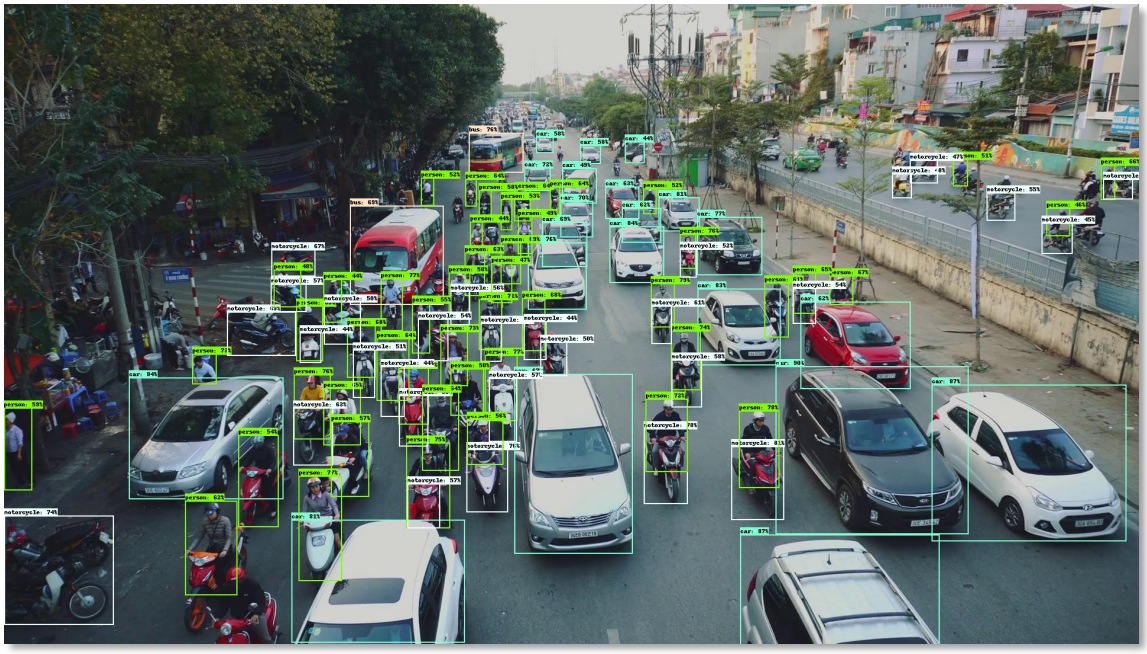


Figure 14: Showcase image of object detection with EfficientDet
Source: https://github.com/google/automl/tree/master/efficientdet

Conventional object detection pipelines are generally based on convolutional neural networks (CNNs), which have proven useful to extract deeper representations of images by applying convolutional operations to images to obtain more relevant feature maps [9]. Thus, both big amounts of computational power and training data are expected in order to build reliable object detection models.

Through the years, the research community has been using and growing benchmarking datasets that are publicly available, which has led to very big amounts of data being available online. These datasets tend to be very big and published models are usually pretrained on them, providing solid starting points for custom applications, where a lot of data is not usually available, by means of transfer learning. Some of the most popular object detection datasets are MS COCO [14] and PASCAL VOC [15].

Object detection methods can be divided in two categories depending on their structure [12]:

- ***Two-stage frameworks***: These models split detection of items into two stages, the first of which consisting on a *region proposal* algorithm (that may use machine learning or not), and a second one for the actual classification of those regions. Models falling in this category are R-CNN, Fast R-CNN, Faster R-CNN and RFCN, among others. These tend to be more accurate, though slower.
- ***Single-stage frameworks***: Unlike the former, these models do not use a region proposal component, but regress bounding box coordinates as well as classes in one shot. Some examples in this category are YOLO, SSD and RetinaNet, among others. These tend to be faster, though less accurate

This work will mainly focus on image-based, single-stage object detection frameworks, the most relevants of which are to be described in the upcoming sections.

2.3.2 Performance measures

While image classification tasks can be rated by accuracy, a relevant performance measure for object detection is harder to define. In an object detection task, the network must not only regress the bounding box coordinates of an item, but also assign a category to it, the quality of both having to be accounted for.

The most common metric for evaluating detectors is the *average precision* or AP. Let's first introduce the following concepts from an object detection point of view:

- ***Intersection over Union* (*IoU*)**: also known as *Jaccard Index*, is an scalar value of intersection for two bounding boxes. The closer to one, the most similar the bounding box coordinates are. Graphical representation is shown in Figure 15.
- ***False positives* (*FP*)**: A false positive is called when a detection is either made for a ground-truth bounding box that had already received one (detection duplicates) or when a detection does not meet a lower-bound IoU.
- ***True positives* (*TP*)**: Which are those predictions made for the correct class, without duplicates and meeting a high-enough IoU with a ground-truth annotation.
- ***False negatives* (*FN*)**: Called when an item is either predicted with the wrong class or on background scenery.
- ***True negatives* (*TN*)**: Which are instances of background that are ignored, thus not classified as objects.

The AP is computed by inferring object detection on a batch of test data as the area under the *precision-recall curve* [5]. From the previous definitions let's point the fact that it is needed to establish a minimum IoU before deciding what category does a prediction fall in. This makes the precision-recall curve, as well as the AP, dependent on a threshold IoU, namely IoU_{thresh}. A precision-recall curve can be drawn by performing object detection over a set of data D and tracking the points of *precision* and *recall*. In vector notation:

$$\boldsymbol{Precision}_i(IoU_{thresh}) = \frac{TP(i, IoU_{thresh})}{TP(i, IoU_{thresh}) + FP(i, IoU_{thresh})}$$

$$\boldsymbol{Recall}_i(IoU_{thresh}) = \frac{TP(i, IoU_{thresh})}{TP(i, IoU_{thresh}) + FN(i, IoU_{thresh})} \tag{6}$$

$$\boldsymbol{PrecisionRecallCurve}_i(IoU_{thresh}) = \begin{bmatrix} \boldsymbol{Recall}_i(IoU_{thresh}) \\ \boldsymbol{Precision}_i(IoU_{thresh}) \end{bmatrix}$$

- IoU_{thresh} : Being the lower-bound IoU for a prediction to be considered accurate.
- i : Denoting the inference progress over D and serving as vector index.
- $TP(i,)*$: Denoting the total number of TP when D has been inferred up to D_i.
- $\boldsymbol{Recall_i}**$: Denoting the i-th element in the recall vector.

$*$ Notation expands to FP & FN.

$**$ Notation expands to $\boldsymbol{Precision}$ & $\boldsymbol{PrecisionRecallCurve}$.

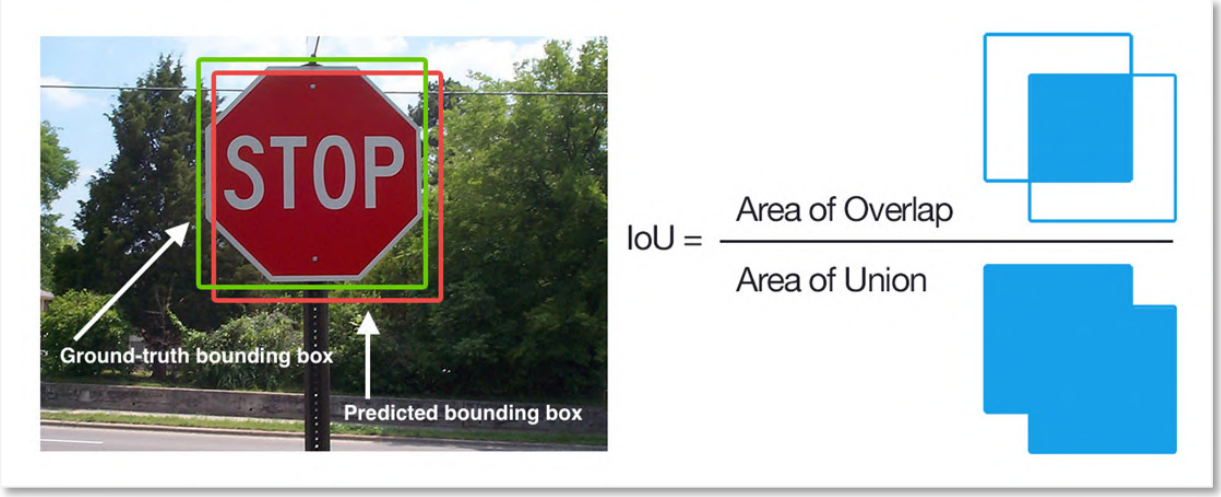


Figure 15: Jaccard Index or intersection over union
Source: https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/

The AP is computed by interpolating the points in the precision-recall curve then computing the area under it. The higher the AP is, the better the model, where 1 is the optimal value. A more general way of expressing this metric is the *mean average precision* or *mAP*, which is computed as the average of many AP values within two different values of IoU.

Other common AP metrics are:

- $AP@0.50$ or $AP@50$: Computed for $IoU_{thresh} = 0.50$.
- $AP@0.75$ or $AP@75$: Computed for $IoU_{thresh} = 0.75$.
- $AP@0.50:0.05:0.95$: Averaging $IoU_{thresh} = 0.50:0.95$ in steps of 0.05.

In practice, the definition of *mAP* is not strictly closed: the bounds and steps for averaging the AP, as well as the rules to interpolate the precision-recall curves, may vary from author to author. However, it is safe to adopt a canonical definition of mAP as the average AP shown above, which corresponds to the definition of the MS COCO dataset official metrics.

2.3.3 MS COCO metrics

The MS COCO [14] dataset is one of the most famous benchmarking baseline datasets for computer vision tasks, with over 200K annotated pictures. Each year, different challenges are proposed for participants to develop innovative computer vision pipelines to solve these tasks. Thus, the dataset is continuously evolving to increase the amount of available data for both existing and new challenging problems.

The MS COCO object detection challenge is, again, among the most common baseline on which object detection pipelines have been compared throught the years. In this challenge, challenger detectors must detect items from 80 different categories. For ranking MS COCO model results, a standard set of metrics has been established, as shown in Figure 16. These metrics provide a general overview on the capabilities of object detection models and are generally used for overviewing results not only on MS COCO, but also on custom datasets for applicatons other than the challenge itself.

```
Average Precision (AP):
   AP                  % AP at IoU=.50:.05:.95 (primary challenge metric)
   AP^IoU=.50          % AP at IoU=.50 (PASCAL VOC metric)
   AP^IoU=.75          % AP at IoU=.75 (strict metric)
AP Across Scales:
   AP^small            % AP for small objects: area < 32^2
   AP^medium           % AP for medium objects: 32^2 < area < 96^2
   AP^large            % AP for large objects: area > 96^2
Average Recall (AR):
   AR^max=1            % AR given 1 detection per image
   AR^max=10           % AR given 10 detections per image
   AR^max=100          % AR given 100 detections per image
AR Across Scales:
   AR^small            % AR for small objects: area < 32^2
   AR^medium           % AR for medium objects: 32^2 < area < 96^2
   AR^large            % AR for large objects: area > 96^2
```

Figure 16: MS COCO challenge official metrics
Source: [16]

The metric named AP in Figure 16 is the same metric that was defined as mAP in Section 2.3.2. The MS COCO team states [16] that no difference among the two terms is to be made and leaves its meaning to context. MS COCO metrics include the concept of AP per size, which checks the precision for different item size intervals, measured in image pixels. Same is applied for the *average recall* or AR metric, which is the maximum recall forcing a fixed maximum number of detections per image, averaged over categories and IoUs.

Among all these metrics, the most relevant for measuring the overall performance of a model is the mAP [16], while others provide only additional details that are useful towards diagnosing and comparing more specific qualities. Regarding notation, this work will adopt mAP as the symbolic name for the mean average precision, under the MS COCO definition, so that:

$$mAP := AP@0.50{:}\,0.05{:}\,0.95 \tag{7}$$

There have been records of some discrepancies on how these metrics may be computed, depending on the platform used to do so [17]. These origin, among others, on the fact that some authors like to flatten the precision-recall curves so that they are monotonously decreasing [18]. The MS COCO team has published a library for computing their standardised metrics, which can be found in the COCO API repository at [19].

This work will adopt these canonical metrics, as defined by the MS COCO team. Implementation-wise, these will be computed by using the tools at [19], in order to obtain relevant and consistent numerical results that will allow the comparison of the different models of interest.

2.4 Relevant object detection pipelines

2.4.1 R-CNN, Fast R-CNN and Faster R-CNN (2014, 2015, 2015)

The *R-CNN* [11] architecture splits the detection of items into three stages: region proposal, feature extraction and bounding box classification. Because region proposal comprises its own stage, R-CNN falls among two-stage models. This pioneer work in object detection shares strong links with the principle behind Alexnet: using CNNs for feature extraction and then conventional densely-connected layers for classification. Similarly, R-CNN stages have the following goals:

- Region proposal : Aims to find image regions that might contain items.
- Feature extraction : Use a CNN to get feature maps from the proposed regions.
- Classification : Classify each region feature map as objects or background.

Aligned with the definition of object detection stated in Section 2.3.1, in R-CNN a series of bounding box are initially computed, then they are classified as if they were individual images. Two major improvements were made over the original R-CNN: *Fast R-CNN* [20] and *Faster R-CNN* [21], an overview of each shown in Figure 17. The main changes these implement being:

- Fast R-CNN : Improve efficiency by swapping the order of the CNN part and the region proposal, so that feature extraction is only performed once.
- Faster R-CNN : Substitute the region proposal algorithm by a neural network.

R-CNN [11]: In the original implementation, region proposal is performed by a non-data-driven *selective search* [22] algorithm to find possible bounding box locations, thought the pipeline is agnostic to the region proposal method itself. The output of this algorithm is a series of image bounding boxes of arbitrary sizes, named *regions of interest* or RoI. To maximize the recall (i.e. not missing items in the picture), lots of overlapping region proposals will be made. The obtained RoIs are then warped to fit the input size of an Alexnet backbone (i.e., the convolutional part of a pretrained Alexnet is imported via transfer learning). For each RoI, a feature vector is extracted and then classified by a densely-connected implementation of *support vector machines (SVM)* or binary classifiers, each returning a confidence index $\in (0, 1)$ of the RoI containing an item of a determinate class. Since it is expected that RoIs overlap, *greedy non-maximum suppresion* is performed so that for a set of overlapping RoIs, only the one with highest index is kept. A later review of R-CNN implemented an additional *bounding-box regression* stage, which improves localization performance by modifying the bounding box coordinates depending on the class that has been detected.

Fast R-CNN [20]: Takes a picture and a set of object proposals as input (region proposal stage considered outside of the pipeline). The picture is fed to a CNN that will compute a common feature map for the whole image. Then, each RoI takes a slice of this common feature map through a RoI pooling layer, the output of which is fed to a classifier network (now implemented as a n-way softmax) and a bounding box regressor. This architecture can be trained end-to-end, rather than requiring the individual progressive training of the backbone + each SVM + bounding box regressor. Because the convolutional map is shared, the CNN is only run once, allowing a massive boost on computational efficiency and inference speed.

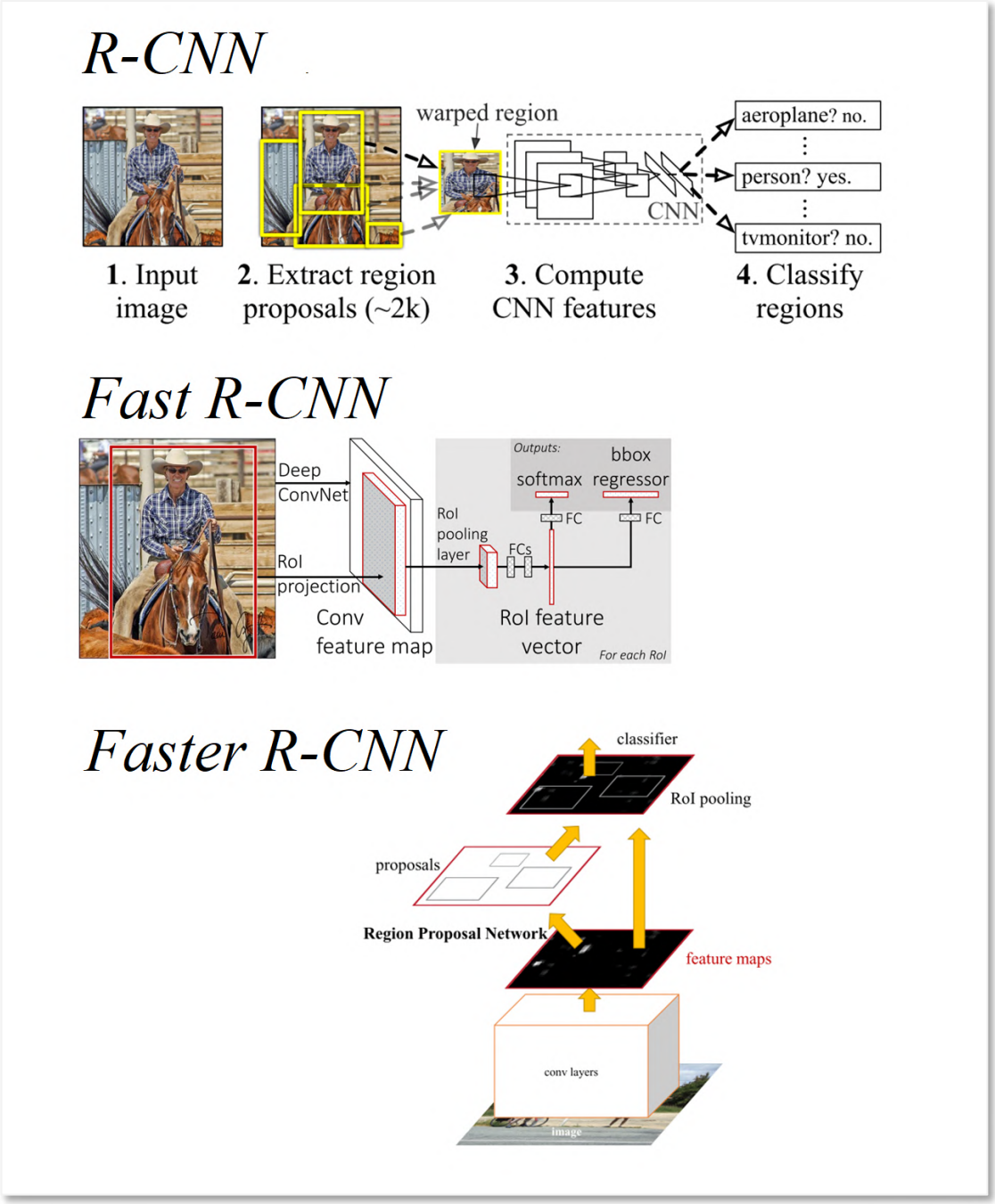


Figure 17: Overview of the object detectors R-CNN, Fast R-CNN and Faster R-CNN
Source: [11], [20], [21]

Faster R-CNN [21]: After Fast R-CNN improved the efficiency of the feature extractor, the main bottleneck of the R-CNN approach was the region proposal. Faster R-CNN substitutes traditional methods with a *region proposal network* or *RPN*, a fully convolutional network that shares convolutional features with the main detection network, allowing for nearly cost-free region proposals. This *RPN* simultaneously predicts object bounds and objectness scores at each position of the image, improving speed, RoI quality and overall detection accuracy. *RPNs* work by applying a number of sliding windows over the input image at the same time, the size and aspect ratios of said windows being predefined as a set of *anchor boxes*.

2.4.2 YOLO (2015)

The *YOLO* (*You Only Look Once*) [23] family (see also [24], [25]) of object detection pipelines is a set of single-stage models initially developed with the aim of maximizing detection inference speed. The original YOLO [23] was published when the baseline object detection model was still the two-stage approach R-CNN. YOLO aimed to implement the whole detection pipeline in a single convolutional neural network, in an efficient manner so that object detection could approach real time. YOLO's official implementation is distributed on *Darknet* [26], an open-source neural network library written in C.

YOLO proposes the use of features common to the whole image to predict bounding boxes, so that the detections are context-aware. The input image is divided into a grid of cells, each one of these predicting a fixed number of bounding boxes and a confidence score of the fact that an item is present in the bounding box. Each bouding box consists of 5 predicitions of width, height, horizontal position, vertical position and confidence score. Furthermore, each cell predicts the conditional class probability provided there is an object in the picture $P(class_i|object)$. Each cell grid proposes thus a series of bounding boxes with objectness score and probabilities of a class provided there is an object within [23]. An overview of this is displayed in Figure 18.

The main limitations of this model [23] are that each cell in the grid can only predict a single class and that it does not generalize well to new objects with different proportions (though it does generalize properly to different domains thanks to being context-aware). Also, the loss function penalizes localization errors in large and small bounding boxes equally, while it is reasonable that small errors in the latter have less effect towards missdetections. Suboptimal localization is the main source of error of this model.

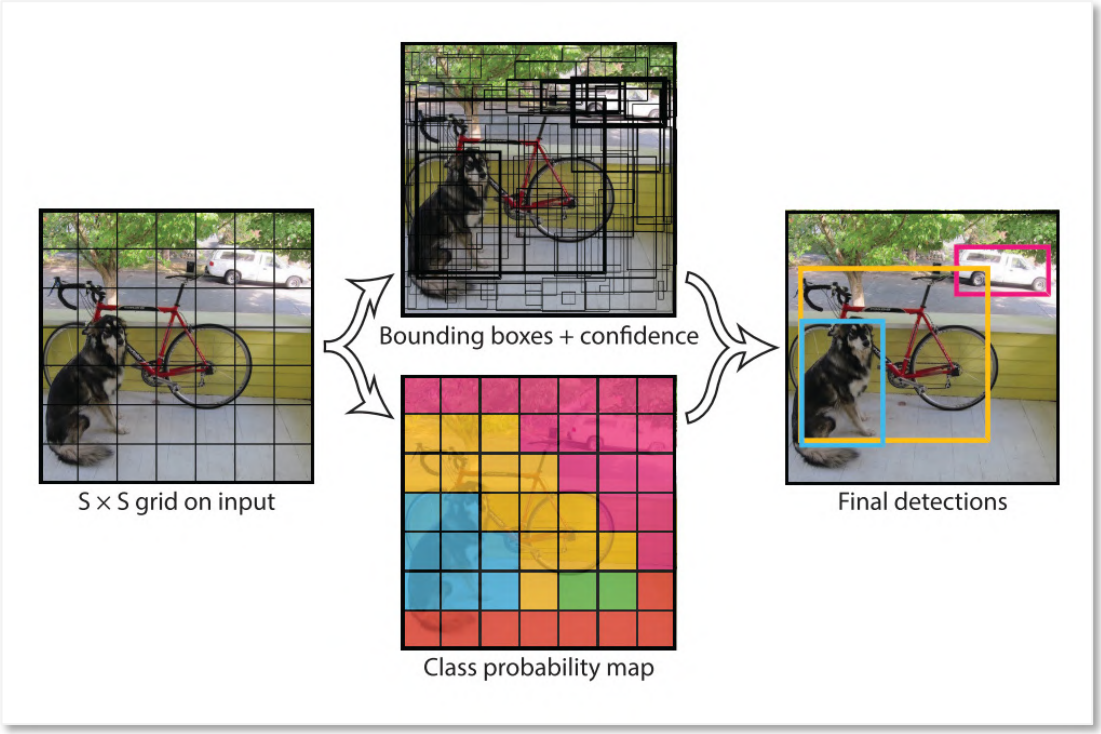


Figure 18: Overview of the object detector YOLO
Source: [23]

2.4.3 SSD (2015)

The single-stage *single shot detector* or SSD [27] is the first deep network based object detector that does not resample pixels or features for bounding box hypotheses and and is as accurate as approaches that do. The architecture of SSD uses a single convolutional network to produce a collection of a fixed number of bounding boxes with scores for the presence of instances of objects in them, then applies a non-maximum suppresion to denoise the final resuts.

The earlier part of SSD network is inherited from the convolutional part of a *VGG-16* network, and its sole purpose is to obtain a representative feature map of the input image. The additional structure afterwards takes care of predicting detections. SSD has the following key features:

- **Multi-scale feature maps**: A set of size-decreasing convolutional layers or *feature layers* with different architectures is used to allow predictions of items at different scales.
- **Convolutional predictors for detections**: Each additional feature layer can produce a fixed set of detections via their convolutional filters.
- **Default boxes and aspect ratios**: A set of default bounding boxes is initially associated with each cell (divisions of the input image), a similar concept to the *anchor boxes* defined in [21]. In each of these, offsets relative to these default bounding boxes are predicted, as well as the per-class score that indiated the presence of a class instance in each box.

Figure 19 shows the SSD architecture, as well as an overview on the image processing taking place within. During training, a small set of default boxes with different aspect ratios is evaluated at each location in the feature map (which is divided in a series of cells, the number of which depends on the scale the feature layer, as shown in Figure 19), in each of the feature layers taking care of different scales. The bounding boxes that best fit with the ground-truth are kept; the rest discarded as negatives. The model loss function is a weighted sum of the localization the confidence loss. Among the key features of the training stage of [27], these two are highlighted:

- **Hard negative mining**: With this approach, most of the default boxes are discarded as negatives, introducing a significant class imbalance. Instead of using all the negative examples for network update, only the top examples confidence loss for each default box are kept (i.e., the ones that the network is surer do not contain an item), leading to faster optimization and more stable training.
- **Data augmentation forcing occlussion**: Among other data augmentation techniques, random patch sampling so that items are partially outside the picture is performed. This helps the model generalize better when facing only portions of objects at inference time.

Compared to methods that do require object proposals, SSD is simpler because it doesn't need to perform neither object proposal nor pixel or feature resampling, holding all computation in a single *feedforward* network. SSD can be trained end-to-end and, compared to other models at the time, was easy to integrate in more complex applications [27].

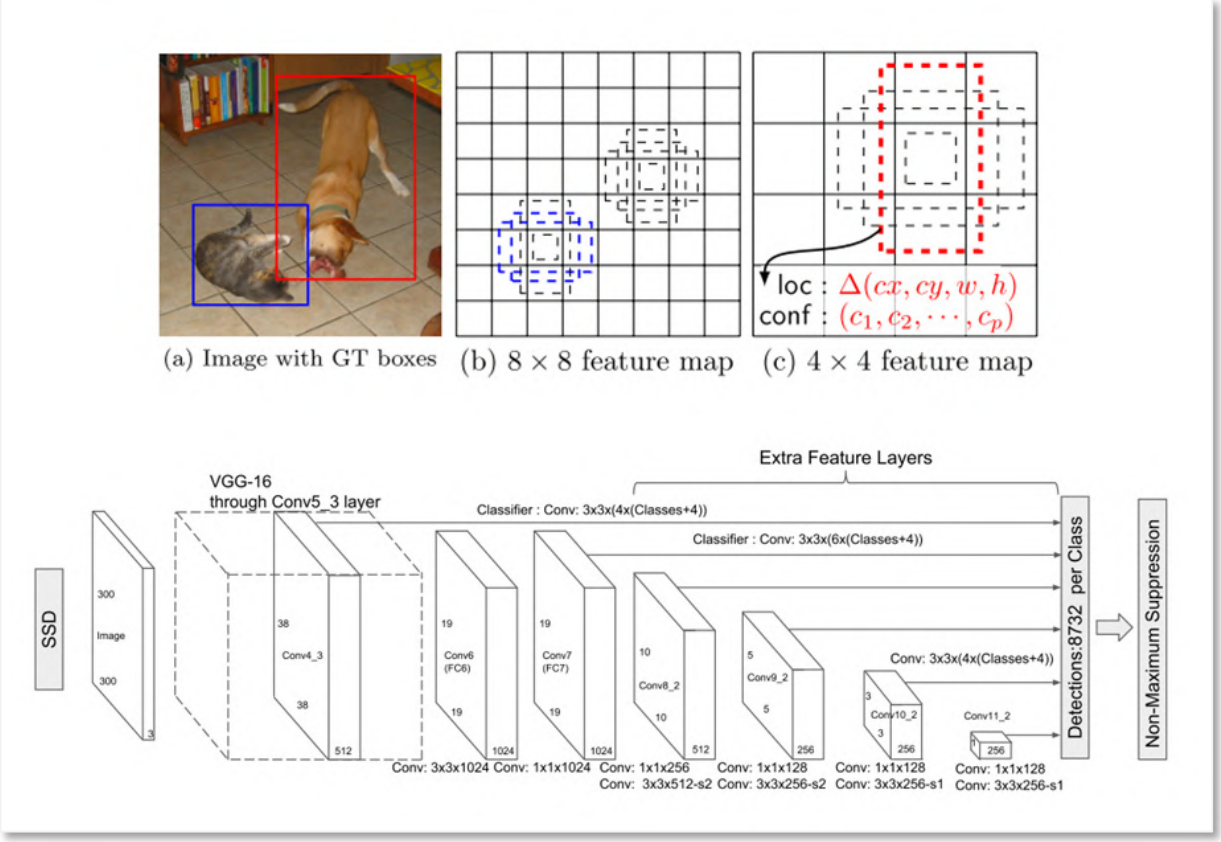


Figure 19: Overview and architecture of the object detector SSD
Source: [27]

2.4.4 EfficientDet (2019)

EfficientDet [28] is a current state-of-the-art single-stage object detector published by the Google Brain Team. EfficientDet was designed as an scalable model, meaning its architecture can be modified to fit a particular input image resolution to adjust for a speed-accuracy trade-off, aiming for an innovative compound scaling, a concept introduced in the convolutional model *EfficientNet* [29], also compound-scalable and proposed by Google. EfficientDet combines an EfficientNet backbone for feature extraction with *BiFPN* layers for *feature aggregation*. Then, two separate convolutional subnetworks predict bounding boxes and their labels. The whole EfficientDet architecture is shown in Figure 20.

The purpose of the convolutional model EfficientNet [29] was to explore more efficient ways of scaling convolutional networks. So far, it was common to scale only one of the network dimensions: depth, width or resolution. Scaling two dimensions arbitrarily required tedious manual tuning that would not reach an optimal working point. The basic architecture of EfficientNet is found via *network architecture search* or *NAS* [30], a data-driven approach to infer a network architectures, by maximizing the image classification accuracy and minimizing the number of *floating operations per second* or *FLOPS* (computational power). The obtained model was born as *EfficientNet B0,* later scaled according to the criteria defined at [29], to obtain a series of compound-scaled convolutional architectures B0-B7.

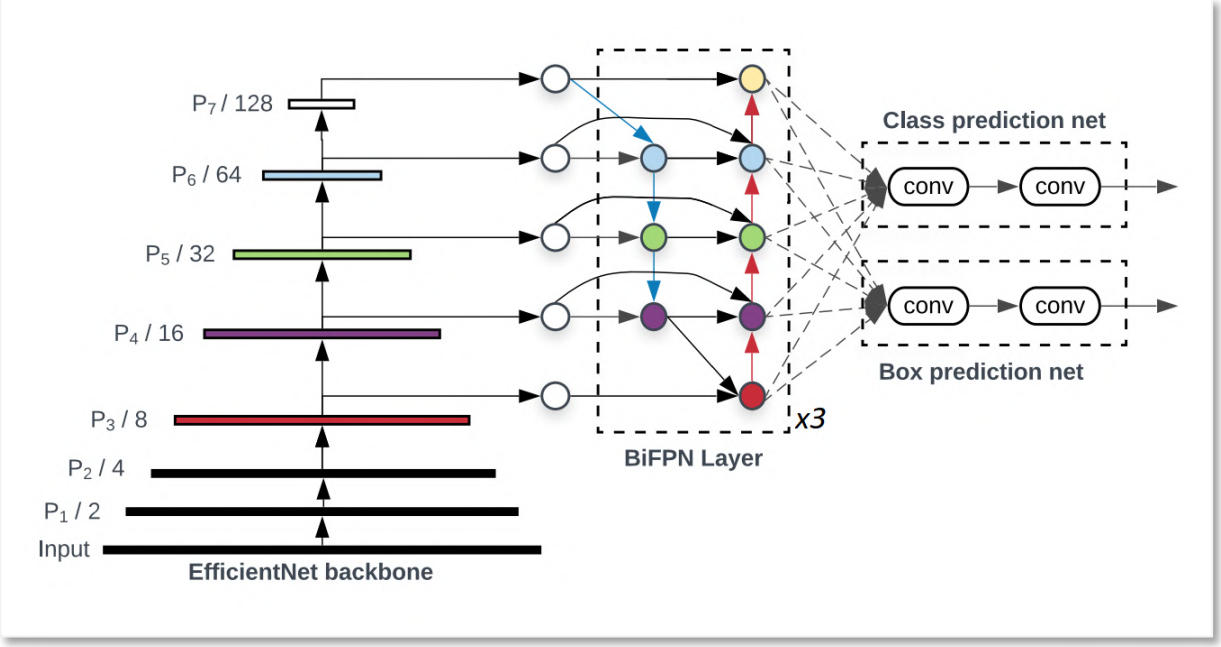


Figure 20: Architecture of the detector EfficientDet
Source: [28]

BiFPN layers are repeatable blocks that aim to combine features that have been detected at different image scales. The intent to combine features at different scales receives various names: *feature integration*, *feature aggregation* or *feature fusion.* As defined in [28], given a list of multi-scale features $\boldsymbol{P_{in}} = (P_{l1}^{in}, P_{l2}^{in}, \dots)$, where P_{li}^{in} represents the feature at level l_i, the problem is to find a transformation f that can aggregate different features and output a representative list of new features $P_{out} = f(\boldsymbol{Pin})$. To solve such task, the following solutions have been traditionally used:

- ***Feature pyramid networks* or *FPN*** [31]: Which adds a top-down pyramid architecture with shortcuts onto a size-decreasing convolutional backbone.
- ***Path aggregation network* or *PANet*** [32]: Which adds an extra bottom-up path to the FPN concept to overcome its one-way information flow.
- ***Network architecture search FPN* or *NAS-FPN*** [30]: Which proposes a complex model for feature aggregation layer discovered via an heuristic data-driven method.

From which, by applying and intuitions exposed at [28], the *BiFPN* layer model is derived as shown in Figure 20. Because of the upsampling/downsampling steps within the pyramids, the features in the aggregated layer and the receiving one should not contribute equally to the output feature, so their addition is weighted by learnable parameters.

The scaling of EfficientDet is performed in a compound manner, meaning that all of its dimensions (depth, width, input resolution) are modified in each of the proposed configurations *EfficientDet D0-D7*. While the backbone EfficientNet is scalable in itself as shown in [29], the rest of the detector follows a series of scaling rules stated in [28]. Towards this work, it is more convenient to show the features of each particular model scale, and thus they are displayed in Figure 21.

	Input size R_{input}	Backbone Network	BiFPN #channels W_{bifpn}	BiFPN #layers D_{bifpn}	Box/class #layers D_{class}
D0 ($\phi = 0$)	512	B0	64	3	3
D1 ($\phi = 1$)	640	B1	88	4	3
D2 ($\phi = 2$)	768	B2	112	5	3
D3 ($\phi = 3$)	896	B3	160	6	4
D4 ($\phi = 4$)	1024	B4	224	7	4
D5 ($\phi = 5$)	1280	B5	288	7	4
D6 ($\phi = 6$)	1280	B6	384	8	5
D7 ($\phi = 7$)	1536	B6	384	8	5
D7x	1536	B7	384	8	5

Figure 21: Scaling configurations for the detector family EfficientDet D0-D7
Source: [28]

2.4.5 YOLOv4 (2020)

YOLOv4 [33] is an upgrade made over the YOLO concept by Alexey Bochkovskiy (*AlexeyAB)*, whose extensive work has included forking an improving the original *Darknet* published by Joseph Redmon (*pjreddie)* when the original YOLO was released. Joseph Redmon was also responsible for the upgrades on the original YOLO: YOLOv2 [24], YOLO9000 [24] and YOLOv3 [25]. The official implementation of YOLOv4, as well as the upgraded backwards-compatible version of Darknet, is available at [34]. The research paper in which YOLOv4 is introduced [33] performs a very deep review on techniques that have been published so far and are believed to improve object detection performance, later performing ablation studies to define the optimal architecture for YOLOv4, which allows for high-quality detections and training on consumer-grade GPUs.

In [33], an essential structure for a generic state-of-the-art object detector is proposed as a sequence of the following items, a graphical representation of which is shown in Figure 22:

- ***Input***: Typically an image, but can also be patches of a picture... etc.
- ***Backbone***: Convolutional model meant to extract a set of features.
- ***Neck***: Performs additional tasks complementary to feature extraction, for instance feature aggregation.
- ***Head***: Responsible for actually predicting bounding boxes and classes.
    - **Dense prediction**: Makes the prediction for bounding boxes. If the model is single-stage, also predicts the class.
    - **Sparse prediction**: Only in two-stage models, classifies the bounding boxes proposed at the dense predictions step.

Furthermore, the following concepts are also defined in [33]:

- ***Bag of freebies* (*BoF*)**: Set of methods that only change the training strategy and tend to improve performance only increasing the training cost.
- ***Bag of specials* (*BoS*)**: Set of methods that slightly increase the inference cost but significantly improve the object detection accuracy.

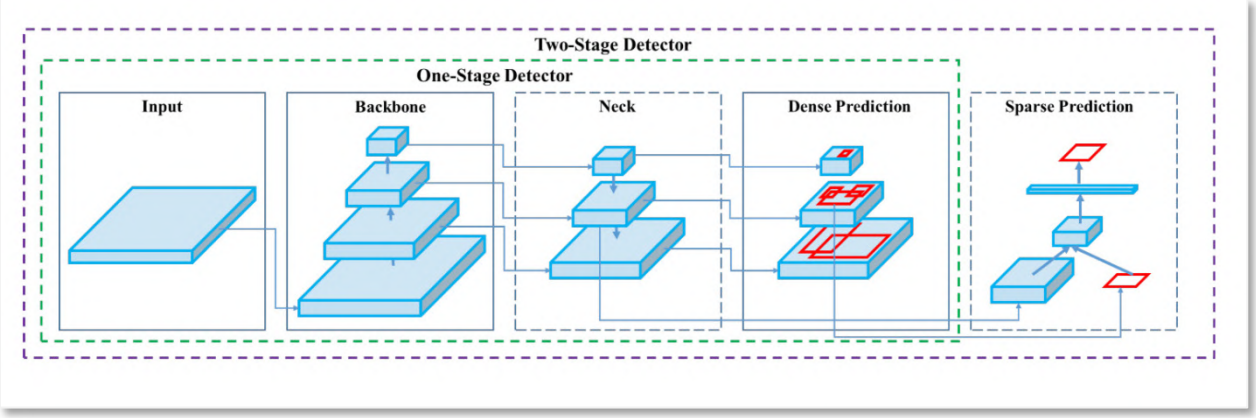


Figure 22: Generic object detection structure stated for the definition YOLOv4
Source: [33]

The final architecture of YOLOv4, as stated, is defined from a series of ablation studies after implementing many different solutions that have been proposed through the years. The finally proposed architecture of YOLOv4 answers to the model depicted on Figure 22.

The YOLOv4 architecture is composed of [33]:

- Backbone : CSPDarknet53
- Neck : *SPP, PAN*
- Head : YOLOv3

The choice of backbone CSPDarknet53 answers to the hypotheses that a high *receptive field* and larger number of parameters will define the best backbone. The receptive field [35] of a convolutional unit is the portion of the input that will be used to produce an output (as opposed to dense layers, convolutional layers do not process its input state entirely). The suitability of CSPDarknet53 is tested experimentally, also proving to be the fastest among the other candidate models. *Spatial pyramid pooling* (*SPP*) is incorporated into the neck to further boost the receptive field, along with a pyramid aggregation network PAN for feature aggregation. Finally, the anchor-based model YOLOv3 is selected as the head, defining YOLOv4 as a one-stage detector.

Though the reader is directed to [33] for further details, the additional methods of which YOLOv4 benefits are mentioned below:

- **Backbone BoS**: Cutmix and Mosaic (data augmentation), DropBlock regularization, Class label smoothing.
- **Backbone BoF**: Mish activation, Cross-stage partial connections, multi-input weighted residual connections.
- **Detector BoS**: CIoU-loss, CmBN, DropBlock, Mosaic, Self-Adversarial Training, eliminate grid sensitivity, using multiple anchors for a single ground truth, cosine annealing scheduler, optimal hyperparameters, random training shapes.
- **Detector BoF**: Mish activation, SPP-block, SAM-block, PAN path-aggregation block, DIoU-NMS.

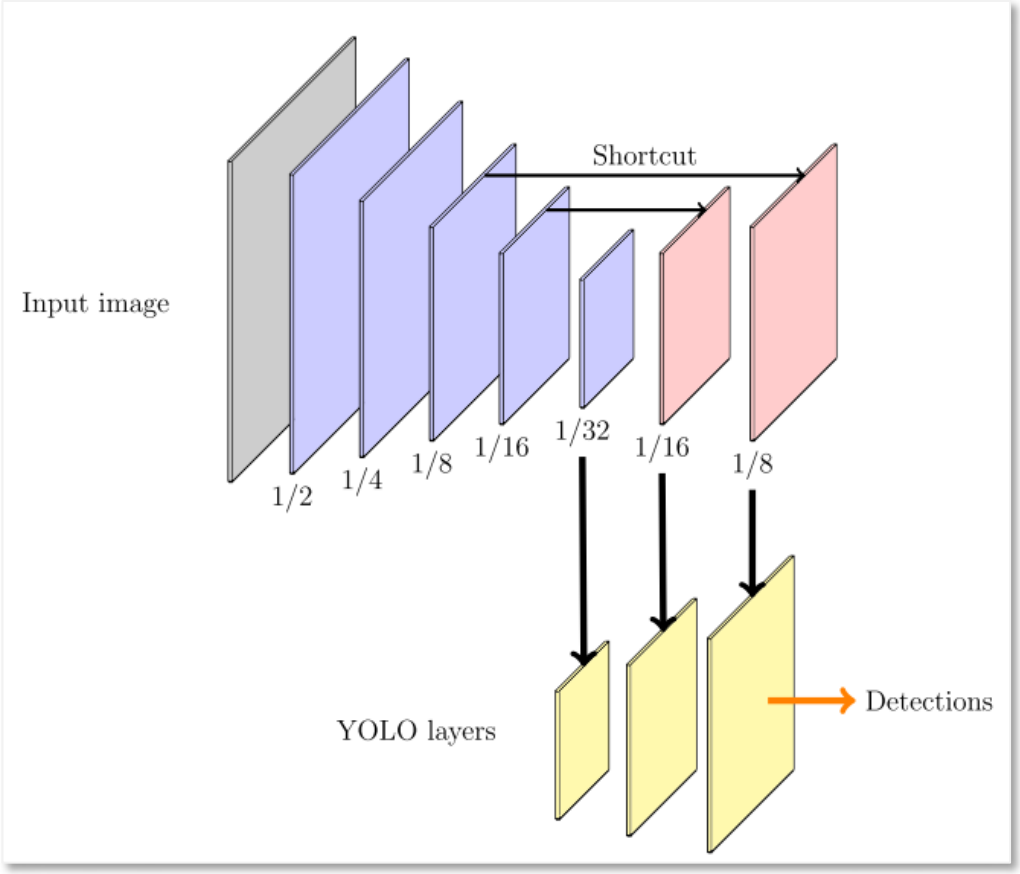


Figure 23: Architecture of the detector YOLOv3
Source: [5]

Figure 23 shows the architecture of the standalone YOLOv3 network [5]. YOLOv3 uses a Darknet-53 backbone with size-decreasing layers for feature extraction then does an upscaling of the feature maps to feed those features to three different YOLO layers, in which object detection takes place at different scales. YOLOv3, opposed to YOLOv1, evolved to be an anchor-based approach, and thus these YOLO layers receive as parameters a series of aspect ratios to serve as anchor references, as well as the indices that link each YOLO layer to the set of aspect ratios that they work with. The same concept is kept in YOLOv4 at the network's head, which is based on an upscaling convolutional structure leading to YOLO layers that perform detection.

2.4.6 Scaled YOLOv4 (2020)

The works presented in [36] show a scalability approach for the YOLOv4 architecture introduced in [33]. Same as EfficientDet, the purpose of optimal scaling is to get a set of models suitable for a broad range of applications with an optimal accuracy-speed trade-off. Opposed to EfficientDet scaling, which consisted on broadcasting a network in various dimensions, the scaling of YOLOv4 is performed by implementing architectural modifications. For arriving to the scaled YOLOv4 models, the steps followed in the work of [36] are the following:

- Design an optimal base model: Which will be a CSP-ized version of YOLOv4.
- Find the optimal way to scale it: Which can be followed in detail at [36].
- Propose a series of upscaled/downscaled YOLOv4 networks.

The concept proposed in *CSPNet* [37] can be applied to multiple CNN architectures to greatly reduce the required amount of parameters and computations. On Darknet networks, the number of FLOPs has been seed to be reduced down a 50% by CSP-izing its architecture [36]. The first step towards applying the scalability principles is to design a base network to start from. This network is chosen to be YOLOv4, which is already designed for real-time performance on general GPUs. A better speed-accuracy trade-off can be attained by CSP-izing its architecture.

The resulting model is *YOLOv4-CSP*, which consists of a CSP-ized YOLOv4 model. From this starting point, tiny and large YOLOv4 models are designed by following the scaling rules exposed at [36]. Thus, the models proposed at [36] are:

- ***YOLOv4-CSP***: Regarding the backbone, the design of CSPDarknet53 is modified by reverting the first CSP stage into an original Darknet residual layer. For the neck, the architecture of PAN is CSP-ized, cutting computations by a 40%, the SPP block placed in the middle positon of the first computation list group of the CSPPAN.
- ***YOLOv4-tiny***: Designed for low-end GPUs. Backbone CSPOSANet with PCB architecture, neck and head based on the already existing tiny model YOLOv3-tiny.
- **YOLOv4-large**: Meant for high-resolution images in pursue of higher accuracies. Born from designing a fully CSP-ized model *YOLOv4-P5* and scaling it to *YOLOv4-P6* and *YOLOv4-P7*.

2.5 Object detection state-of-the-art

MS COCO benchmarking for state-of'the-art detectors is shown in Figure 24 and Figure 25.

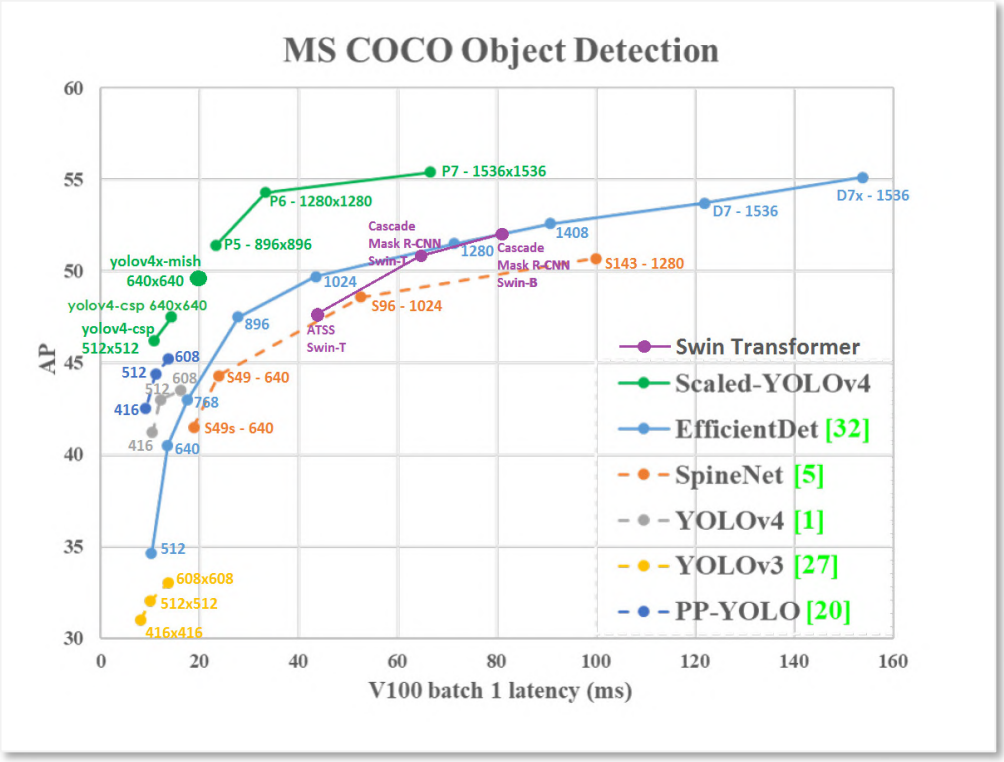


Figure 24: State of the art of different object detectors in MS COCO dataset I
Source: [34]

Regular-size models

Method	Backbone	Size	FPS	AP	AP_{50}
YOLOv4-CSP	**CD53s**	512	97/93*	**46.2%**	**64.8%**
EfficientDet-D0 [31]	EfficientNet-B0 [30]	512	91*	34.6%	53.0%
EfficientDet-D1 [31]	EfficientNet-B1 [30]	640	74*	40.5%	59.1%
YOLOv4-CSP	**CD53s**	640	73/70*	**47.5%**	**66.2%**
YOLOv3-SPP [26]	D53 [26]	608	73	36.2%	60.6%
YOLOv3-SPP ours	D53 [26]	608	73	42.9%	62.4%
PP-YOLO [19]	R50-vd-DCN [19]	608	73	45.2%	65.2%
YOLOv4 [1]	CD53 [1]	608	62	43.5%	65.7%
YOLOv4 ours	CD53 [1]	608	62	45.5%	64.1%
EfficientDet-D2 [31]	EfficientNet-B2 [30]	768	57*	43.0%	62.3%
RetinaNet [16]	S49s [5]	640	53	41.5%	60.5%
ASFF [17]	D53 [26]	608*	46	42.4%	63.0%
YOLOv4-P5	**CSP-P5**	896	43/41*	**51.4%**	**69.9%**
RetinaNet [16]	S49 [5]	640	42	44.3%	63.8%
EfficientDet-D3 [31]	EfficientNet-B3 [30]	896	36*	47.5%	66.2%
YOLOv4-P6	**CSP-P6**	1280	32/30*	**54.3%**	**72.3%**
ASFF[17]	D53 [26]	800*	29	43.9%	64.1%
SM-NAS: E2 [38]	-	800*600	25	40.0%	58.2%
EfficientDet-D4 [31]	EfficientNet-B4 [30]	1024	23*	49.7%	68.4%
SM-NAS: E3 [38]	-	800*600	20	42.8%	61.2%
RetinaNet [16]	S96 [5]	1024	19	48.6%	68.4%
ATSS [41]	R101 [10]	800*	18	43.6%	62.1%
RDSNet [35]	R101 [10]	600	17	36.0%	55.2%
YOLOv4-P7	**CSP-P7**	1536	16/15*	**55.4%**	**73.3%**
CenterMask [14]	R101-FPN [15]	-	15	44.0%	-
EfficientDet-D5 [31]	EfficientNet-B5 [30]	1280	14*	51.5%	70.5%
ATSS [41]	R101-DCN [4]	800*	14	46.3%	64.7%
SABL [34]	R101 [10]	-	13	43.2%	62.0%
CenterMask [14]	V99-FPN [14]	-	13	46.5%	-
EfficientDet-D6 [31]	EfficientNet-B6 [30]	1408	11*	52.6%	71.5%
RDSNet [35]	R101 [10]	800	11	38.1%	58.5%
RetinaNet [16]	S143 [5]	1280	10	50.7%	70.4%
SM-NAS: E5 [38]	-	1333*800	9.3	45.9%	64.6%
EfficientDet-D7 [31]	EfficientNet-B6 [30]	1536	8.2*	53.7%	72.4%
ATSS [41]	X-32x8d-101-DCN [4]	800*	7.0	47.7%	66.6%
ATSS [41]	X-64x4d-101-DCN [4]	800*	6.9	47.7%	66.5%
EfficientDet-D7x [31]	EfficientNet-B7 [30]	1536	6.5*	55.1%	**74.3%**
TSD [29]	R101 [10]	-	5.3*	43.2%	64.0%

Tiny models

Model	Size	FPS_{1080ti}	FPS_{TX2}	AP
YOLOv4-tiny	416	**371**	**42**	21.7%
YOLOv4-tiny (3l)	320	252	41	**28.7%**
ThunderS146 [21]	320	248	-	23.6%
CSPPeleeRef [33]	320	205	41	23.5%
YOLOv3-tiny [26]	416	368	37	16.6%

Figure 25: State of the art of different object detectors in MS COCO dataset II
Source: [36]

3 DATA COLLECTION

3.1 Introduction

3.1.1 Requirements & approach

It is well-known that training object detection models is a data-hungry process. Techniques such as data augmentation or pseudo-labelling can help training when data is scarce, but the most powerful tool towards using few training pictures is transfer learning. Transfer learning allows to start the convolutional layers of object detection models from a pre-trained point, usually trained on a very big, difficult problem such as the MS COCO object detection challenge. These pretrained weights hold lots of valuable low-level information such as common shapes of edges and textures. By using these high-quality pretrained weights, better models can be produced much faster, requiring less data to catch on low-level features. In Figure 26, feature maps learned by a CNN network at different grades of complexity are shown. Earliest convolutional layers focus on low-level information such as edge shapes or dots, while deeper ones manage to build representations of more complex items such as faces or cars.

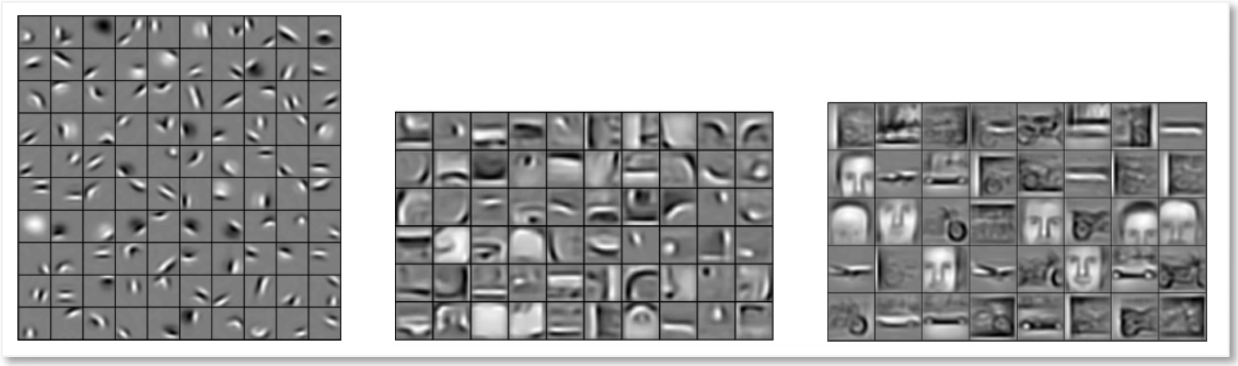


Figure 26: Progressively richer feature maps from left to right
Source: [38]

Data can be either imported or produced. In this case, the items that are to be detected have a very specific appearance, so that finding data over the internet is not easy. If we wanted to detect, for instance, any backpack, finding thousands of backpack images wouldnt't be hard: this could be done from random google pictures, by labelling them ourselves. Also, in this case the backpack is a common enough item, so it may be present in big, public and labelled datasets. Producing data would mean to actually take pictures of the items to be detected ourselves, then labelling them.

Another possibility would be producing synthetic images, be it from real or virtual sources. Pseudo-realistic images coming, for instance, from video games have proven to be representative enough for training models that are then deployed in the real world in some scenarios (and viceversa).

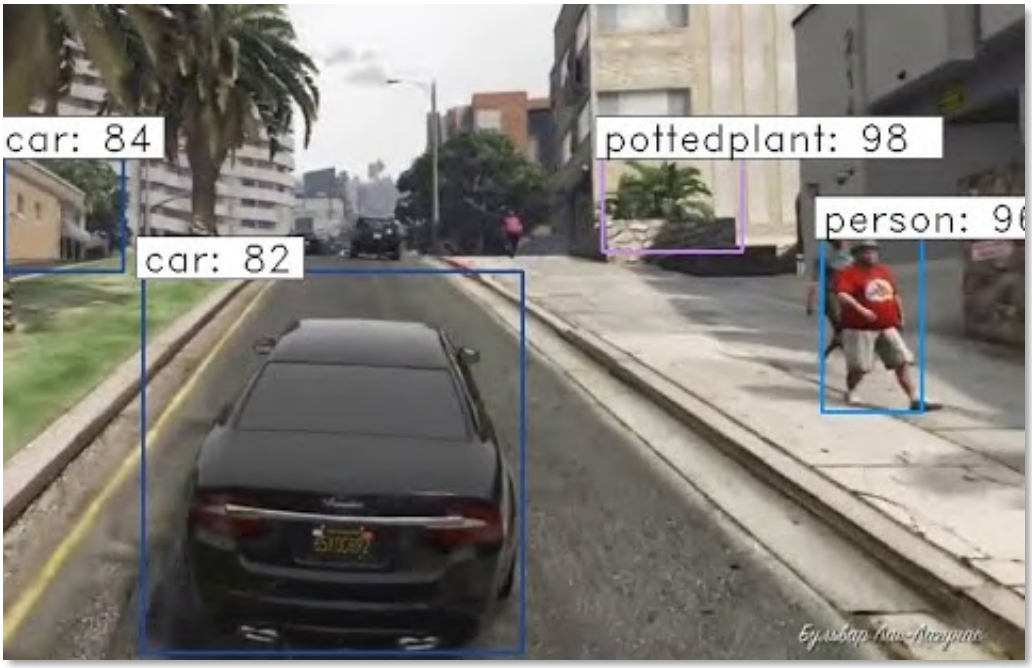


Figure 27: Faster R-CNN inferring some detections on the game GTA:V
Source: https://www.youtube.com/watch?v=CyvN4hZTMxo

Figure 27 shows a Faster-RCNN performing object detection on Grand Theft Auto: V footage. Over the years, many authors have experimented with synthetic data to train computer vision models [4]. Depending on how it is produced, labelled synthetic data can be more easily obtained than going through the effort required to manually label a whole dataset.

When assembling a dataset, it is also important to pay attention to the split that is to be performed. Not all the pictures that are to be produced will be used to train the network: some will also be needed to evaluate it. A typical split for training an artificial neural network could be 50% training, 30% test, 20% validation. While this can vary substantially depending on the application, the author... etc. it is undeniable that some data must be spared for evaluation. This is even more critical when different models are to be compared: the baseline on which they are benchmarked must be strong and representative, so a rich test dataset must be provided. The validation set size is less critical in this case, since it is generally used to provide information on where to cut the training: in computer vision we generally apply data augmentation techniques that aim to extend the representational power of the train set and, at the same time, greatly diminish the overfitting potential of the networks since no two training examples are the same.

Another item to be addressed is the class imbalance. To prevent model bias, the dataset should have balanced class, this is, an even number of instances in all of the pictures. This is a machine learning basic check that can be more or less critical depending on the application.

In this work, three different approaches were adopted for collecting a benchmarking dataset: some portion of the data was imported, some produced on real images, some extra produced as synthetic images. A fourth approach for generating synthetic was initially tested but later discontinued due to finding a better alternative. The whole joint procedure to produce a benchmarking dataset project is outlined in the schematic of Figure 28. The upcoming sections will discuss each of these approaches in more detail.

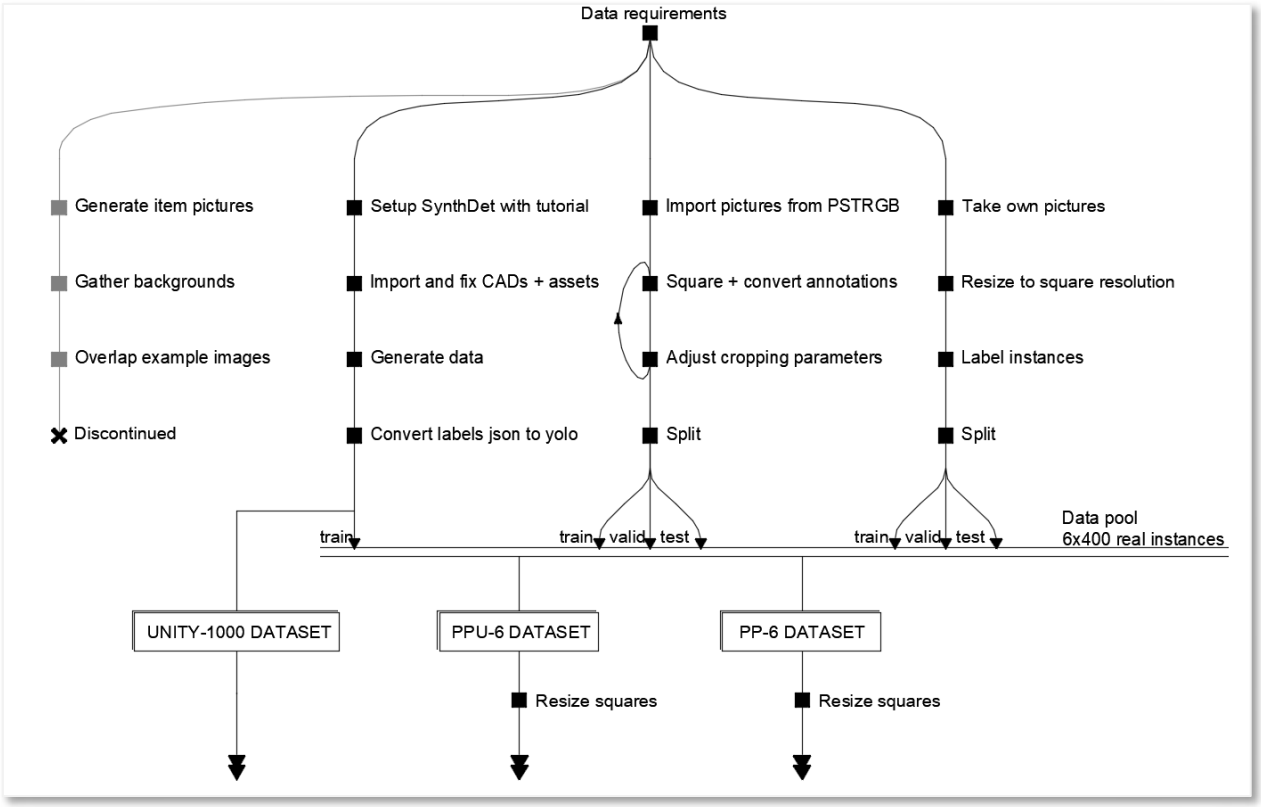


Figure 28: Data collection strategy followed in this work
Source: Self-made

3.1.2 Object detection data formats

Object detection labelled data generally consists on:

- **A set of pictures**: usually 3-dimensional RGB matrices with shape *width x height x channel*, in any common format such as *.png* or *.jpg*. Though they can work, image extensions longer than three digits (*.jpeg*) may require some adjusting in the preparation pipeline simply because they are less common.
- **A set of annotations**: which may be implemented differently and contain different information depending on the platform the network is to be run on. Two different relevant annotation formats are used in this work: YOLO and MS COCO annotations.

In one hand, YOLO bounding box annotations are stored in text files placed in the same directory as its source image [39], with the same name excluding extension. A full example of a YOLO annotation is shown in Figure 29. Each line in the annotation text file contains a separate bounding box, with the following parameters separated by a single space.

- The class the item inside in the box belongs to.
- Horizontal position per-unit (divided by image width) of the bounding box center.
- Vertical position per-unit (divided by image height) of the bounding box center.
- Bounding box width, per-unit (divided by image width).
- Bounding box height, per-unit (divided by image height).

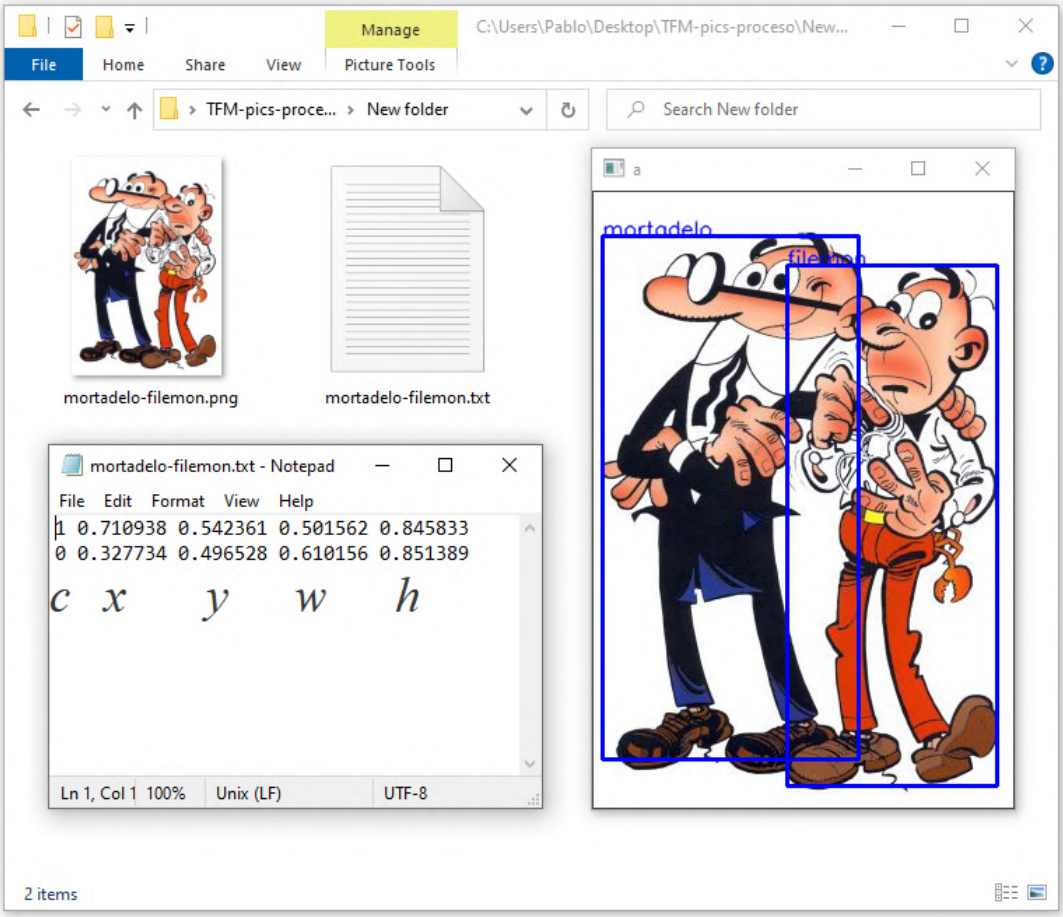


Figure 29: Full example of a YOLO annotated picture
Source: Self-made

On the other hand, MS COCO object detection annotations for many pictures are stored in common *.json* files [40]. If the dataset is small all annotations can fit in a single file, otherwise this file can be split in multiple *.json*, all with identical structure but containing information of different pictures. MS COCO annotations are a bit more complex than those of YOLO and may include some extra information, such as the image license or the bounding box area. They also store the bounding box coordinates not in per-unit values and the actual size of the picture. The exact data structure for both annotations and results is provided in the COCO Dataset webpage at [40]. Figure 30 shows the MS COCO annotations for the very same picture in Figure 29, both if it was a plain annotation or an inference result. Note that in the MS COCO annotation results some information required at [40] is not shown since it was generated from a YOLO annotation and thus lacks content for some fields.

The reader may have noticed that a MS COCO result format is specified, but not a YOLO one. Besides hosting a challenge that accepts submissions in a normalized format, both annotations and results must be provided in MS COCO format to compute the MS COCO performance metrics with the official MS COCO tools, available at [19], which are to be used as a benchmarking standard in this work. This is the reason that the result format is also outlined in this section, while a particular YOLO result format can be equal to a YOLO example annotation.

Annotation


```
{
  "images": [
    {
      "file_name": "mortadelo-filemon.png",
      "height": 440,
      "width": 300,
      "id": 0
    }
  ],
  "categories": [
    {
      "supercategory": "Disinfect_5obj",
      "id": 1,
      "name": "mortadelo"
    },
    {
      "supercategory": "Disinfect_5obj",
      "id": 2,
      "name": "filemon"
    }
  ],
  "annotations": [
    {
      "id": 1,
      "image_id": 0,
      "bbox": [
        138,
        52,
        150,
        372
      ],
      "area": 55800,
      "iscrowd": 0,
      "category_id": 2,
      "segmentation": []
    },
    {
      "id": 2,
      "image_id": 0,
      "bbox": [
        6.5,
        31,
        183,
        374
      ],
      "area": 68442,
      "iscrowd": 0,
      "category_id": 1,
      "segmentation": []
    }
  ]
}
```


Result


```
[
  {
    "image_id": 0,
    "category_id": 1,
    "bbox": [
      6.5,
      31,
      183,
      374
    ],
    "score": 0.99
  },
  {
    "image_id": 0,
    "category_id": 2,
    "bbox": [
      138,
      52,
      150,
      372
    ],
    "score": 1.0
  }
]
```


Figure 30: MS COCO annotations for the picture in Figure 29
Source: Self-made

3.2 *Cut, Paste and Learn* from virtual models

Cut, Paste and Learn is a concept proposed in the research paper [41] to produce synthetic data from real image sources. In this method, a set of instance pictures are snipped out of their original background, then these foreground items are randomly pasted on top of a separate set of background images, and because the place they are pasted can be known, bounding box annotations come for free. A visual example of the procedure is shown in Figure 31.

This way, items can be virtually placed at random in any pose within the backgrounds, allowing for the generation of a pseudo-infinite number of annotated images. In the right settings, it is proven that a training a model with 10% of real data plus a 90% produced under this approach might perform better than a model using only real data [41], since the variability of the train set becomes much higher. While promising, this method still has some drawbacks:

- The time required to snip out the foreground items is higher than just labelling bounding boxes.
- The lighting conditions on the images may originate some bias if the variability of the foreground item lights is not enough.
- If the items are not reasonably rigid (or at least consistent in silhouette) the required number of foregrounds might increase considerably for the model to behave well.
- It still requires access to some extent of data.

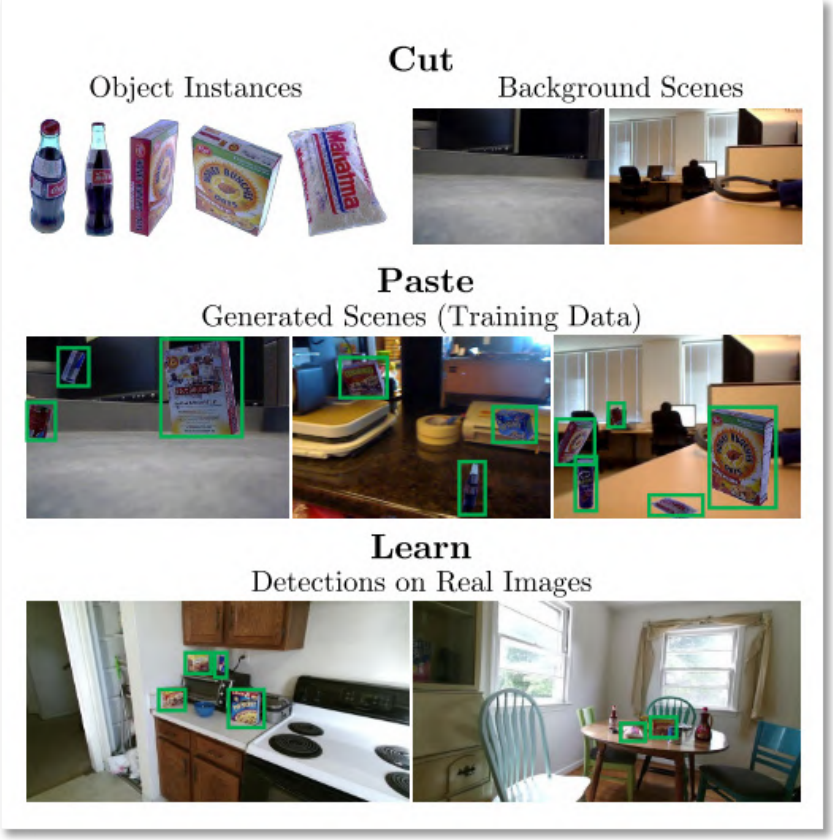


Figure 31: ***Cut, Paste and Learn*** **overview**
Source: [41]

By the time the adoption of this method was considered, there were still no real sources available for building such dataset. There were, however, artifact 3D models available for the DARPA challenge in simulation environments at the Ignition Robotics web such as [42]. Since there was no real data available and at the time training a model on the virtual environment was an open possibility, an approach based on *CutPaste&Learn* was considered.

While available to download, the 3D models posted at [42] can be previewed online from a web browser over a solid-color background. With screen capture and very little image processing, it is possible to automatically extract snips of these models from different angles very quickly, making the segmentation part of the *Cut, Paste and Learn* approach extremely quick and painless. Backgrounds were then extracted by converting drone footage video into frames, then using those frames as backgrounds, again in a very automated fashion than makes the method as simple as it can be. Figure 32 shows an outline of this approach.

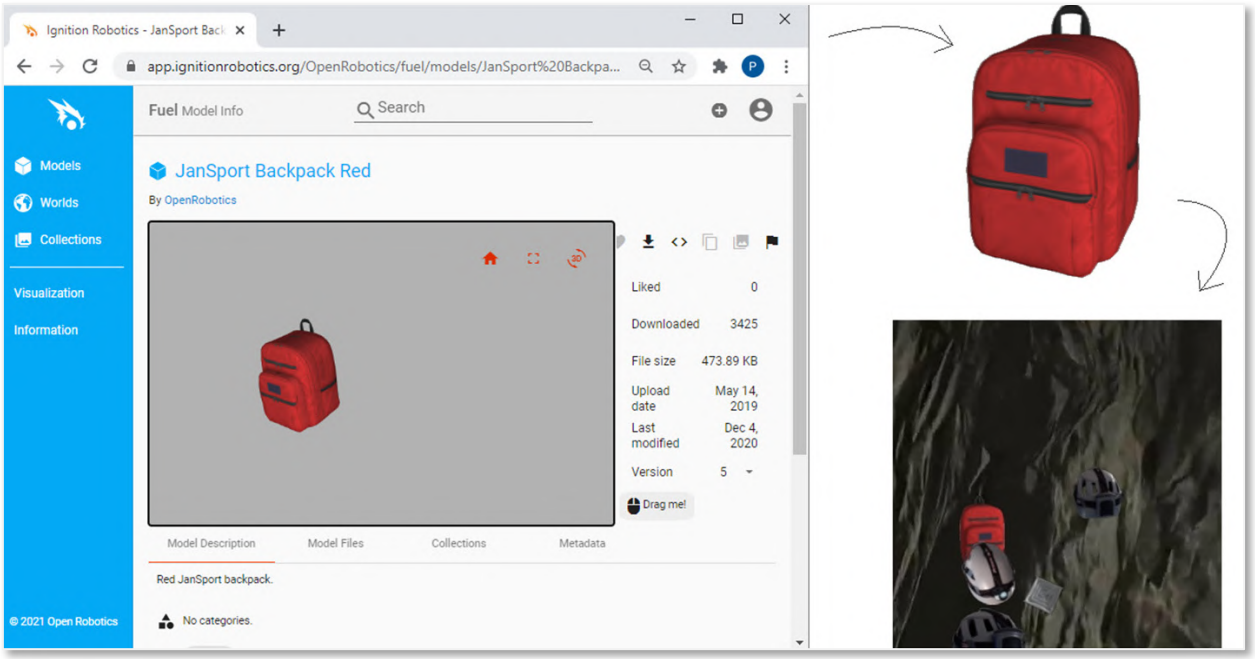


Figure 32: *Cut, Paste and Learn* concept applied to synthetic DARPA artifact models
Source: Self-made

Despite the concept being pretty basic, this method produces quite photorealistic pictures with minimal effort. While the *Cut, Paste and Learn* drawbacks related to labelling effort and data accesibility are completelly nullified, it is true that the rest of them hold. Furthermore, the foreground instances are virtual, so its representativeness on real world scenarios is uncertain.

While the results looked promising, specially for training models meant to be run on simulation environments, this concept was discontinued as soon as the SynthDet package was discovered.

3.3 Unity Perception's SynthDet

SynthDet [43] is a tool for the cross-platform game engine Unity, offering a pipeline designed for generating photorealistic images from 3D models to provide synthetic training data for computer vision models. The goal of this tool is to produce rendered pictures of a randomized scenario with a foreground layer of interesting objects and a background layer filled with higly distracting clutter. Indeed, labels are generaed automatically for the items appearing in the images. This tool can produce both object detection (in *.json* format) and instance segmentation labels by default, though other annotation types can be implemented via coding custom scripts.

It is shown in the research paper [44] that this is a very effective method to obtain an infinite amount of data to train convolutional neural networks, and that it is posible to produce object detection models solely trained on synthetic data that outperform those that are trained on real one. To achieve this, the authors follow a strict curriculum approach in which the pose and size of the shown items is progressively updated during training: the orientation and size of the objects are not random, but deterministic. Objects are progressively rotated to shown the network all of the relevant angles at one initial size, then this process is repeated showing the object in a smaller size, etc. An outline of this approach is shown in Figure 33.

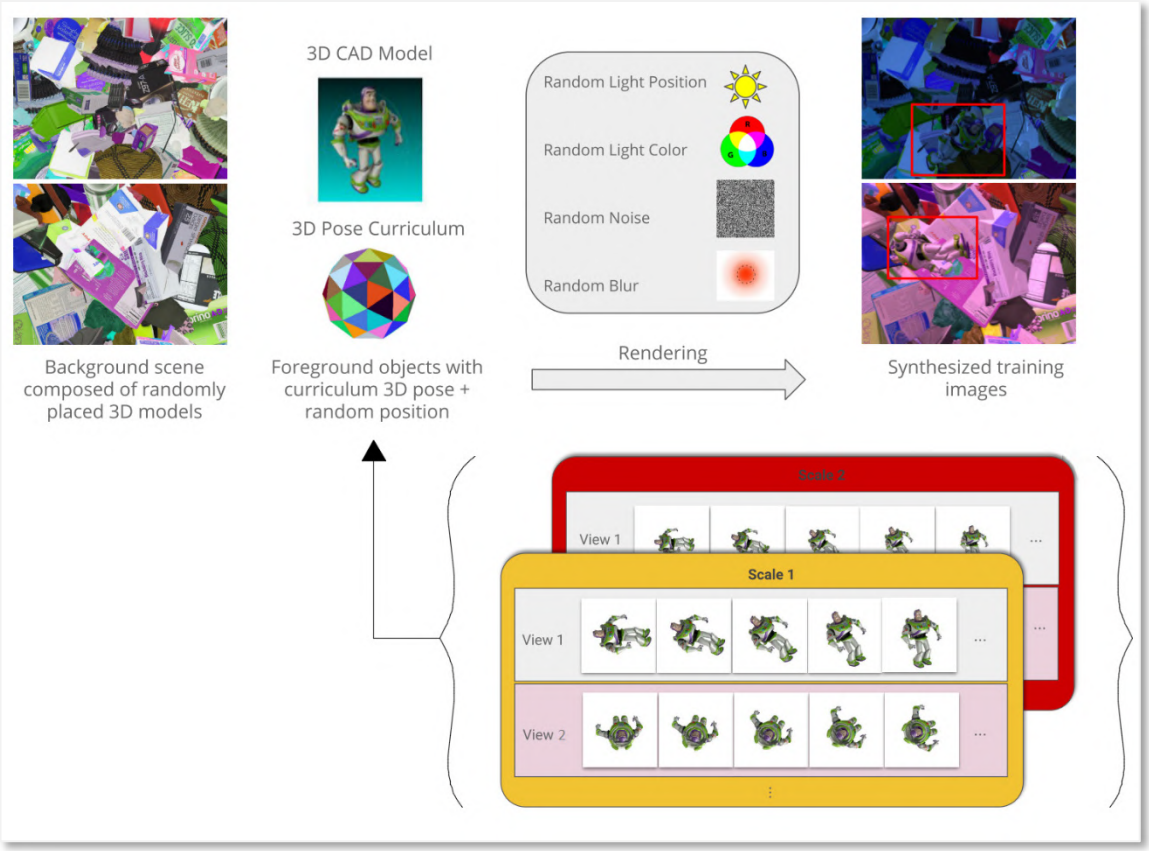


Figure 33: Synthetic data generation pipeline used in [44]
Source: [44]

After the assets and parameters are set, running a SynthDet simulation will place the foreground and background items, as well as the light source, in a randomized pose for a number of iterations, taking a picture at each of them. Figure 34 shows an overview on how these items are placed in the environment during one arbitrary iteration, while Figure 35 shows a series of pictures that would be produced with this setup.

While requiring a bit more effort to get running, it can be seen that this approach poses several advantages over the previous concept based on *Cut, Paste and Learn*:

- Increased photorealism via rendering.
- No need for edge blend-in.
- Randomized, rich, much more distracting backgrounds.
- Variability in light angle and hue.
- Presence of shadows.

However, this method also has the following drawbacks:

- High-quality 3D models are required, usually expensive and hard to find for free.
- First-time implementation of a new set of items is not easy.
- Limited representativeness of non-rigid items.
- High skills on 3D modelling and Unity are required to do anything more than random pose item placing, even though the potential of the tool is way deeper.

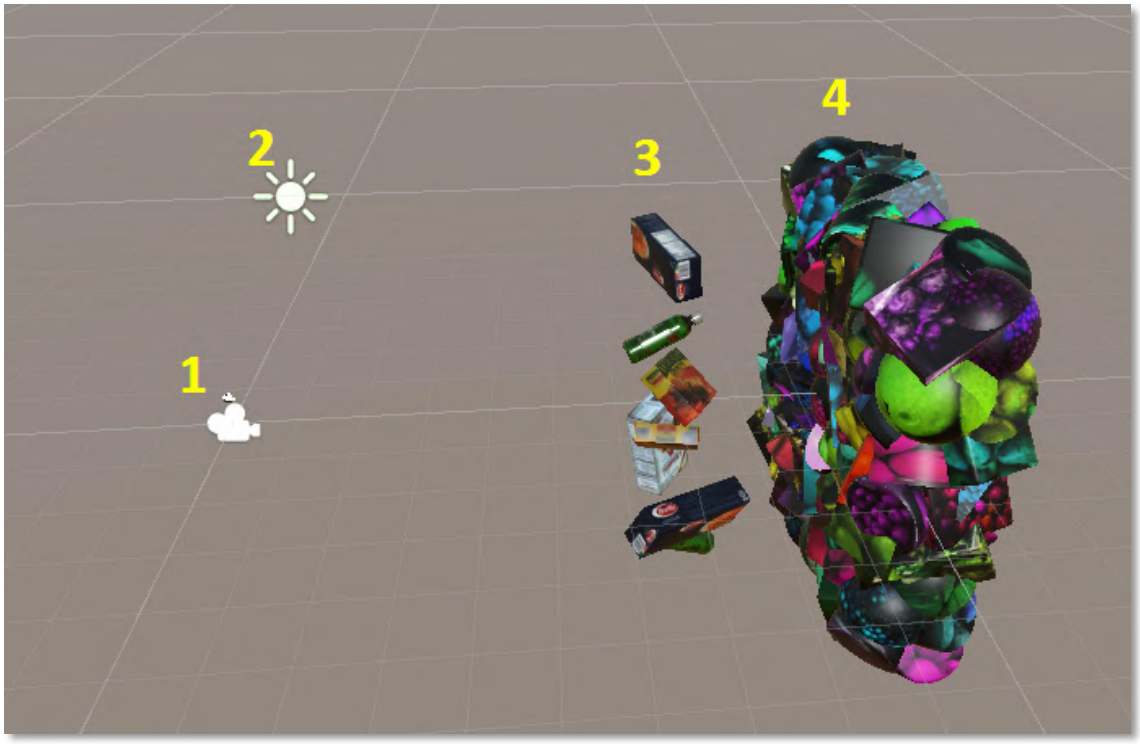


Figure 34: SynthDet virtual setup: camera, light, foreground items and background clutter
Source: Self-made

Figure 35: SynthDet output for the setup displayed in Figure 34
Source: Self-made

To generate annotated images for the DARPA artifacts, the following steps were followed:

- Follow the tutorial at [45] to set up the Unity environment, including default assets and scripts.
- Prepare 3D models and import them into the project:
  - Retrieve 3D models available at [42].
  - Convert the 3D models from *collada* to *.prefab* by erasing a few lines from the *.dae* mesh files and importing the models into Unity.
  - Heal incomplete/missing surfaces.
  - Resize models to a baseline size.
  - Edit textures to obtain a realistic item reflectiveness/color.
- Write a custom script for random item scaling and apply it to the foreground items.
- Generate annotated pictures with randomized foreground poses.
- Convert annotations from *.json* to YOLO.

As outlined in the previous list, a few issues were found during the importing of the models. Since the models at [42] were meant to be used in a realistic simulation environment, in the sense that they would be placed on floors or walls, the rope and vent (finally not included in this work) were only partially drawn. Since SynthDet will place the items randomly rotated on the air, healing the back side of the rope was a necessary. For this, the mesh was cut in half and mirrored so that its back face would be a mirror copy of the front one. While this resulted in minor surface interference in the model, this was not considered a big issue since the main visual features of the rope still remain intact. On the other hand, after importing into Unity the models showed a slight glow, like they were covered in plastic film, so their textures and color were slightly adjusted for a mate, more realistic texture. The imported models before and after these changes are shown in Figure 36.

Among the assets that were used to generate synthetic data, some have been stored in this project's repository [46] for record and reproducibility. Most of the assets that were not included come by default or can be easily produced when following the environment setup, with the exception of background textures that should be loaded from any common picture that authors consider representative. The assets that were stored are:

- 3D models, uploaded as *.prefab* and should include texture and preserve said resizing and retexturing.
- Model textures, which should already be integrated in the *.prefab* files, but are additionally included in case of version mismatches or other issues that might require reloading the models.
- Scripts for randomizing light (essentially copied from the tutorial) and scale (self-written) within the simulation, as well as the tags that the items need to undergo the scripted modifications (see [45] for more details).

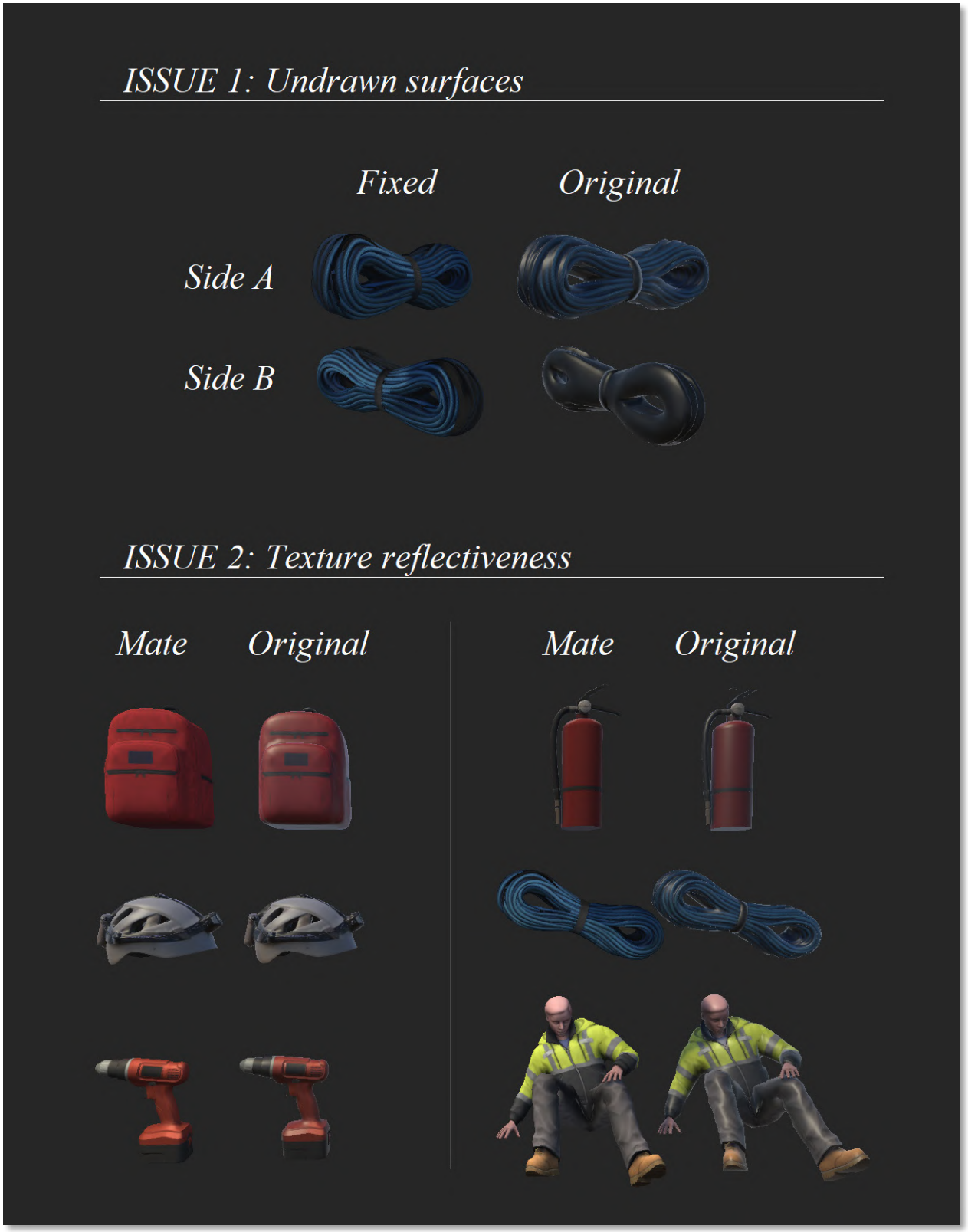


Figure 36: Ignition Robotics model issues after being imported into Unity
Source: Self-made

3.4 Importing data from PST-RGB

The *PST-RGB dataset* [47] contains a total of 3359 samples of RGB and thermal camera pictures with instance segmentation annotations for the items backpack, drill, extinguisher and survivor, developed for the purpose of taking part in the official DARPA Subterranean Challenge. Additional sources and information can be found in the Penn Subterranean Thermal repository at [48]. Each sample in the PST-RGB dataset is composed of:

- A thermal image, which is of no interest to this work.
- A depth map, which is of no interest to this work.
- A RGB picture of shape $1280x720x3$ and *.png* extension.
- An instance segmentation label picture with shape $1280x720x3$ and *.png* extension.

Instance segmentation label maps are plain RGB images in which pixels with value (0,0,0) denote background, while pixels with values (1,1,1) , (2,2,2) etc. denote the different classes. Since the RGB scale goes from 0 to 255, these instance maps are apparently black pictures. In Figure 37, an example sample is shown, with its RGB, its label map, and a remapped label map. This remapped label is just a processed copy of the standard label, in which a pixel-wise operation was made to remap the range $[0, 3]$ to $[0, 255]$, just so the instance masks appear visible for the reader.

To use the PST images as data, it was required to convert the masks to bounding boxes, which can be done with some basic image processing. Also, it was desired to import these images as pictures of square shape so, to mantain aspect ratio, the resizing is to be made by first resizing the picture height to a baseline value, then cropping out the sides to match width and height. Both the RGB sources and the instance segmentation labels have to undergo this process, then the bounding boxes can be computed around the resized instance masks.

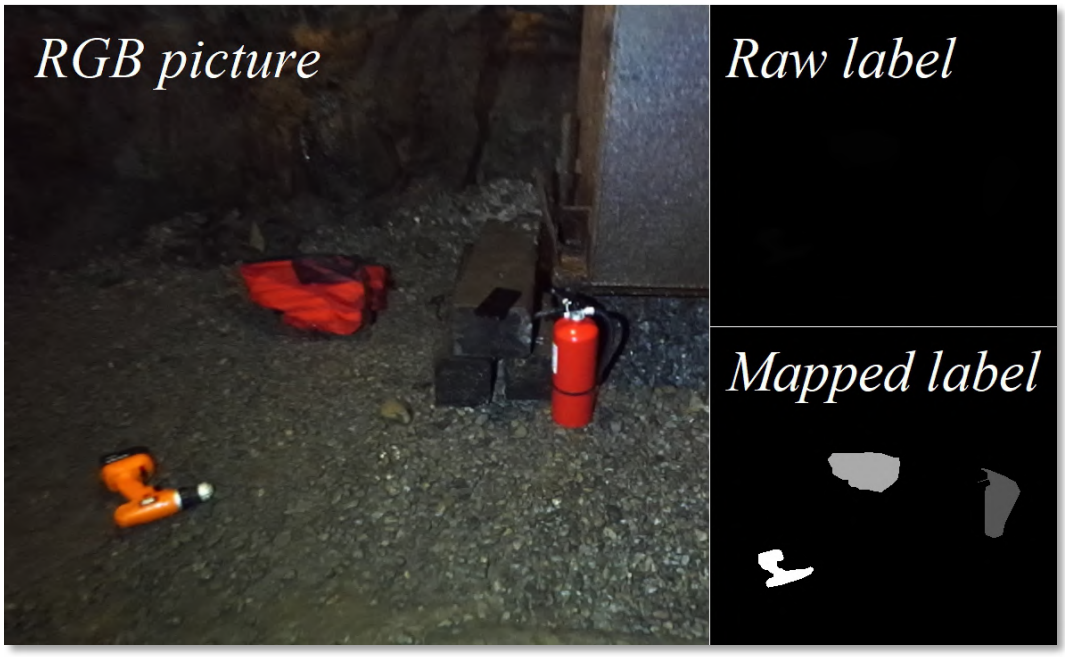


Figure 37: PST-RGB dataset sample
Source: Self-made

The conversion from pixel masks to bounding boxes can be problematic in some cases, and it required some tuning. The approach for accomplishing this is the following:

- Load the source label map as an RGB picture.
- Resize it without pixel interpolation. Pixel interpolation reduces noise, but damages our RGB annotations since bad pixels, with values other than its corresponding label, can be generated around true pixels.
- Perform a countouring operation, which will identify the contours of items.
    - Ideally, the process would end here, by drawing a bounding box around such contours. However, due to the mask annotations being noisy, the following additional steps are also needed.
- Draw a thick contour around the contouring results and fill the interior with solid color. This will slightly expand the area of the mask and will absorb loose pixels and blobs that should already be a convex part of the mask.
- Perform a new contouring operation. This time, if some pixel masks with the same value were brought together in the previous operation, they will be detected as a single contour and, thus, as a single item.
- Compute bounding boxes by checking the maximum and minimum coordinates of the contours in both axes.

The two most common issues that make these steps necessary, instead of just stopping at the 3rd item, are the following:

- **Discontinuous masks**: if a single mask pixel is surrounded by background ones, a detection box would pop around it if not for this workaround. Depending on the mask quality, this may lead to tens to hundreds of fake example $1x1$ bounding boxes being generated around item edges.
- **Partially occluded items with more than one visible region**: meaning that 2 or more different bounding boxes are generated for the same item. This is not an issue if these regions are representative, since they become additional occluded item instances, else they become noisy, useless annotations. This tends to happen a lot on backpack straps that stop being visible due to either items on top of them or coming out of the picture and then back in.

Indeed, a proper set of bounding boxes wasn't reached in the first try. The whole procedure was scripted and tested a few times by monitoring the number of instances in each picture, tuning some parameters like how thick should the expansion of the contour be (4th step) or how many times the steps 3rd to 5th should be repeated in series. When big outliers on the number of instances per picture stopped coming up, a quick visual inspection on the output was performed to detect and look into new anomalies and, around the 3rd visual inspection, the parameters used to convert the images were deemed satisfactory.

The scripts available at this project's repository at [46] have the parameters used to have labelled pictures undergo this process and its use is not limited to this particular dataset. If using them, it should be kept in mind that a little trial-and-error is an expected part of the process.

3.5 Producing real data by taking and labelling pictures

The PST-RGB dataset provided more than enough samples of images for the artifacts backpack, drill, extinguisher and survivor. The two remaining artifacts, helmet and rope, still required the collection and labelling of pictures, which was performed once they available for borrowing. Considering the findings of [5] and the time it took us to collect and label a batch of 100 pictures, it was estimated that a total of 400 real instances of each of these two items should satisfy the needs of this work.

The next steps were followed to accomplish the collection of the required real pictures:

- Take pictures of helmet and rope – both in the same picture and separate pictures – in different environments and from different angles.
- Resize pictures to square shape.
- Label the pictures using a labelling tool.

All of the pictures were taken at the main LTU facilities, most of them in the underground corridors, which provided a variety of settings with different features and light conditions. The pictures were taken both carefully and static but also while moving to promote blurred and unclear images. Some pictures were also taken on purpose to partially occlude the items or hide them in plain sight in places where visually similar items were around.

The pictures are labelled in YOLO format, due to the simplicity of this format and the availability of the Yolo Mark labelling tool [49], whose preview is shown in Figure 38.

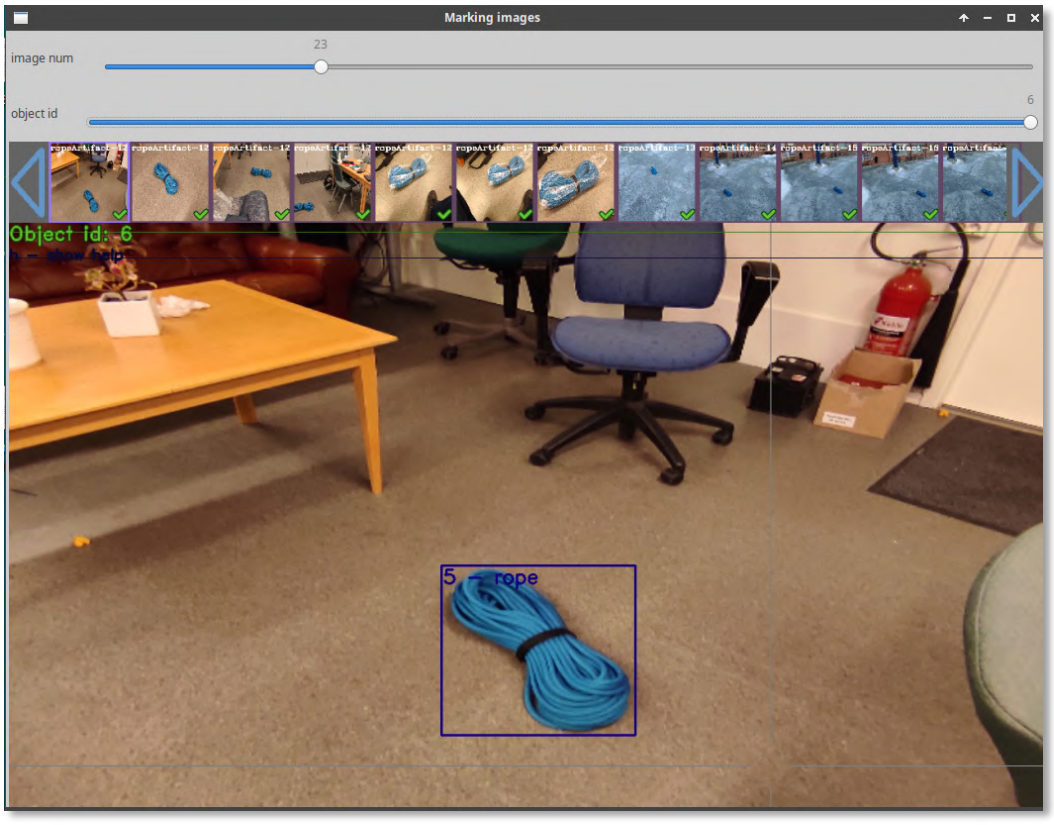


Figure 38: Detail of the annotation tool Yolo Mark
Source: Self-made

3.6 Performing a split

Splitting object detection data is not a trivial task, since it is desired that, in each split, a balanced number of instances of each class are present, however it is the pictures that have to be separated, each with an arbitrary number of instances of each item.

A first methodical approach for performing the split was building a table with the instances present in each picture. Set a target number of instances, first the pictures that have many instances are split and then the final number is approached with pictures that have only one item. The main issue of this approach is that the picture shuffling cannot be controlled very well, and there is a big risk that most of the pictures in one of the splits are very visually similar.

A second approach, which was finally used, was to develop a script to monitor a YOLO-annotated folder, telling the number of instances that were present in it whenever a certain key is pressed. Then, by manually moving images from one folder to the other by using a file explorer and checking with this tool, one can approach the target number of instances quickly. To solve the issue of the previous concept, the file explorer zoom was reduced so lots of pictures appeared on the screen at the same time, then entire columns within the same folder were moved around. Similar pictures are bound to be horizontally close (under the current settings of the file explorer), because they are usually taken close in time, their original names being probably very similar too, so by moving entire columns we make sure that every split has a varied set of images. While simpleton, this method proved to be equally effective, more reliable and way less time-consuming than the former. A quick overview of the Windows 10 enviroment while working with this tool is shown in Figure 39.

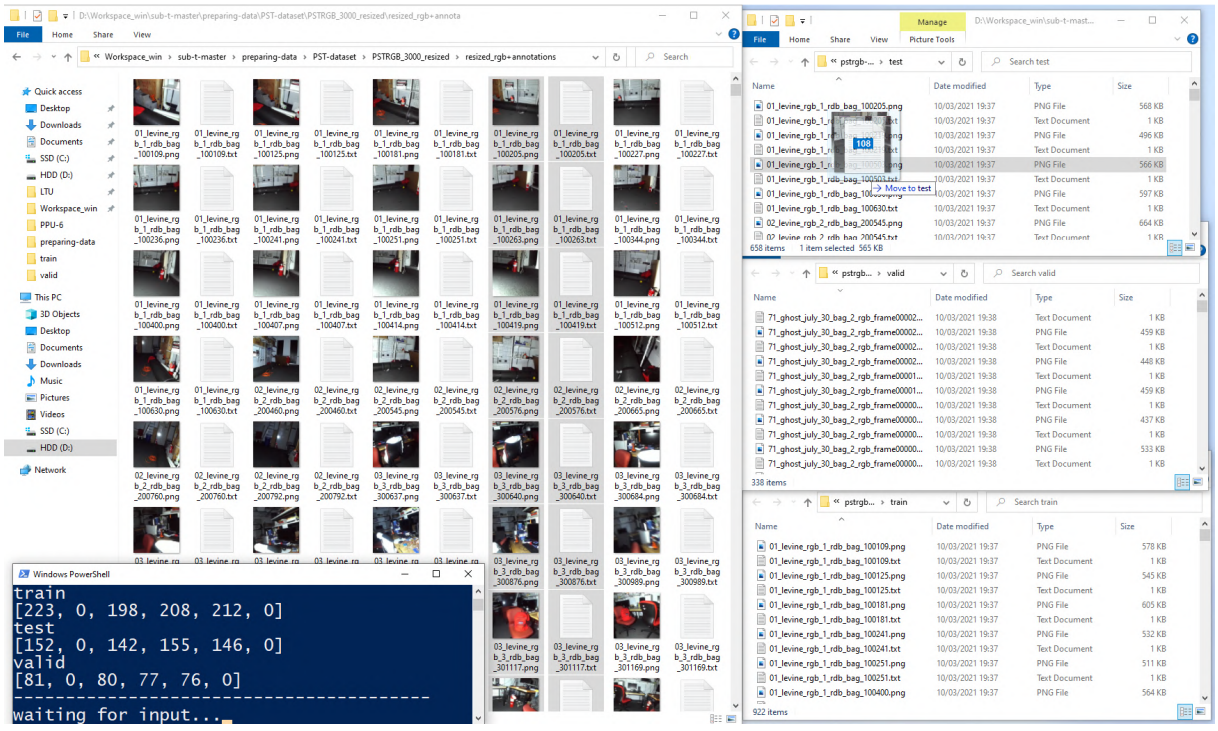


Figure 39: Overview of the split monitor tool usage
Source: Self-made

3.7 Data collection outputs

3.7.1 The PPU-6 dataset

The PPU-6 datset is the first baseline dataset produced in this word. It contains ~500 YOLO-annotated instances of each of the artifacts backpack, helmet, drill, extinguisher, survivor and rope; divided in a train, test and validation split, as shown in Figure 40. The PPU-6 dataset is assembled from:

- ~400 instances of the items backpack, drill, extinguisher and survivor borrowed from the PST-RGB dataset, present in all three splits.
- ~400 instances of rope and helmet artifacts, in pictures taken by the author at the LTU facilities, present in all three splits.
- A set of virtual pictures providing ~100 instances of each artifact produced with Unity SynthDet, present only in the train split.

A preview of the images contained in this dataset is shown in Figure 41, where pictures obtained from the methods outlined in the previous sections are shown.

The pictures in the PPU-6 dataset are all square pictures of arbitrary size. While the resolution itself is not that critical, the fact that the pictures are square is a matter of efficiency: networks that resize images at the input usually pad zeros to make images square, meaning that a significantly big area of the picture will be a useless zero matrix.

PPU-6 Split	Images	Backpack	Helmet	Drill	Extinguisher	Survivor	Rope
Train	912	320	336	306	317	295	309
Test	520	152	158	142	155	146	137
Valid	275	81	80	80	77	76	80

Figure 40: Number of pictures and instances of each item in the PPU-6 splits
Source: Self-made

3.7.2 The PP-6 dataset

The PP-6 dataset is a variation of the PPU-6 dataset so that only real images are used. Since the PPU-6 dataset only uses Unity SynthDet generated images on the train set, the test and validation splits remain equal, having the train set fewer instances, although all in real pictures.

3.7.3 The unity-6-1000 dataset

The unity-6-1000 dataset is a collection of 1000 virtual pictures of all the artifacts (backpack, helmet, drill, extinguisher, survivor and rope), generated with SynthDet. Its purpose is to test the performance of photorealistic virtual training data generated with randomized pose and scale for this application.

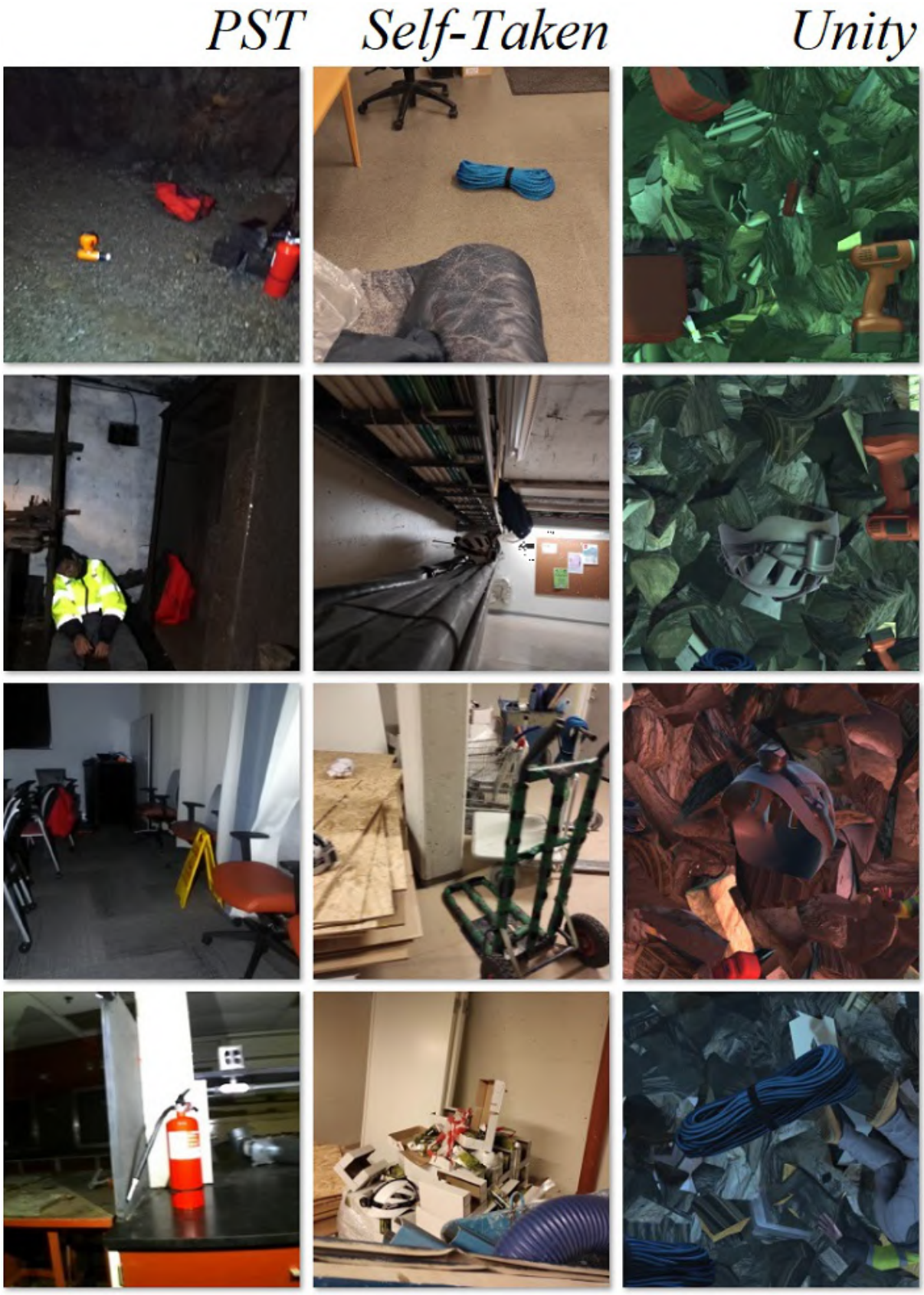


Figure 41: PPU-6 dataset showcase, each column showing pictures of a different source
Source: Self-made

4 BENCHMARKING OBJECT DETECTORS

4.1 Introduction

4.1.1 Requirements and approach

Towards selecing a best performing object detector for this particular application, a benchmarking among a series of state-of-the-art models is to be performed for the baseline datasets generated at Section 3. This benchmarking is to measure the performance of each by means of accuracy and inference speed. Accuracy is to be measured according to the standardized MS COCO metrics described in Section 2.3.3, from which the most relevant are the derivates of AP. On the other hand, inference speed is a characteristic of a model and does not require a trained model to be evaluated, being proportional on the number of computations in a forward pass. In other words, training is not required for measuring inference speeds.

The benchmarking of object detectors is performed by the completion of the following steps:

- **Gathering data**: Entirely addressed in Section 3.
- **Model selection**: Choosing a subset of models from the state-of-the-art that might be of interest due to their performance on baseline metrics. Discussed in Section 4.2.1.
- **Platform-specific checks**: For the models selected in the previous stage, this step comprises fetching official tools for their implementation (if available, else non-official), learning how to use them, and verifying that they can be used before committing to extensive tests on a platform whose use might be too challenging.
- **Model training**: Training all the models of interest, with model or platform-specific strategies.
- **Model testing**: Testing trained models on a common baseline test set. Despite using common MS COCO metrics, this step is also platform-specific implementation-wise.
- **Result analysis**: Analysis and discussion of the results obtained in the testing stage.
- **Further testing**: Consider performing additional tests if the results are inconclusive, looping the very same steps while considering what has been learned so far.

Implementation-wise, the accuracy metrics will be computed with the baseline MS COCO tools available at [19], while inference speed is to be measured in a local machine with GPU accelerated graphics. Measured inference speeds are usually estimates and slightly unstable magnitudes, since the time it takes to do a forward pass may vary depending on what background processes are happening at the same time on the host machine.

Training will be performed both remotely by using cloud-computing services and locally on an available machine, the specs of which appear depicted on Figure 43. Training, testing and auxiliary data management is almost always performed on Linux operating systems, by using local versions of the official platforms in which the models of interest are published. While other implementations exist on more generalizable tools (Torch, TensorFlow), this work opts for a benchmarking only on trusted official implementations.

4.2 Experimental setup

4.2.1 Model selection

The best one-stage object detection models up to date appear plotted in Figure 42, in which their speed and accuracy on the MS COCO dataset is compared. In Figure 42, the horizontal axis represents inference time (left is faster) and the vertical axis the AP (higher is more accurate). For the model families contained in this plot, various baseline sizes have been tested for each of their architectures, defined in their respective research papers [28], [33], [36]. The numbers shown for each model represent its design baseline input resolution. Big models, such as EfficientDet D7, with an input resolution of 1536x1536 pixels, are meant for running object detection on high quality pictures on very powerful GPUs. This work focuses on lightweight computations and thus aims for smaller models, mostly within the red bounding box.

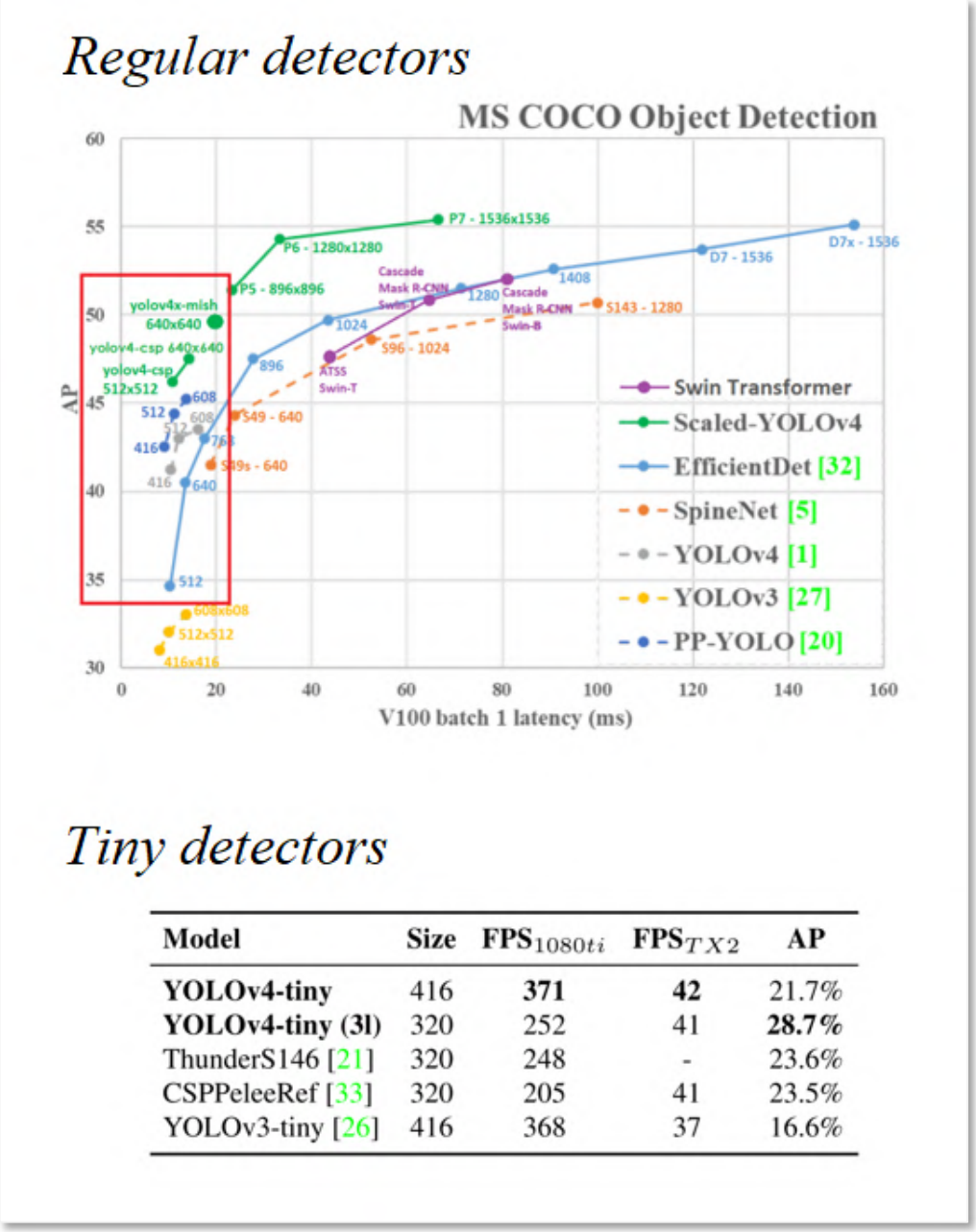
These models are all regular sized, meaning that, while one-stage, they still pursue high-accuracy. For faster applications, the trend has been to make scalable models that get smaller –and thus faster – when processing small-size pictures, at the cost of some accuracy. Aimed specifically for extremely fast applications and low-end GPUs, *portable* or *tiny models* have also been published through the years, the most relevant of them being also shown in Figure 42.

Towards choosing the models, as stated, inference speed has been a bigger concern than detection accuracy, because since a live detection is desired, achieving real-time is a relevant matter. The chosen models of interest towards performing a benchmarking are those in Table 3, in which each model resolution and implementation platform is also stated.

Resolutions are chosen to be nearby for all models and square, as well as the training images. While it is possible to feed a square model with rectangular images, the training efficiency will significantly decrease since a relevant portion of the input feature map will be padded with zeros. These resolutions allow crafting light-weight model-specific datasets by simply resizing the already-available PPU-6.

Table 3: Models selected for benchmarking

Model	**Platform**	**Network size**
yolov4-tiny	darknet	416x416
yolov4-tiny-3l	darknet	416x416
yolov4	darknet	416x416
yolov4-csp	darknet	512x512
yolov4x-mish	darknet	640x640
efficientdet-d0	google-automl	512x512
efficientdet-d1	google-automl	640x640



Model	Size	FPS_{1080ti}	FPS_{TX2}	AP
YOLOv4-tiny	416	**371**	**42**	21.7%
YOLOv4-tiny (3l)	320	252	41	**28.7%**
ThunderS146 [21]	320	248	-	23.6%
CSPPeleeRef [33]	320	205	41	23.5%
YOLOv3-tiny [26]	416	368	37	16.6%

Figure 42: State-of-the-art object detection models on the MS COCO dataset.
Source: [34], [36]

Regarding implementation platforms, there are two different tools involved:

- Darknet: Which is the official publishing platform for YOLO models. Originally developed by Joseph Redmon [26], this work will use Alexey Bochkovskiy's fork, available at [34]. Darknet is an open-source library written in C and CUDA. Source is available at `github.com/AlexeyAB/darknet`.
- Google-automl: Which is the name of the repository holding the official published implementation of EfficientDet models, written in a series of high-level Python scripts calling TensorFlow source code. Code is available at `github.com/google/automl`.

4.2.2 Practical insights on Darknet

The official implementations of YOLOv4 and most of its variations run on Darknet, which is a fast, open source neural network framework written in C and CUDA [26]. Darknet is easily imported and built from its GitHub source [34] and holds the official implementation of YOLOv4, as well as some other models both from the YOLO family and not. The tutorial at [50] was used to get acquainted with YOLOv4 implementation in Darknet. While running the tutorial on a Jupyter notebook is straightforward, using Darknet on a local machine will require a proper installation of CUDA and CUDNN to be able to perform GPU-accelerated computations.

A neural network model in Darknet is composed of the following files, with which one can perform inference on some data without requiring additional assets:

- Configuration file or *cfg* : A text file defining the architecture of a given network.
- Weights file or *weights* : A binary file containing the network's trained weights.

For the training of object detection models, the following assets are used:

- Configuration file or *cfg* : A text file defining the architecture of a given network.
- Weights file or *weights* : A binary file optionally containing pretrained weights.
- Train and validation data: Folder with annotated pictures, as shown in Section 3.1.2.
- *backup* directory : Empty directory where training output will be stored.
- *train.txt, valid.txt* : Text files indicating the path to each data sample.
- *obj.names* file : Text file with the names of each prediction category.
- *obj.data* file : A text file with paths to all the other assets.

Note that all the previous assets but the weights are case-specific. The configuration file must be modified so that the architecture is properly designed for the number of categories to be predicted [50]. During training, Darknet will generate a series of files in the backup folder. These files include updated intermediate weights that will be overwritten as training progresses or *checkpoints*, so that it is possible to resume the training if the machine crashes, as well as a *best.weights* file that will always store the checkpoint with the best performance on the validation set, which is computed at the end of every epoch. This means that the *best.weights* file is the output of the network's training, since Darknet is automatically "interrupting" the training at the optimal point, as pointed out in Section 2.2.4.

Darknet will also output a predefined training plot showing the evolution of the training and validation accuracy and loss. Darknet does not have a *graphical user interface* or *GUI* and is thus used from the *command line interface, CLI* or simply *terminal*. Numerical values from training that are typically displayed in terminal as it progresses can be extracted to a text file by using the proper terminal-specific applications.

AlexeyAB's distribution of Darknet already contains most of the models of interest by default, in a directory where numerous cfg files for popular modes are placed. Pretrained weights on MS COCO can be externally downloaded from his repository at [34], providing a sound checkpoint for starting training on custom data.

4.2.3 Practical insights on Google-automl

Google-automl is the name of the repository hosting the official code for EfficientDet models, available at [51], implemented as a series of Python scripts running TensorFlow source code. Thus, input and output are in high-level TensorFlow format. A walkthrough on how to work with these is available at [52], structure then recycled for running the networks on custom data. This implementation is messier to work with than Darknet, so training on a local installation was avoided.

A neural network model in Google-automl is composed of the following assets, which sometimes can be distributed over various files:

- A model directory : Binary files storing the architecture and metadata.
  - *model.data*
  - *model.index*
  - *model.meta*
- Checkpoint *.ckpt* file : A binary file containing the network's trained weights.

When performing inference, it is more convenient to convert this distributed version of the model into a compact *.pb* file with frozen weights. For the training of EfficientDet models, the following assets are used, again sometimes each distributed over various files:

- Train and validation data : Provided in TensorFlow closed format *.tfrecord*.
- Checkpoint *.ckpt* file : A binary file optionally containing pretrained weights.
- Output model directory : Where output model and .ckpt are updated when training.
- An hyperparam *.yaml* file : Defining some of the model's hyperparameters.

Same as Darknet, Google-automl is run with commands via the Python executable from a command interface. Besides hyperparameters, the yaml file also contains the number of classes plus their names and indices. The output model folder will contain a series of files, some defining its architecture and some storing numerous checkpoints. The stored checkpoints are both backup checkpoints and best, in the same fashion as Darknet to "interrupt" the training at the right moment.

Towards evalution, Google-automl automatically imports the package *pycocotools*, which contains Python bindings for computing the official MS COCO metrics shown at [16]. Thus, the built-in function for model testing already outputs standardized metrics, as opposed to Darknet, for which additional steps need to be followed.

4.2.4 Machine specifications

Three different alternatives were adopted in different stages of the training, in accordance to the available resources at each time, considering the admisible disk usage, RAM availability, training time, GPU power... etc.

- Stage 1 EfficientDet models were trained online using google colaboratory [53].
- Stage 1 YOLO models were trained on a remote laboratory computer.
- Remaining stages being performed on the author's *local machine*.

Figure 43: Local machine specifications
Source: Self-made

The specifications of the local machine used for training are shown in Figure 43, including:

- Hardware specifications
- Operative system and kernel
- CUDA version
- Darknet source and clone date

4.2.5 Training and evaluation in practice

The training and evaluation steps are very platform-specific. Towards the benchmarking, both start from a dataset annotated in YOLO format.

YOLO (Darknet) models are worked by following the following steps:

- Install CUDA and CUDNN on local machine for GPU accelerated computations.
- Clone and make Darknet with GPU and OpenCV support from its source at [34].
- Prepare training assets:
  - Assemble folder structure and generate path holder and other text file assets.
  - Modify each network's architecture to fit the target number of classes.
- Train and export a trained model, storing training terminal output to a file.
- Gather training plots produced by Darknet.
- Infer the test set and store results to a text file (the most convenient way to perform batch-inference for our purposes). From these text results:
  - Draw inferred bounding boxes to pictures.
  - Estimate the model's FPS by averaging the recorded per-image latency.
  - Convert these bounding boxes to .json MS COCO format.
- Convert the YOLO-annotated test set to MS COCO format.
- Use the MS COCO formatted test data and MS COCO inference results (MS COCO format required) to obtain AP COCO metrics with the official COCO tools at [19].

The number of training epochs and other hyperparameters were set according to the pointers provided at [50].

EfficientDet (Google-automl) models are worked by following the following steps:

- Build code blueprint in a colab notebook (online) for training and evaluation mostly from the tutorial at [52], plus data importing + resizing to network size.
- Setup a training folder structure on google drive for long-lasting cloud file hosting, containing the model, the required assets and an output folder for trained models.
- Resize the YOLO annotated images to network size and convert to .tfrecord format with very minor modifications to the script available at [54].
- Train and export a trained model, storing training terminal output to a file (validation AP metrics per epoch).
- Produce training plots from the collected data, since no plots are generated by default.
- Compute accuracy COCO metrics while performing batch-inference to images of the test set with the Google-automl built-in test and infer script.
- Install CUDA, CUDDN and Google-automl repository locally with GPU support, and perform built-in latency test to get the inference speed of each model [52].

Training is performed for an infinite number of iterations, until the best checkpoint has not been updated for a significant amount of time. The rest of hyperparameters and training choices were made according to the sample provided in the tutorial at [52].

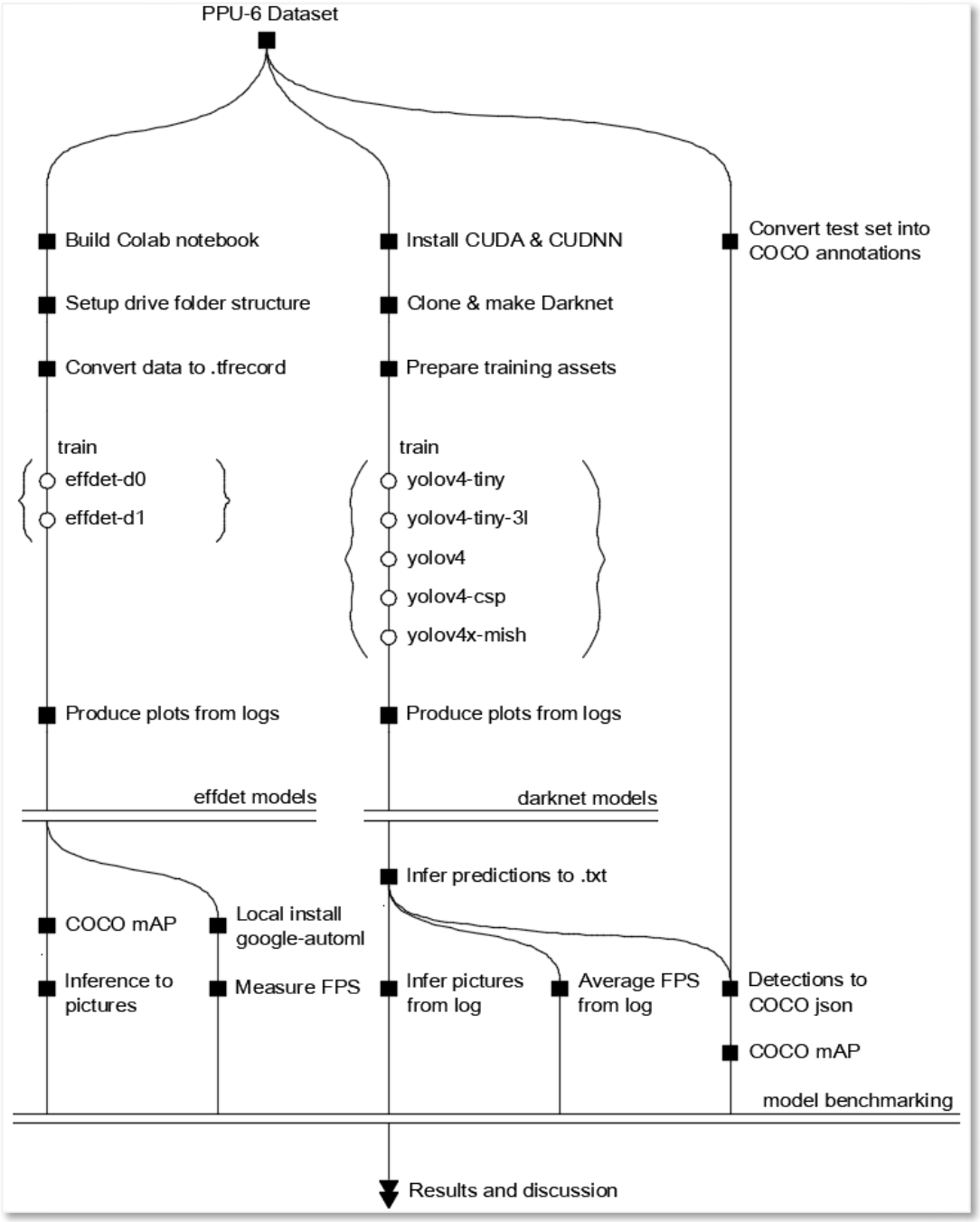


Figure 44: Training and evaluation roadmap
Source: Self-made

A complete roadmap on how the training and evaluation is performed is shown in Figure 44. More implementation details, pointers, commands, examples, etc. can be found at this project's repository, available at [46].

The benchmarking problem is approached by performing the steps previously described, repeating the whole experiment if the results are inconclusive until a model is seen to obviously outperform the others. After performing said experimentation, three different stages took place. While detailed benchmarking results can be found throughout Section 4.3, a brief summary for them can be found at Section 4.4.1.

4.3 Detector benchmarking

4.3.1 Benchmarking strategy

Comparison and benchmarking of the different models of interest is performed through a series of stages, for the first training results were unexpected and deemed unconclusive. Overall results and joint comparison of all tested models can be found in Section 4.4, while Sections 4.3.2, 4.3.3 and 4.3.4 will go over the particular results obtained in each stage and how they justify performing an additional one, before settling for a definitive best model choice at the third stage. These three stages pursue the following goals:

- *Stage 1: Default training on PPU-6* : Out-of-the-box comparison of all models.
- *Stage 2: Default training on PP-6* : Addressing synthetic overfitting suspicion.
- *Stage 3: Custom anchors on PPU-6* : Addressing performance gap on tiny models.

4.3.2 Stage 1. Default training on PPU-6

The purpose of this stage is to determine which models achieve a higher out-of-the-box accuracy in the baseline dataset PPU-6. All of the models gathered in Section 4.2.1 were trained under default configuration, as advised in [50] and [52], for the whole PPU-6 dataset, including synthetic samples. Figure 45, Figure 46, Figure 47, Figure 48, Figure 49, Figure 50 and Figure 51 show the training loss, validation *mAP* or both (platform-dependent) of the different models during training. Darknet plots are standardized and generated automatically, while EfficientDet plots were manually built from logged information.

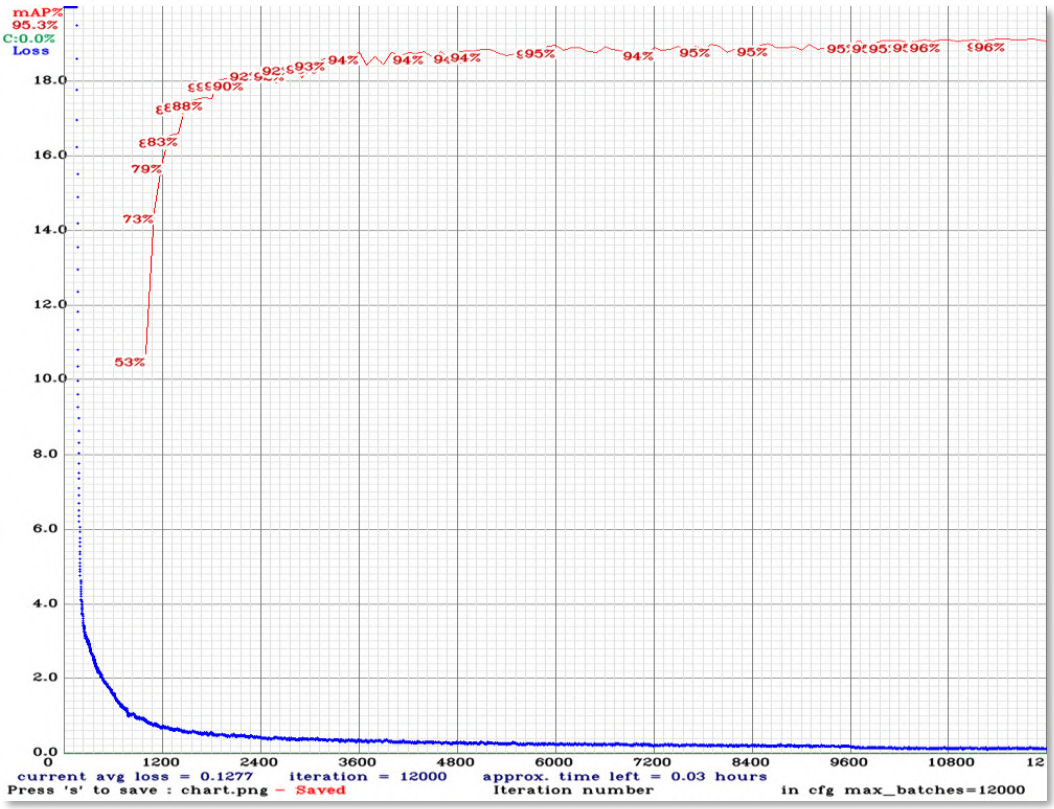


Figure 45: Stage 1. Loss and mAP during training for YOLOv4-tiny
Source: Self-made

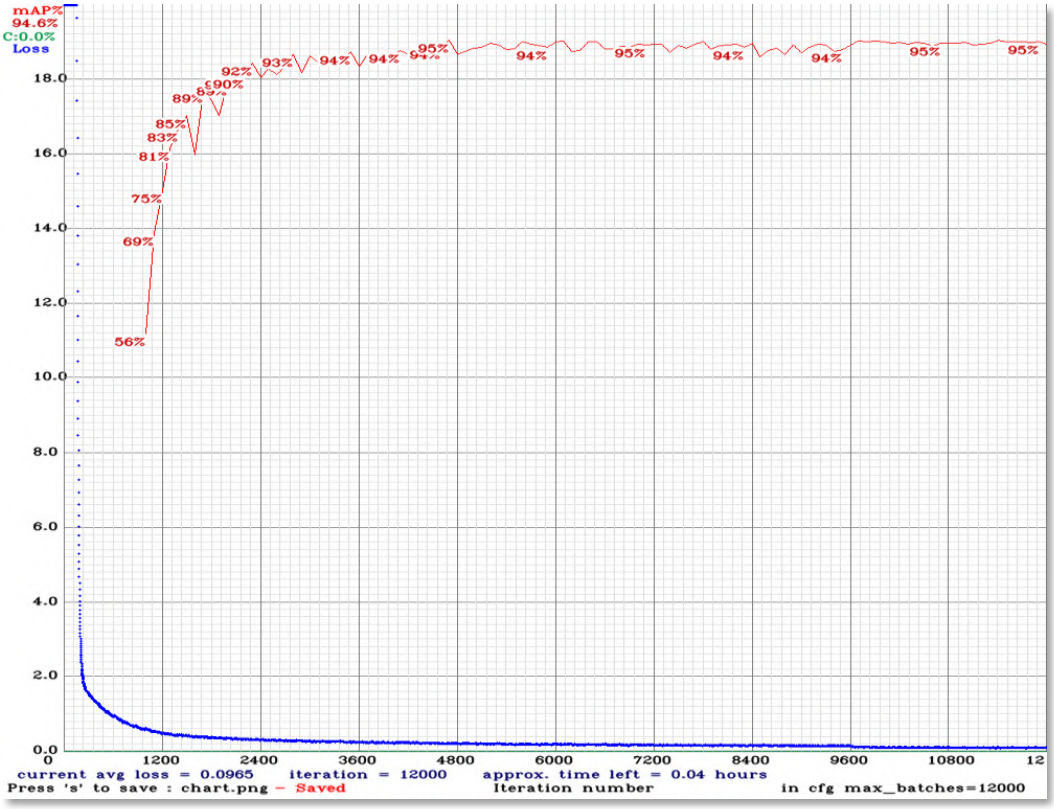


Figure 46: Stage 1. Loss and mAP during training for YOLOv4-tiny-3l
Source: Self-made

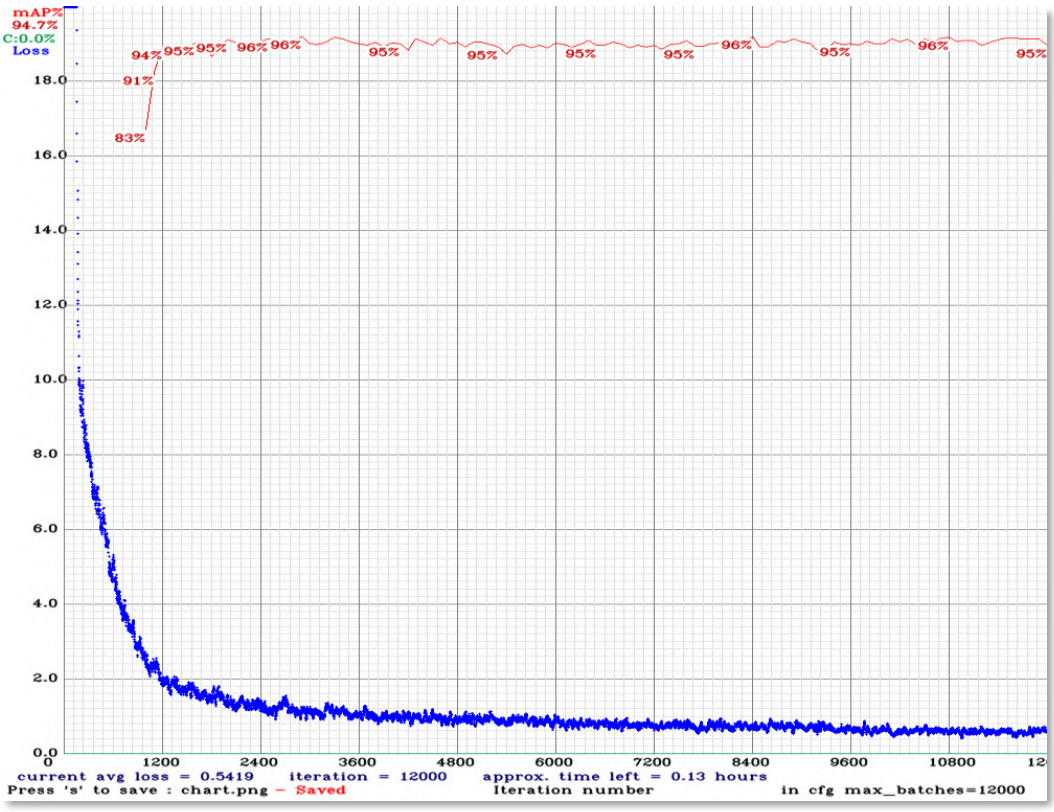


Figure 47: Stage 1. Loss and mAP during training for YOLOv4
Source: Self-made

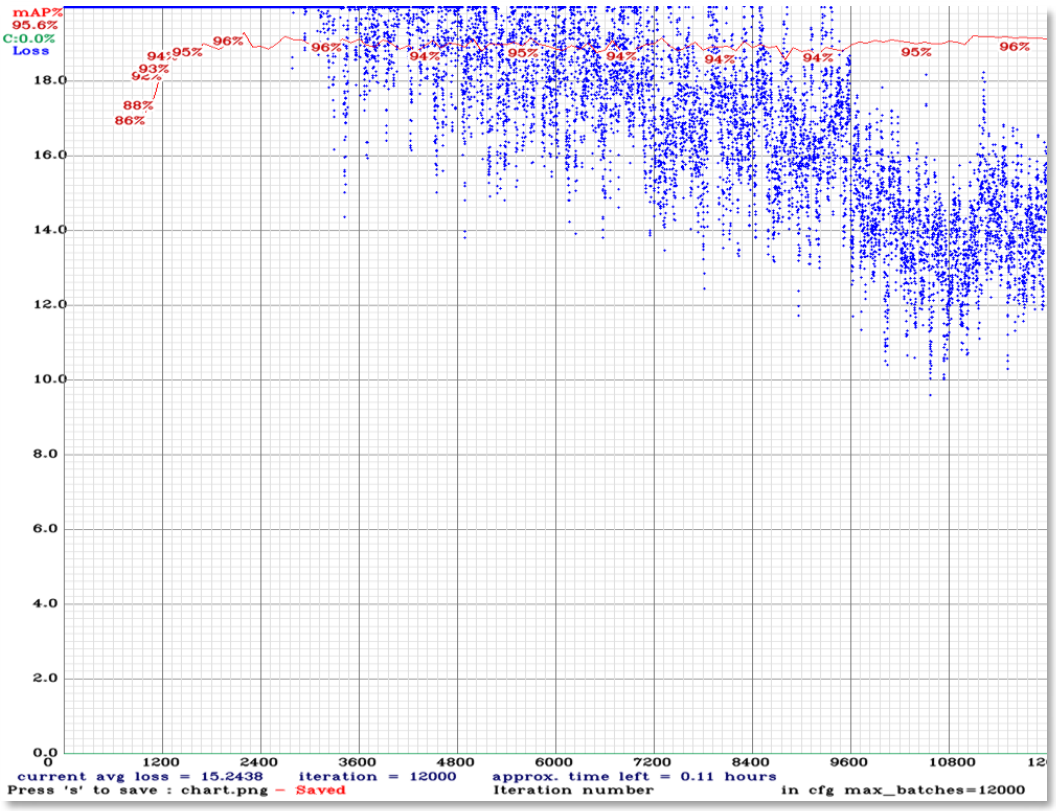


Figure 48: Stage 1. Loss and mAP during training for YOLOv4-CSP
Source: Self-made

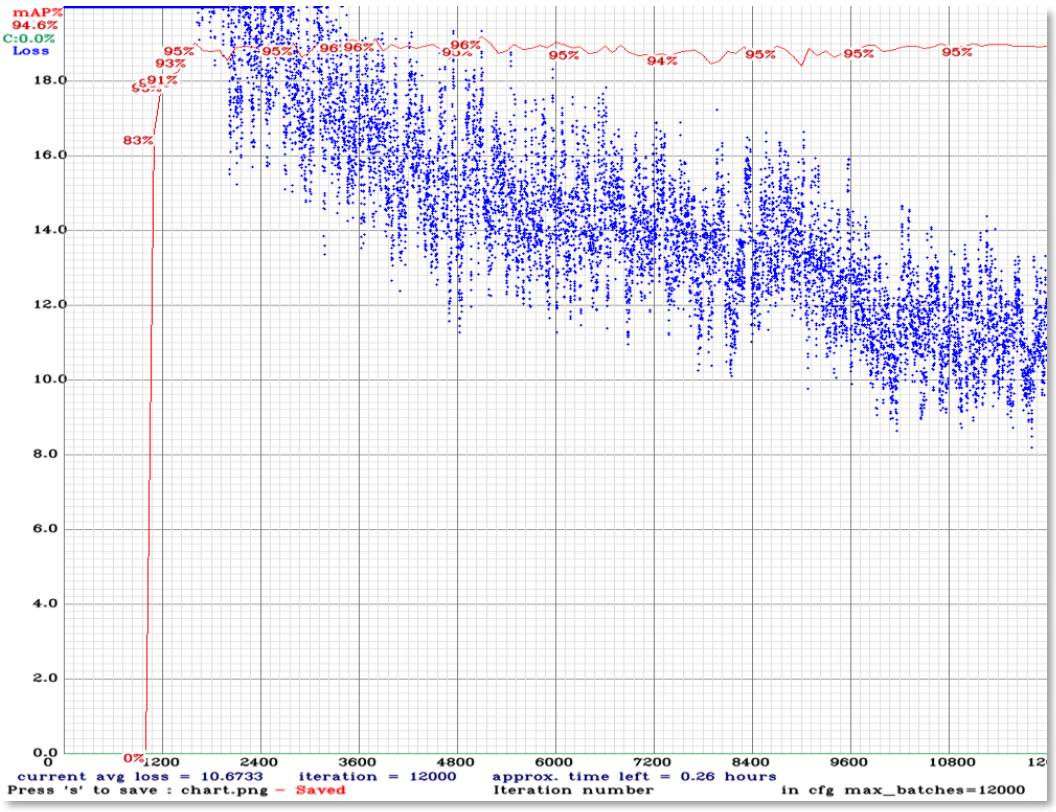


Figure 49: Stage 1. Loss and mAP during training for YOLOv4x-mish
Source: Self-made

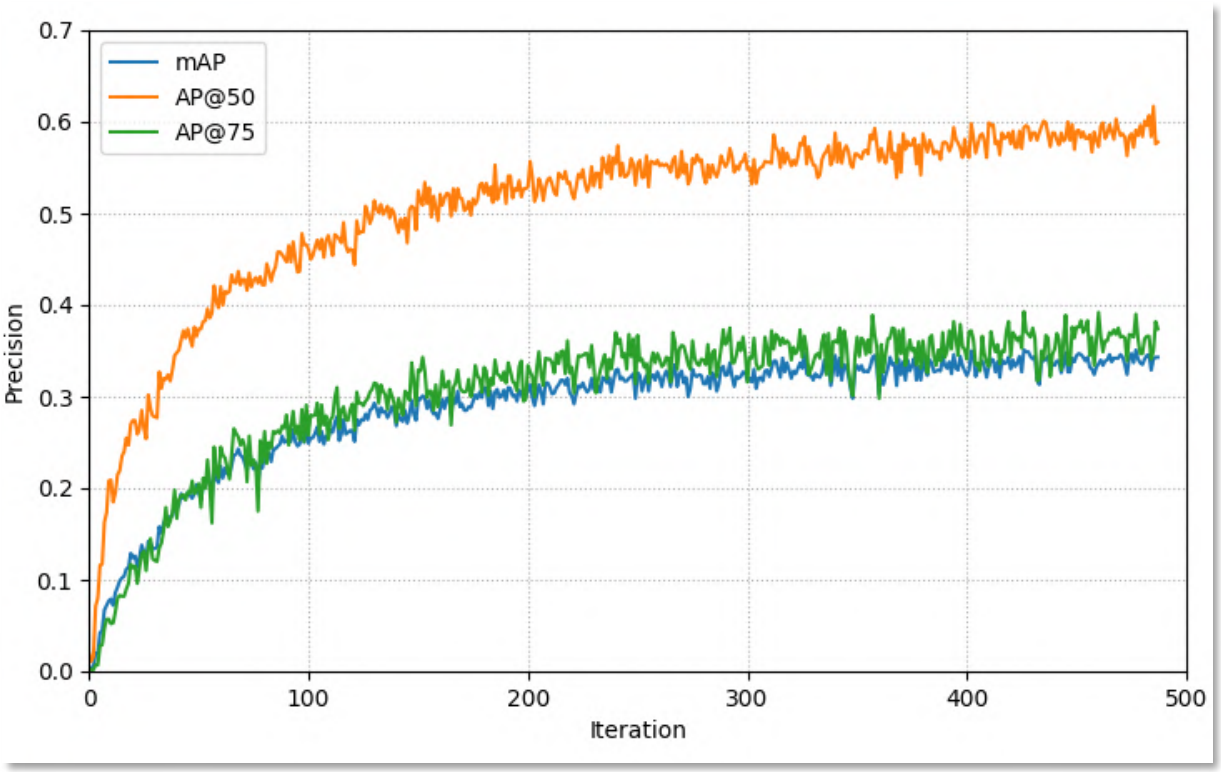


Figure 50: Stage 1. AP metrics during training for EfficientDet-D0
Source: Self-made

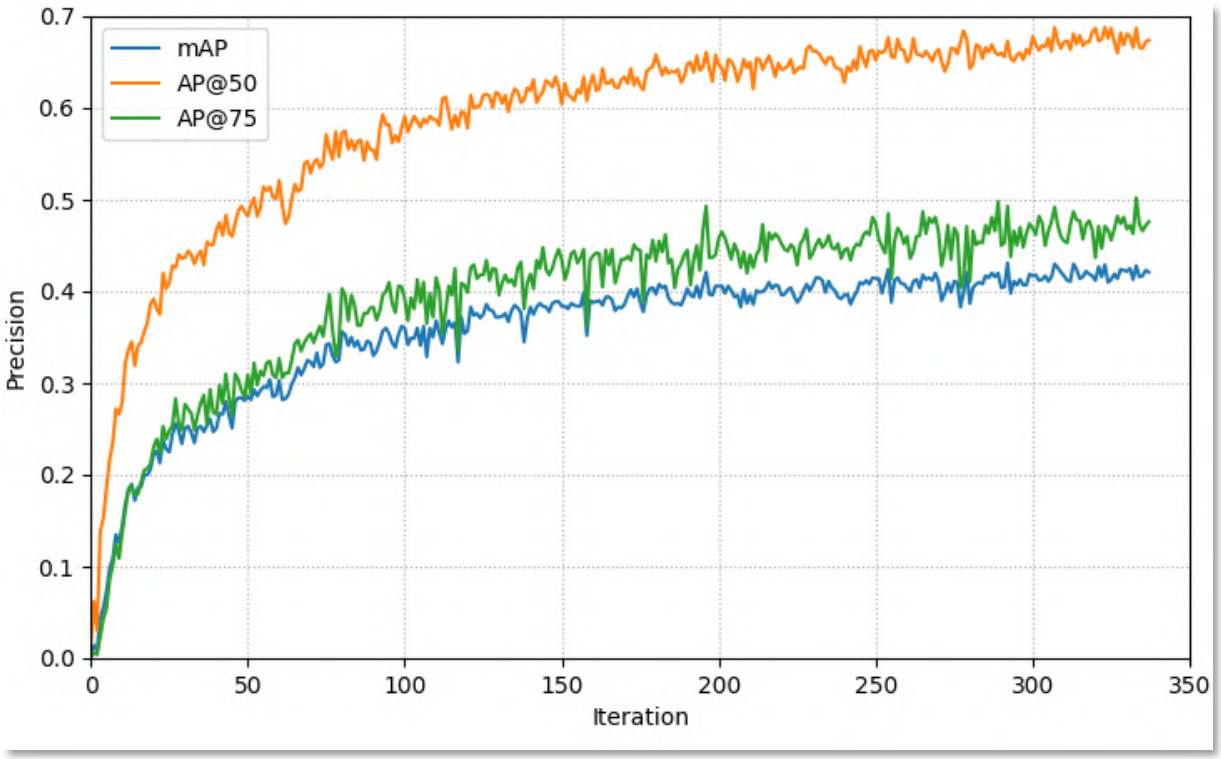


Figure 51: Stage 1. AP metrics during training for EfficientDet-D1
Source: Self-made

While training losses cannot be compared for different models (since generally each of these models uses a different one), a common intuitive magnitude is the validation mAP. From the training plots, the following observations on the evaluated models are pointed out:

- All the Darknet models quickly saturate on a stationary mAP value and don't appear to require a full-time training, though their loss slowly decreases until the very end.
- The smaller models YOLOv4-tiny and YOLOv4-tiny-3l take more iterations to converge to the top mAP compared to other YOLOs, though their training time is much smaller.
- EfficientDet models show very poor validation performances with respect to YOLOs.

Evaluation of the best weights for each of the models is performed on the PPU-6 test split. Testing is always performed by resizing the PPU-6 images to each model's native resolution (Darknet automatically applied this resize when training, but not when testing [55]). AP metrics are shown graphically in Figure 52. The most relevant observations being:

- Darknet models perform better the smaller the number of classes.
- YOLOv4-tiny outperforms all other models in AP@50.
- YOLOv4-tiny-3l significantly lags behind YOLOv4-tiny on mAP.
- YOLOv4 slightly outperforms YOLOv4-CSP and YOLOv4x-mish.

Compared to the MS COCO detection results, where the best contender was YOLOv4-mish with a mAP slightly below 50% (see Figure 24), the YOLO networks have greatly improved their performance. This is not so much true for the EfficientDet models, which have stalled into mAPs around $\sim 30\%$. YOLOv4-tiny is not only the best contender, but has gone from a 21% mAP on MS COCO to a 71% on PPU-6. These massive boosts on accuracy are thought to be due to the fact that only 6 classes are to be detected, versus 80 for the MS COCO benchmarks. This doesn't hold for EfficientDet models, which have shown not only test mAP, but also top validation $mAPs$, similar to the low precisions obtained for the MS COCO data (30% & 40%). On the test set, their performance has been very similar and around the 30% for both EfficientDet-D0 and EfficientDet-D1.

Model speeds were found within expectations, being measured with the machine with specs shown in Section 4.2.4. The fastest model is YOLOv4-tiny ($197\ FPS$), its YOLOv4-tiny-3l ($182\ FPS$) variant slightly slower, both way ahead of the standard-size models. From these, the YOLOv4 ($28\ FPS$) and YOLOv4-CSP ($26\ FPS$) are close to each other, while YOLOv4x-mish ($9\ FPS$) lags noticeably behind. EfficientDet-D0 ($51\ FPS$) and EfficientDet-D1 ($23\ FPS$) are notably slow considering their size is comparable to those of the tiny networks.

The most striking observation of this stage is that YOLOv4-tiny outperforms all other models, both tiny and regular-sized in $AP@50$, $AP@75$ and mAP. Because YOLOv4 models use very strong data augmentation techniques, this raises the suspicion that bigger models might be overfitting on the synthetic graphics of the artificial images in the dataset, since bigger networks are more prone to overfitting. On the other hand, YOLOv4-tiny-3l, which has the same architecture as YOLOv4-tiny with one additional YOLO layer, should have a slightly better detection accuracy than the latter, not true for any of the AP metrics.

Benchmarking on the PPU-6 dataset

Model	Platform	Train size	mAP	AP@0.5	AP@0.75	FPS
yolov4-tiny	Darknet	416x416	**.712**	**.946**	**.875**	**197**
yolov4-tiny-3l	Darknet	416x416	.516	.912	.532	182
yolov4	Darknet	416x416	.619	.931	.735	28
yolov4-csp	Darknet	512x512	.544	.907	.602	26
yolov4x-mish	Darknet	640x640	.615	.922	.751	9
efficientdet-d0	google-automl	512x512	.304	.512	.350	51
efficientdet-d1	google-automl	640x640	.319	.551	.345	23

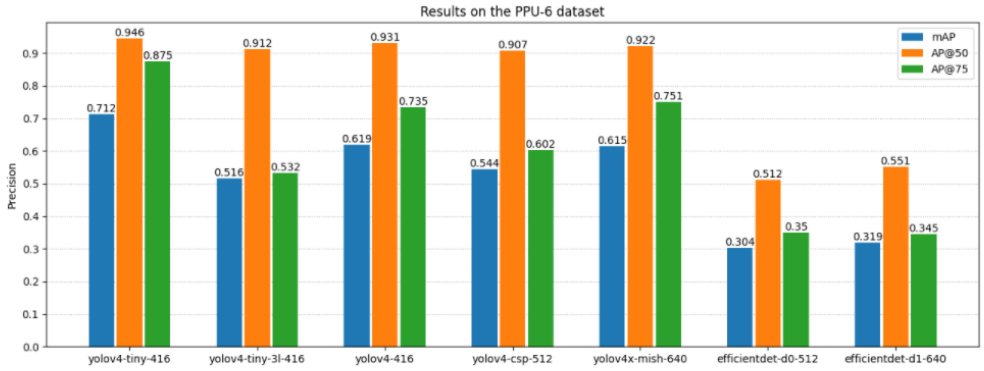


Figure 52: Stage 1 benchmarking results on the PPU-6 dataset
Source: Self-made

The main observations so far and their impact on design choices for the next benchmarking stage can be summarized as the following bullet points:

- YOLOs blow EfficientDets on few-class detections
  → *discard efficientdets from further experiments*
- YOLOv4x-mish does not show a performance that justifies its low framerate
  → *discard yolov4x-mish from further experiments*
- Tiny models outperform regular sizes (overfitting suspicion)
  → *retrain on only real data*
- YOLOv4-tiny outperforms YOLOv4-tiny-3l
  → *deeper experiments with anchor sizes / layer attribution*

4.3.3 Stage 2. Default training on PP-6

The purpose of this stage is to check the synthetic overfitting suspicion of bigger models so far. For this, training is repeated on the PP-6 dataset, equal to the PPU-6 only without synthetic training samples. Since test data is the same, since no synthetic pictures are present in the test split, performance results shown here are comparable to those of the previous stage, and the inclusion/exclusion of synthetic data can be considered a Bag of Freebies feature. Figure 53, Figure 54, Figure 55 and Figure 56 show the training metrics for each of the models in this stage.

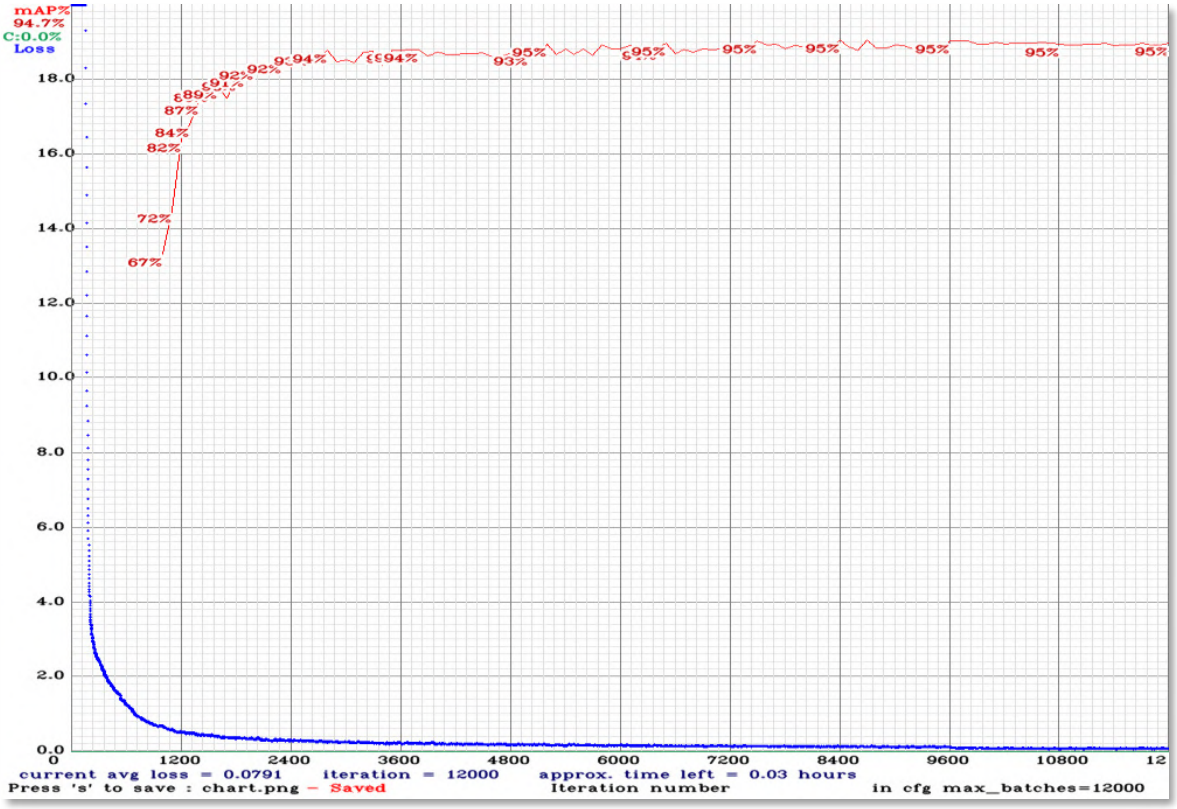


Figure 53: Stage 2. Loss and mAP during training for YOLOv4-tiny
Source: Self-made

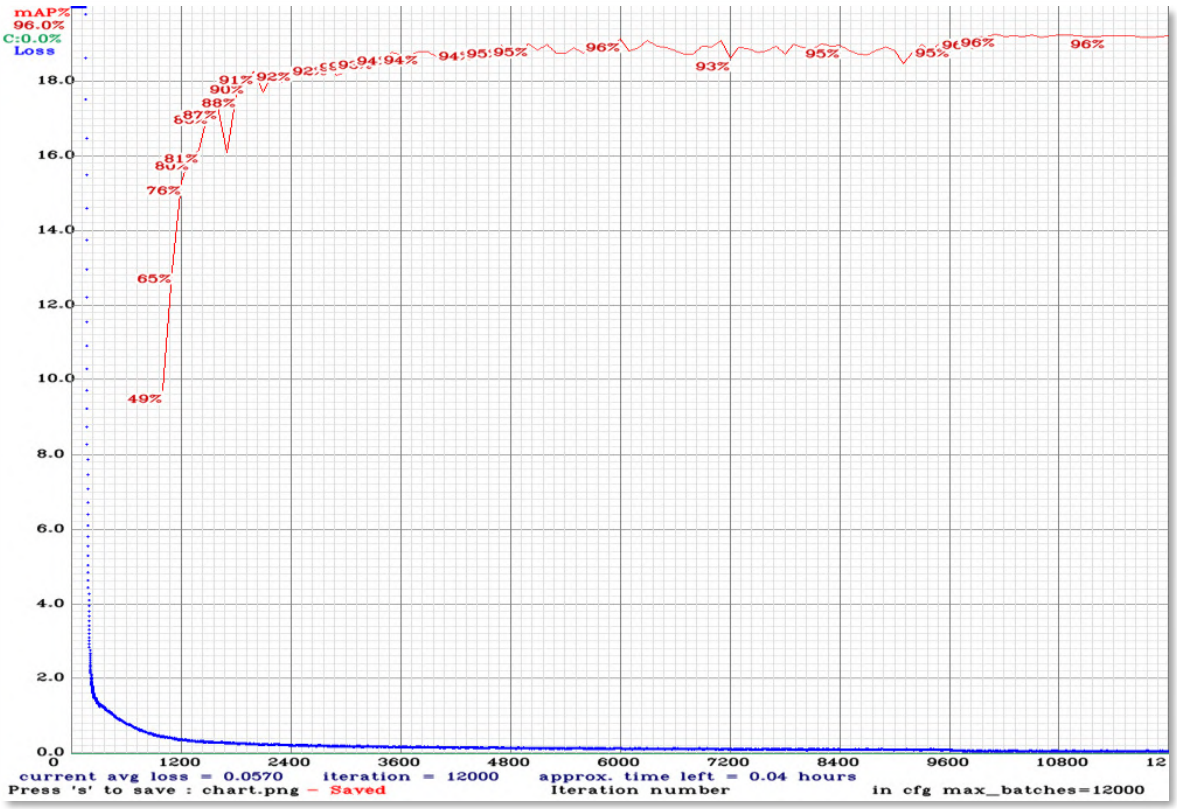


Figure 54: Stage 2. Loss and mAP during training for YOLOv4-tiny-3l
Source: Self-made

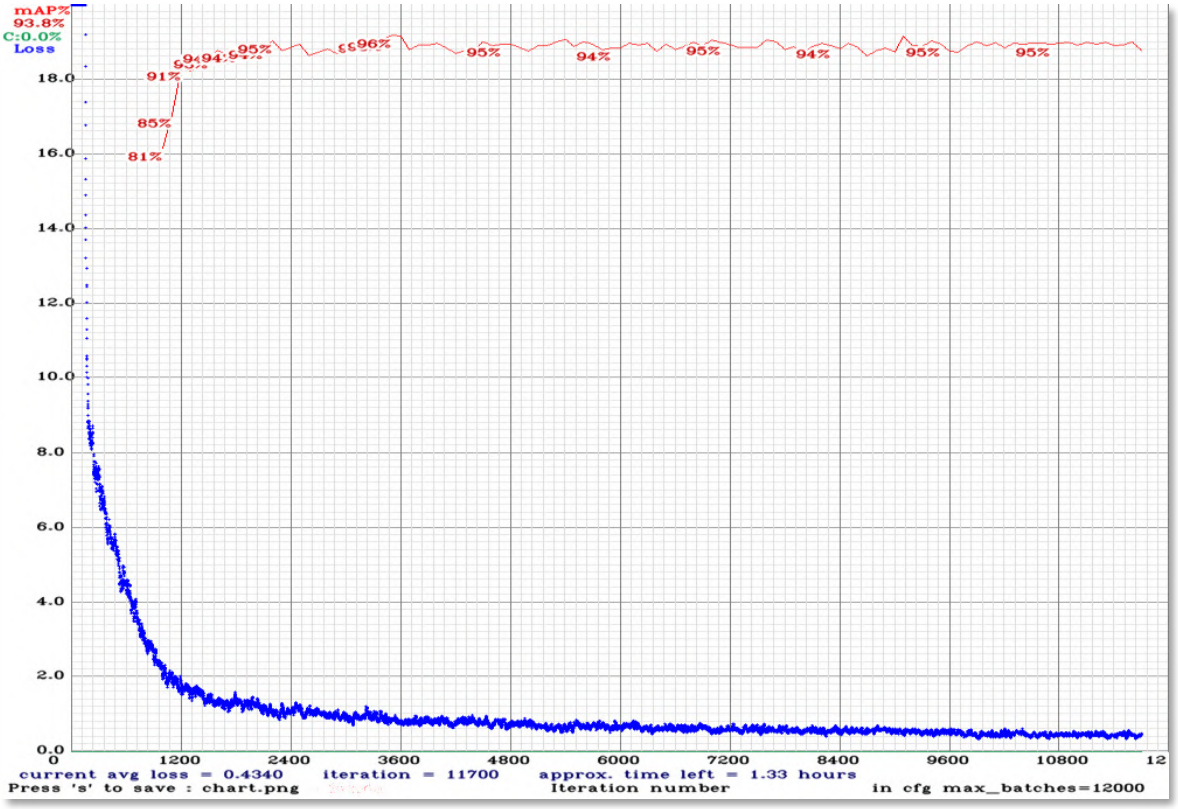


Figure 55: Stage 2. Loss and mAP during training for YOLOv4
Source: Self-made

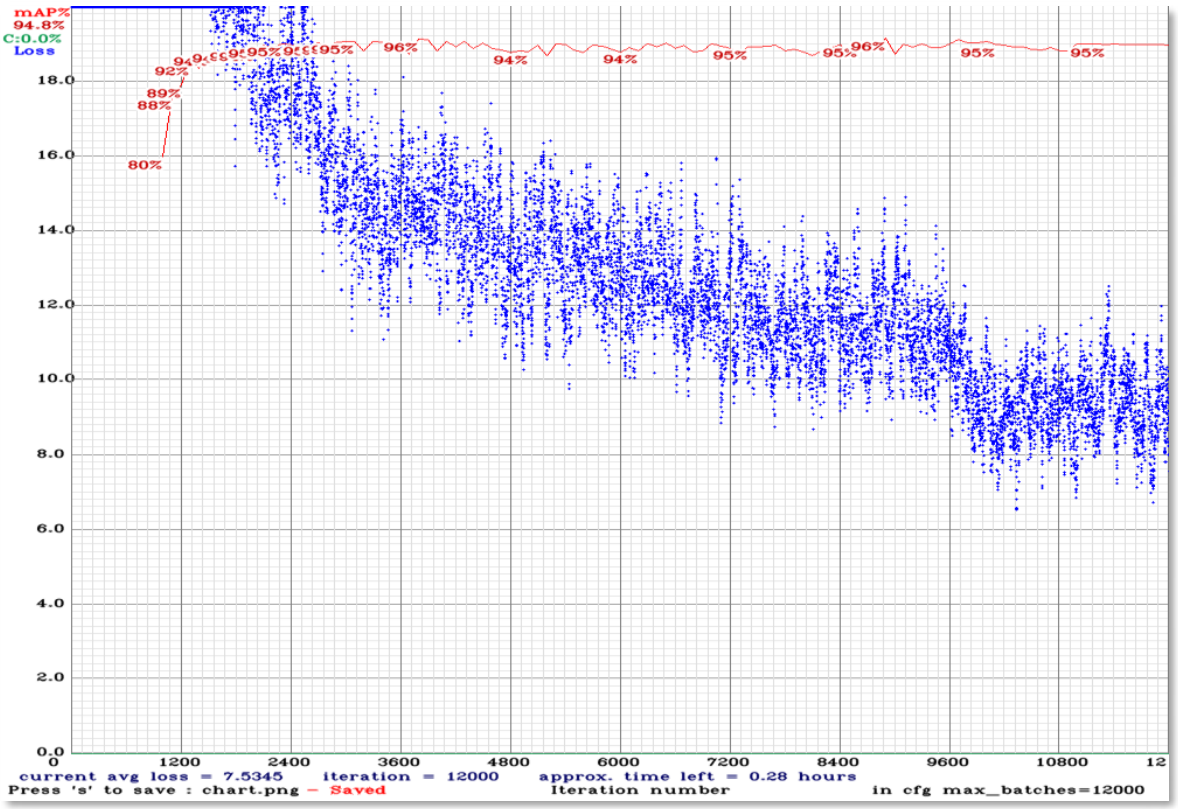


Figure 56: Stage 2. Loss and mAP during training for YOLOv4-CSP
Source: Self-made

Much the same way as in the previous experiments, all models are trained on square images of size equal to that of the network, by using the built-in resizer within Darknet. As stated in Section 4.2.4, this stage took place in a local machine so the training times are relevant (because the hardware is known). Table 4 shows the time it took for each network to produce the best validation accuracy and to complete its training.

Table 4: Training times of Darknet models on GTX 1060-mobile

Model	**Time to best val mAP [hh:mm]**	**Full train time [hh:mm]**
yolov4-tiny	02:55	04:10
yolov4-tiny-3l	03:47	04:30
yolov4	11:14	40:44
yolov4-csp	24:50	33:32

For testing, three different settings were used, altering the network resolution (note that the networks were only trained once, for their canonical resolution, as shown in Table 4). Let's first define the following parameters towards this work:

- $TrainSize$: Size of the training images that were used for training a model.
- $NetworkSize$: Network resolution, stated in its architecture cfg file (YOLO nets).
- $TestSize$: Size of the test images that are inferred during testing.

And thus, the three following cases are studied by modifiying the previous variables in all models, each under a new substage:

2.1) $[TrainSize, NetworkSize, TestSize] = [TrainSize,\ TrainSize,\ TrainSize]$

Whose purpose is to evaluate the hypotheses that synthetic images might be corrupting the bigger models. From the evaluation results, shown in Figure 57, the following is observed when comparing them to those of stage 1:

- $AP@50$: Very slight variations have taken place, though in a random manner: some models are slightly better, some are worse.
- $AP@75\ and\ mAP$: Significant variations have happened in the different models, arbitrarily for better or worse, bigger for tiny models.

These results lead to the thought that synthetic data has not played a significant role on the detection of items. The high and consistent $AP@50$ indicates that items are still being properly detected. The varying $AP@75$ and mAP reveals that the usage of synthetic images has a relevant impact on bounding box accuracy, which makes sense since the generated samples have pixel-grade precision bboxes while the rest of the data does not. Under these, it is concluded that no overfitting is damaging the bigger models; tiny ones just have enough optimization power for 6-class detection.

Benchmarking on the PP-6 dataset (PPU-6 without SynthDet samples).

Test size equals train size

Model	Platform	Train size	Test size	mAP	AP@0.5	AP@0.75	FPS
yolov4-tiny	Darknet	416x416	416x416	.576	**.943**	.628	**184**
yolov4-tiny-3l	Darknet	416x416	416x416	**.657**	.935	**.798**	183
yolov4	Darknet	416x416	416x416	.559	.915	.644	28
yolov4-csp	Darknet	512x512	512x512	.619	.915	.763	26

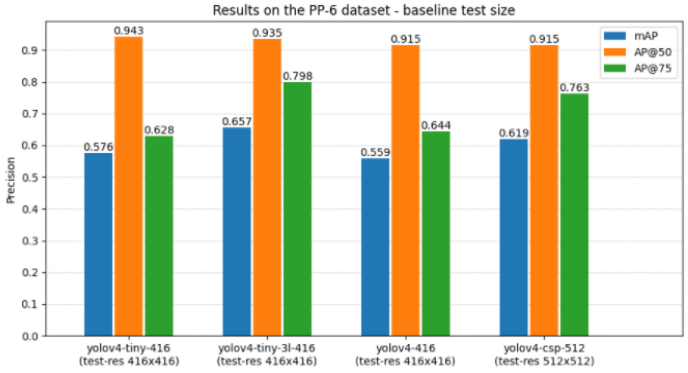


Figure 57: Stage 2.1 benchmarking results on the PP-6 dataset
Source: Self-made

2.2) $[TrainSize, NetworkSize, TestSize] = [TrainSize,\ TrainSize,\ 640x640]$

Whose purpose is to evaluate if the accuracy of a model increases if it is fed bigger images than its native resolution, as suggested in [34]. For this, evaluation is performed as usual, only resizing the test set to a new size of $640x640$ for all models. Comparing with the results of stage 2.1, it is found that this might be true in some cases, however the differences are not very noticeable, at least for a 6-class detection. Numerical results are shown in Figure 58.

2.3) $[TrainSize, NetworkSize, TestSize] = [TrainSize,\ 640x640,\ \ 640x640]$

In this stage, the size of the networks is modified to further explore if a trained then expanded model can provide better detection accuracy. As seen in the results shown in Figure 59, all networks have seen their performance decreased but YOLOv4-CSP, whose metrics have remained more or less consistent. Since model sizes have been made bigger, the inference speeds in this experiment are noticeably lower.

While YOLOv4-tiny has been seen to perform worse than YOLOv4-tiny-3l in stage 2, as expected yet opposed to the initial controversial result of stage 1, the variability of the results from one experiment to the other is still a matter of attention. The need for a deeper revision of these two models suggests performing a new experiment to further diagnose the current benchmarking.

Benchmarking on the PP-6 dataset (PPU-6 without SynthDet samples).

Fixed test size - canon network size

Model	Platform	Train size	Network size	Test size	mAP	AP@0.5	AP@0.75	FPS
yolov4-tiny	Darknet	416x416	416x416	640x640	.574	**.946**	.628	**194**
yolov4-tiny-3l	Darknet	416x416	416x416	640x640	**.668**	.937	**.818**	173
yolov4	Darknet	416x416	416x416	640x640	.568	.915	.648	28
yolov4-csp	Darknet	512x512	512x512	640x640	.619	.911	.711	26

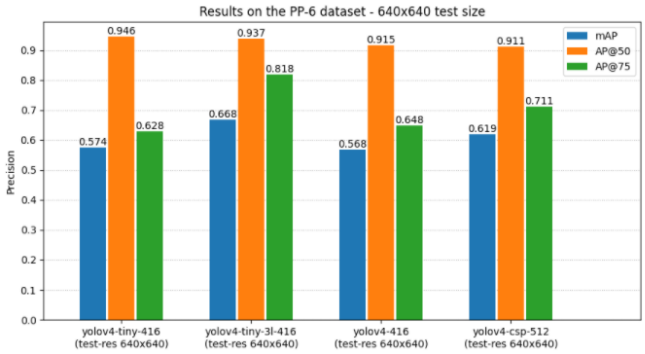


Figure 58: Stage 2.2 benchmarking results on the PP-6 dataset
Source: Self-made

Benchmarking on the PP-6 dataset (PPU-6 without SynthDet samples).

Fixed test size - higher network size

Model	Platform	Train size	Network size	Test size	mAP	AP@0.5	AP@0.75	FPS
yolov4-tiny	Darknet	416x416	640x640	640x640	.503	.897	.519	102
yolov4-tiny-3l	Darknet	416x416	640x640	640x640	.592	.896	.722	97
yolov4	Darknet	416x416	640x640	640x640	.499	.874	.513	13
yolov4-csp	Darknet	512x512	640x640	640x640	.601	.914	.718	16

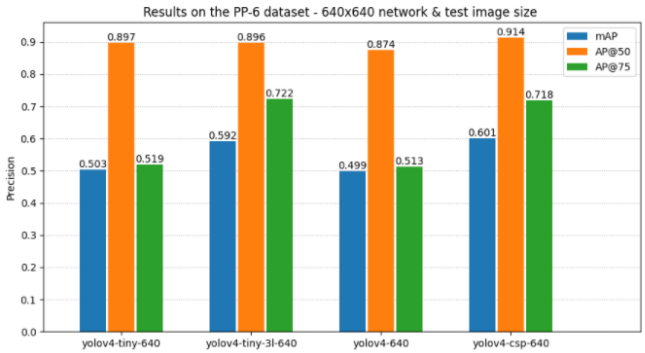


Figure 59: Stage 2.3 benchmarking results on the PP-6 dataset
Source: Self-made

4.3.4 Stage 3. Customized anchors on PP-6

So far, the models offering the best speed-accuracy trade-off are the tiny ones. While all models show very similar $AP@50$, bounding box accuracy (mAP & $AP@75$) has been seen to change significantly for each experiment. The main concern so far lies between the models YOLOv4-tiny and YOLOv4-tiny-3l, which are supposed to be equal only the latter having an extra YOLO layer. This extra layer should be granting it a somewhat improved detection accuracy, which should be especially reflected on mAP & $AP@75$. This has not been true so far, since there has been a random variability between these metrics of both models.

YOLO layers take as input a series of bounding box priors, a better selection of which can lead to better detection results. These priors are usually taken by clustering the annotation sizes of some split of the data [46], and contain information about the most common sizes and aspect ratios of the instances in the dataset. In all previous experiments of this work, they were left as default, belonging to the MS COCO dataset. Given that it is such a big collection of items, MS COCO anchor boxes can be considered representative of the general case, hence the reason they hadn't been a matter of attention so far.

The purpose of this third stage is to retrain YOLOv4-tiny and YOLv4-tiny-3l on the PP-6 dataset using recomputed anchors from the very same data. Synthetic data is excluded because the sizes and aspect ratios of the 3D models are not expected to be representative of the available real data. These new anchors can be computed with Darknet in a straightforward manner and mapped to YOLO layers on the *cfg* files.

Numerical results after training (train metrics plotted in Figure 61 & Figure 62) are shown in Figure 60. From these, it can be seen that, even though YOLOv4-tiny outperforms YOLOv4-tiny-3l, mAP & $AP@75$ metrics for both have substantially improved and do make sense now.

Benchmarking tiny models on the (PPU-6 dataset without SynthDet samples). Adjusted anchors to fit our training data.

Model	Platform	Network size	mAP	AP@0.5	AP@0.75
yolov4-tiny	Darknet	416x416	.692	.942	.829
yolov4-tiny-3l	Darknet	416x416	.658	.929	.817

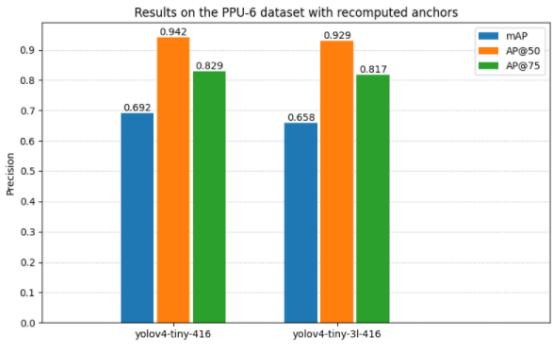


Figure 60: Stage 3 benchmarking results on the PPU-6 dataset
Source: Self-made

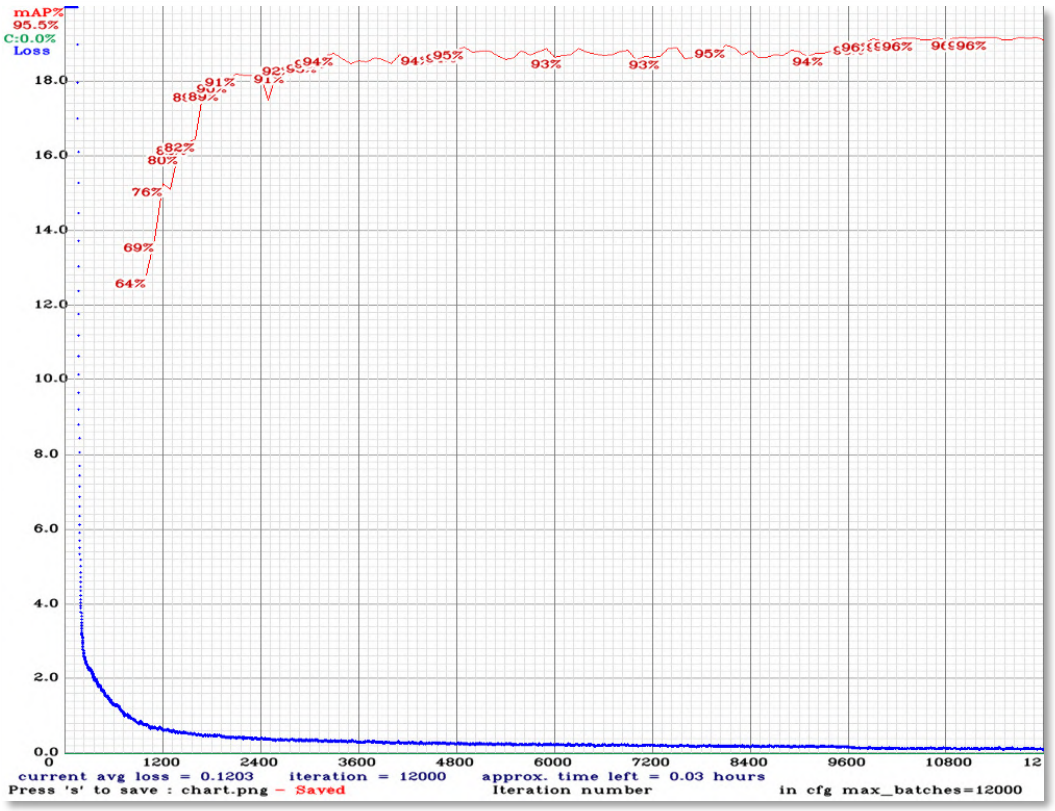


Figure 61: Stage 3. Loss and mAP during training for YOLOv4-tiny
Source: Self-made

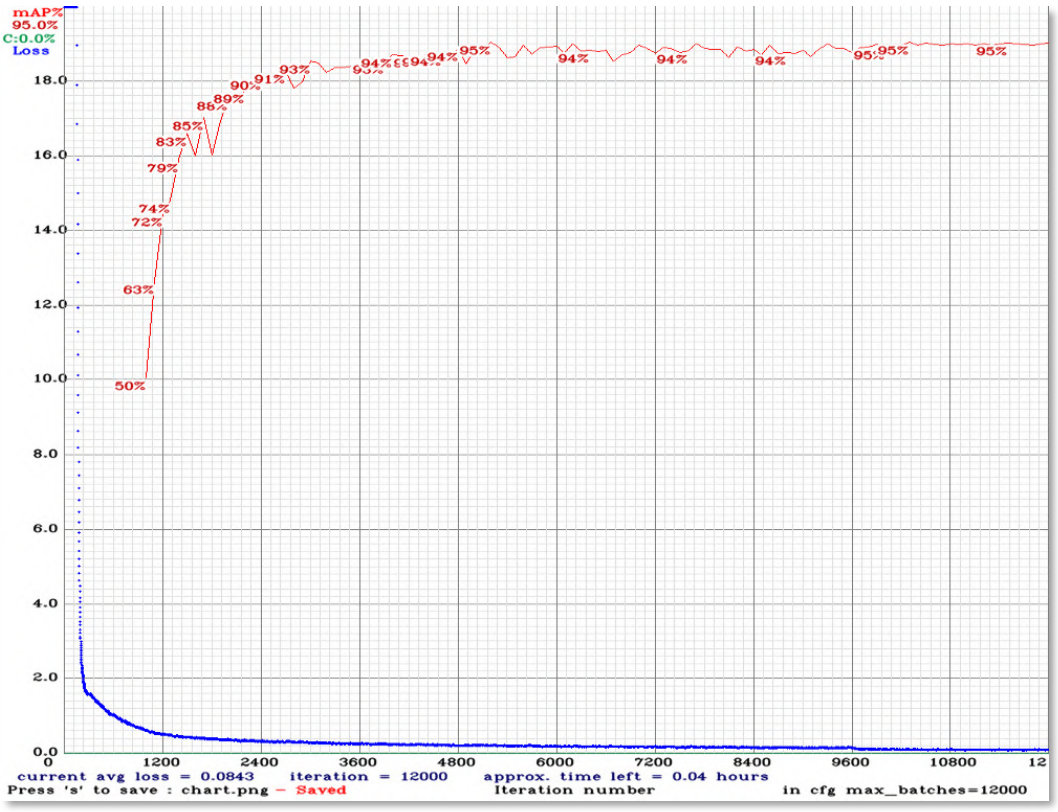


Figure 62: Stage 3. Loss and mAP during training for YOLOv4-tiny-3l
Source: Self-made

4.4 Results

4.4.1 Benchmarking on the PPU-6 dataset

Table 5 shows a full summary of all of the stage-specific models discussed in the previous sections, benchmarking their *AP* metrics and inference speed. The total GPU time for setting up and performing these tests was of $195h$. The features in the result table stand for the discussed extra features of each experiment:

- *SD* : Use of synthetic data, always examples produced with SynthDet.
- *ITS* : Increased test size, meaning that bigger images are used at testing
- *INS* : Increased network size, meaning that the net resolution is manually increased.
- *RA* : Recomputed anchors, meaning that data-specific bbox priors are used.

Table 5: Object detector benchmarking on the PPU-6 dataset

Model	Trained Size	SD	ITS	INS	RA	AP50 %	AP75 %	mAP %	FPS 1060m
yolov4-tiny	416					94.3	62.8	57.6	184
yolov4-tiny	416	✓				**94.6**	**87.5**	**71.2**	197
yolov4-tiny	416		✓			**94.6**	62.8	57.4	194
yolov4-tiny	416		✓	✓		89.7	51.9	50.3	102
yolov4-tiny	416				✓	94.2	82.9	69.2	198
yolov4-tiny-3l	416					93.5	79.8	65.7	183
yolov4-tiny-3l	416	✓				91.2	53.2	51.6	182
yolov4-tiny-3l	416		✓			**93.7**	**81.8**	**66.8**	173
yolov4-tiny-3l	416		✓	✓		89.6	72.2	59.2	97
yolov4-tiny-3l	416				✓	92.9	81.7	65.8	184
yolov4	416					91.5	64.4	55.9	28
yolov4	416	✓				**93.1**	**73.5**	**61.9**	28
yolov4	416		✓			91.5	64.8	56.8	28
yolov4	416		✓	✓		87.4	51.3	49.9	13
yolov4-csp	512					**91.5**	**76.3**	**61.9**	26
yolov4-csp	512	✓				90.7	60.2	54.4	26
yolov4-csp	512		✓			91.1	71.1	**61.9**	26
yolov4-csp	512		✓	✓		91.4	71.8	60.1	16
yolov4x-mish	640	✓				92.2	75.1	61.5	9
efficientdet-d0	512	✓				51.2	35	30.4	51
efficientdet-d1	640	✓				55.1	34.5	31.9	23

4.4.2 Discussion

Compared to the known performance of the models in the 80-class MS COCO, it appears that YOLO models perform better on smaller classification tasks, reaching values around ~60% mAP, while the EfficientDets have stalled around ~30% mAP. This can be clearly seen from Figure 63, which showcases a small sample of detections made by all stage 1 models. Regular size-YOLO models are observed to have tendency for proposing additional bounding boxes (false positives), much like EfficientDets when items appear crooked or bent in shape. Inference speed results also suggest that YOLO models are inherently better for this setting since it is not worth to use a bigger EfficientDet model to try to boost AP given the current speed gap between these models and the YOLO ones. In the performed experiments, most YOLO models are seen to surpass the 90% $AP@50$, implying that YOLOs have outstanding performance at finding items, though not so great when accurately regressing the bounding boxes around them. While it is common for $AP@50$ metrics to be higher than mAP and $AP@75$ for all models, this trait is usually more prominent within the YOLO networks [25].

The best model architecture-wise for this 6-class problem was experimentally found to be the deafault stage 1 YOLOv4-tiny, according to the MS COCO metrics and its outstanding inference speed. A showcase of this model's detections can be seen in Figure 64. The fact that this is the best model is found controversial since it:

- Does not include custom anchor box sizes,
- Uses a mix of training data,
- Belongs to the small-size set of models.

Custom prior box sizing is studied in stage 3. Defining data-specific custom anchor boxes should have a stronger impact on mAP and $AP@75$ than on $AP@50$, since these are linked to regression accuracy. Generally, custom anchor boxes should improve the performance of the model: in stage 3, this is clearly seen for the YOLOv4-tiny though not for YOLOv4-tiny-3l, which barely improves. The difference between these two models is again closely related to the concept of anchor boxes, the latter expected to be better at bbox regression out-of-the-box, so it is possible for the default YOLOv4-tiny-3l to be very close to its peak performance already. These ideas hold for real data, though they miss the point when dealing with rigid synthetic samples since they are not representative of real item aspect ratios, so custom anchor boxes were only addressed for real training data. In any case, defining custom anchor boxes did not produce a model outperforming the Stage 1 YOLOv4-tiny according to the metrics of choice. A detection showcase by the custom-anchor stage 3 models is shown in Figure 65.

Regarding synthetic data, including artificial example pictures in the present setting has not been as impactful as expected towards reaching higher detection accuracy. While it is true that the best metrics for YOLOv4-tiny were found by training on synthetic examples, the performance gap between models trained on mixed and real data only are notably inconsistent, some models improving but some deteriorating their metrics. However, experiments suggest that the networks can actually learn useful features from these artificial images (see Section 4.4.4). In our scenario, it is possible that the amount of real data in our models is already high enough so that synthetic data introduces random noise instead of providing more knowledge to the network.

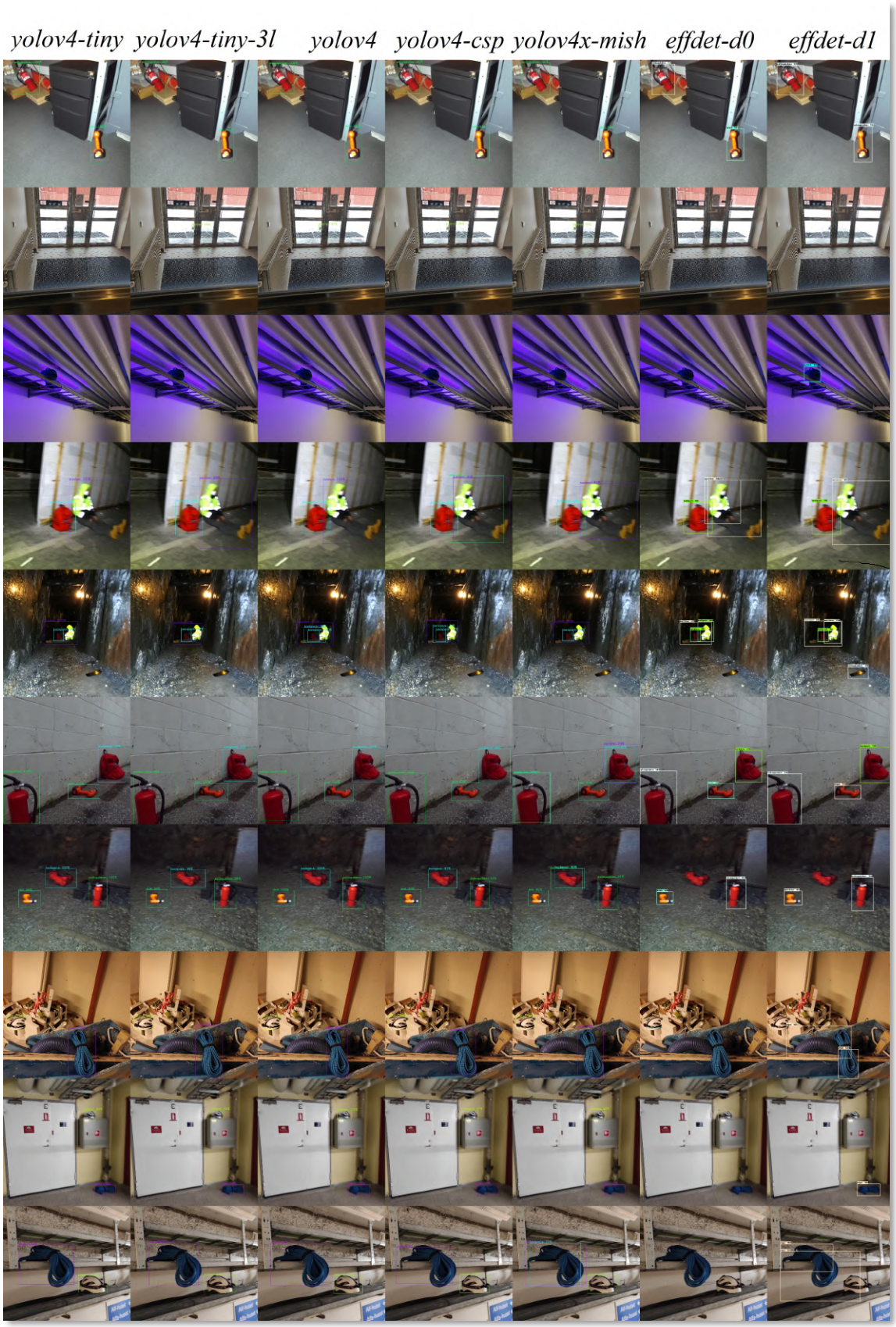


Figure 63: Example inference results of all stage 1 models
Source: Self-made

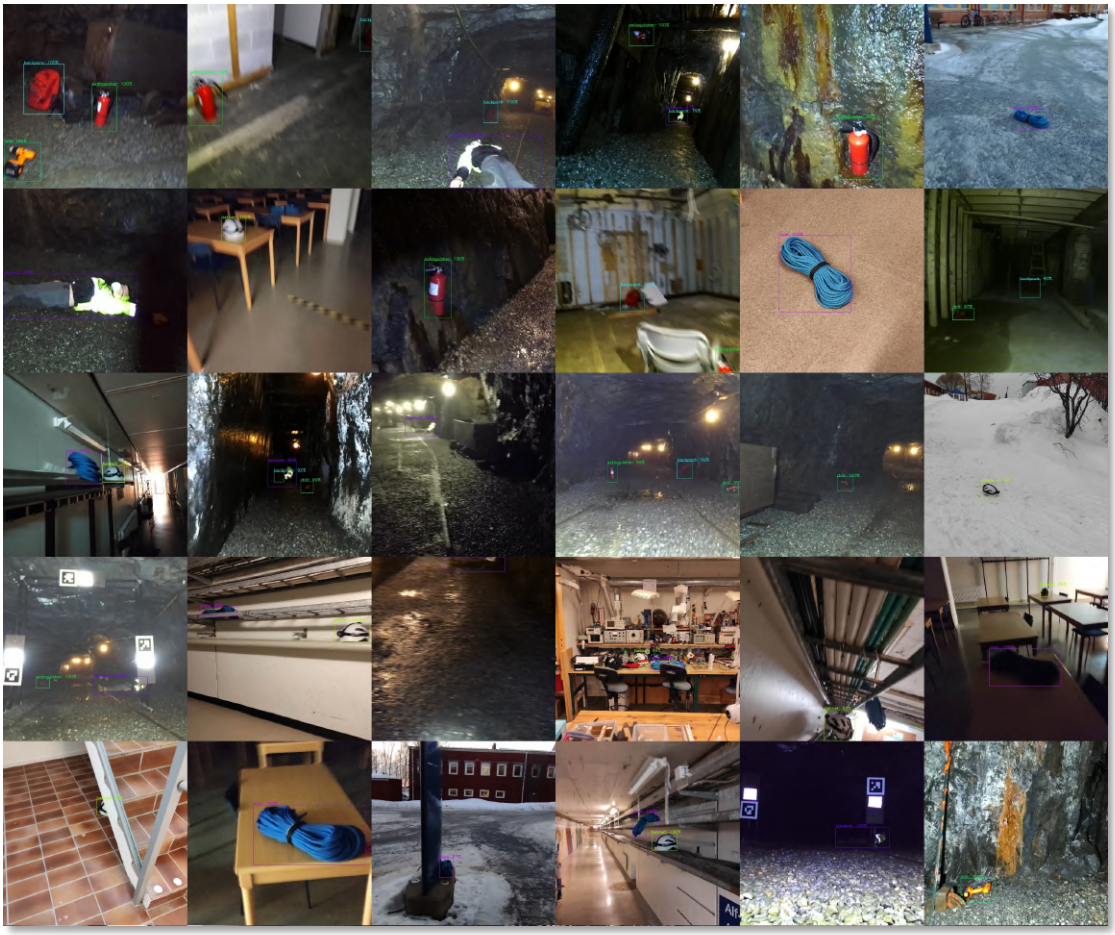


Figure 64: Example inference results of stage 1 YOLOv4-tiny
Source: Self-made

Addressing network size, smaller YOLO models have been seen to perform particularly well in this few-class object detection task, proving that smaller class problems can be solved by very small networks with mAP well over the MS COCO benchmarking (Figure 25) and outstanding $AP@50$, higher than 90%. In this work, $AP@50$ is deemed more relevant than mAP or $AP@75$ because bbox accuracy is not as critical as having true predictions with minimal false positives and false negatives. These features are more representative when evaluated at a single minimum IoU rather than after averaging APs over various thresholds. $AP@50$ performance gaps of small size have been seen between the regular and tiny models, which could partially be due to the probabilistic nature of the deep learning approach. Even if this was the case, the blatantly superior speed of tiny models greatly favours them towards choosing a preferred model, even if the bigger models still had margin for AP improvement. Additionally, while the purge of models through experimentation stages has been experimentally justified, further analysis on the topic of custom anchor boxes on bigger models could have been of interest, possibly leading to a more accurate model than the proposed YOLOv4-tiny, though only mAP and $AP@75$ would have been expected to improve. However, seen the performance of the stage 1 YOLOv4-tiny model and its outstanding inference speed of almost $200FPS$, investing an additional $70h$ of GPU training could not be justified to produce models inferring at $30FPS$, even if these proved to have better AP metrics.

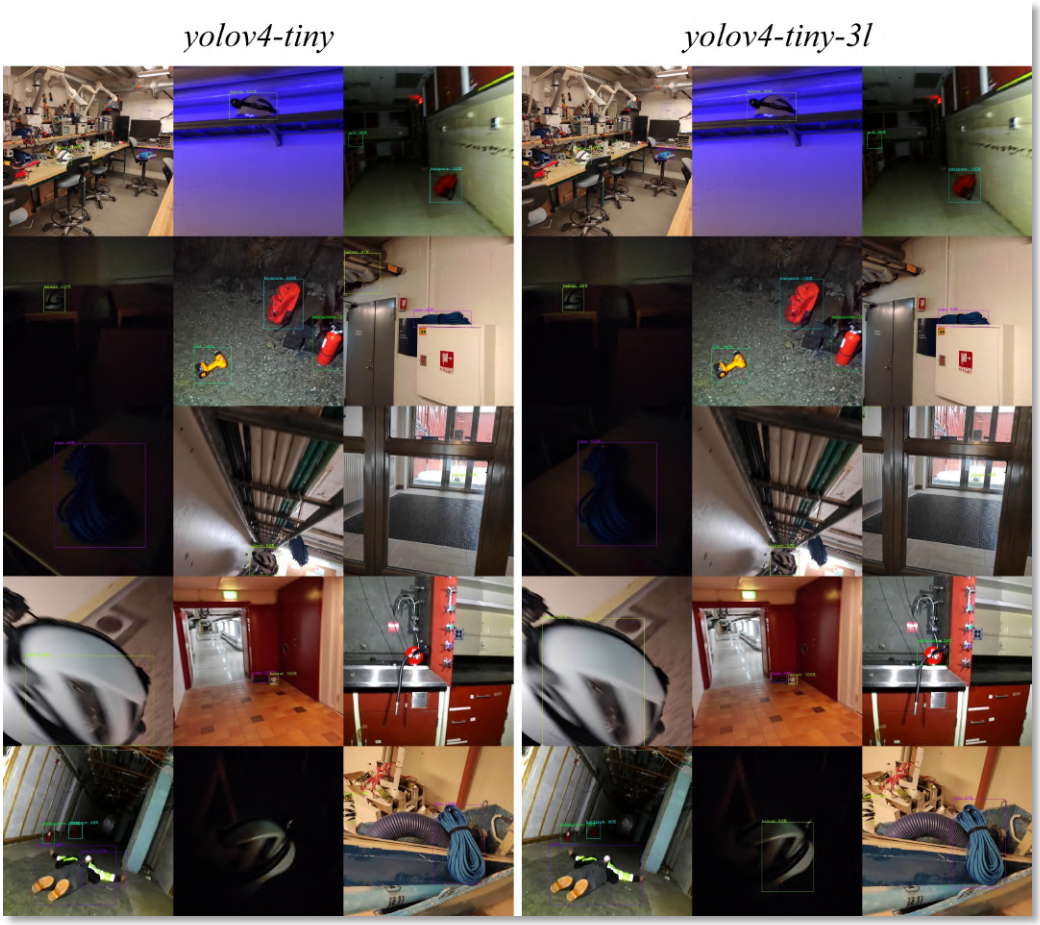


Figure 65: Example inference results of stage 3 YOLOv4-tiny and YOLOv4-tiny-3l
Source: Self-made

4.4.3 On using object detection in dark settings

The design choice of using visual item detection in underground settings, characterized by the scarcity of light, might seem a bit controversial. However, the conventional discretization of the color spectrum in values within the range $[0, 255]$ is enough to provide machines with a higher color discrimination capability than that of the usual person. Provided there is at minimum enough light so that pixel gradients exist around these features, neural networks have proven useful tools towards identifying them. Figure 66 shows some examples of object detection in dark pictures, extracted from the YOLOv4-tiny model trained at stage 1. Images are shown in pairs in which the right pictures have been modified so items become clearly visible.

4.4.4 On the representativeness of synthetic data

While previous experiments have found the benefits of using synthetic training examples ambiguous, undocumented tests have shown that models can actually learn useful features from these examples. In these, it is observed that models trained exclusively on synthetic data can identify items, however they tend to produce an unadmissible high number of false positives. Detection examples of a YOLOv4 model trained exclusively on SynthDet data are shown in Figure 67. These observations suggest that the produced photorealistic synthethic samples are representative of the items to be found, yet they do not generalize well to real backgrounds.

Figure 66: Object detection inference consistency in dark conditinos
Source: Self-made

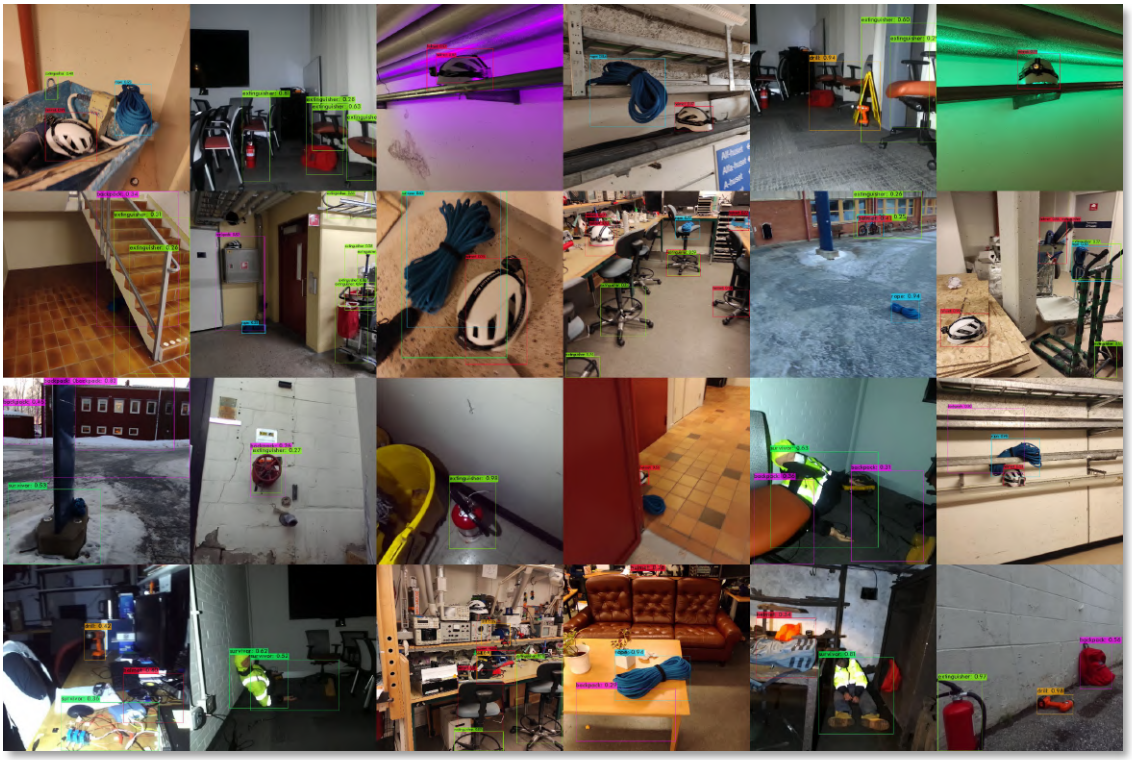


Figure 67: Inference results of a YOLOv4 trained exclusively on SynthDet examples
Source: Self-made

5 TOWARDS ITEM LOCALIZATION

5.1 Introduction

5.1.1 Requirements and approach

This section aims to introduce a series of concepts that allow 2D object detections from camera images to be converted to real world absolute coordinates, with the aims of later developing a perception item localization pipeline for the task of item search. The concepts here introduced relate to the linked yet separate areas of agent localization and item localization.

Agent localization, usually known as robot localization, implies knowing the position of an agent, typically a robot. Visually solving the localization problem for a camera usually generalizes to solving it for the robot or autonomous device, since the pose of a camera mounted on a robot with respect to the robot itself is generally known. Item localization implies detecting items with respect to the agent and framing them in a representation of the environment.

The work shown in this section attempts to solve the 2D horizontal localization problem for a camera device and items in the environment by using only computer vision and consumer degree technology. Besides these conditions, no specific targets or requisites are set a priori.

5.1.2 Experimental setup

Live feed is to be taken from the camera of an an Android device and sent over 2.4GHz or 5GHz signals (WiFi) to a local server hosted on the smartphone. The IP Webcam app [56] performs this function in a plug-and-play fashion. This live stream is captured by the host computer, where real-time processing of the images takes place. The main assets used for the development of the following stages are defined a posteriori as:

- Smartphone device : Android device with camera, namely a Redmi Note 9 Pro.
- IP Webcam app : Android app used to send phone camera feed to local server.
- GPU-ready PC : Main computing device for script hosting & running.
- Tripod : Used for static phone holding during live tests.
- Printer, paper & tape : For printout and layout of ArUco markers.
- Tape measure : For mapping ArUco locations into the enviroment model.

The IP Webcam app hosts a local web server on the smartphone device. This server is typically hosted at the direction *192.168.x.x:8085*, which corresponds to the private IP of the phone within the user WiFi network, typically randomized at server boot, plus a user-defined port. The main directions of this server are the following:

- *http://192.168.x.x:8085* : Hosts the main application GUI.
- *http://192.168.x.x:8085/video* : Hosts the raw camera video feed.
- *http://192.168.x.x:8085/sensors* : Hosts a sensor measurement GUI.
- *http://192.168.x.x:8085/sensors.json* : Hosts a raw *json* file with sensor history.

5.2 The localization problem

5.2.1 Localization in underground settings

Localization is the problem of knowing the position of an agent within a determinate environment. In the context of robotics, solving the localization problem means being able to know the position and orientation, also called pose, of an agent (robot, drone...) over time and within its deployment environment.

Nowadays, outdoors localization is almost a trivial problem, thanks to modern global positioning system (GPS) solutions. However, GPS won't work in environments without radio reception, such as indoors, underwater or underground settings [57]. Since this work aims for a solution suitable for subterranean environments, GPS cannot be used for localization. Dead reckoning is the estimation of location based on known speed, direction and time of travel with respect to a previous estimate. This means that each position estimate will depend on the "quality" of previous estimates: the errors will add up over time. To counter this, modern solutions often recur to landmarks, or features in the environment whose pose with respect to a global coordinate frame is known, and thus can provide a trustable position reference [57].

In any case, the true position of a robot agent cannot be known. The formal localization problem assumes that the robot has a true and unknown position $\boldsymbol{x}$, being $\hat{\boldsymbol{x}}$ its best available estimate. The uncertainty of this estimate is statistically defined as the standard deviation associated with the estimate $\hat{\boldsymbol{x}}$ [57].

Hence, it is common to approach the position of the robot with a *probability density function* or PDF. The intuition behind this concept is shown in Figure 68, which contains three PDF examples for different localizations. In these, xy coordinates denote spatial position, z being the value of the PDF at that point. Note that the PDF does not indicate the probability of a robot being at a certain point; instead, chosen a specific area in the xy plane, the volume under the PDF over that region would be the probability of the robot being inside said area. A relaxation of this idea would be considering likelihoods instead of PDFs, which is intuitively the same yet free from additional statistical constraints.

Typical robot applications make use of motion sensors to estimate change in position over time (odometry) for approaching the localization problem. Some examples of sensors that are typically used in pose estimation are encoders on ground vehicle wheels, electronic compass, gyroscopes, accelerometers, etc. All odometry measurements tend to be noisy, either in systematic error (such as the incorrect wheel radius or gyroscope) or random (such as slippage between wheels and ground or local magnetic field alterations) [57]. Also, due to the fact that change in position is usually computed indirectly from addition or integration of magnitudes, sensor uncertainty and drift error are representative issues.

The Kalman filter is a widely-spread mathematical tool typically found in localization applications for estimating a robot's true configuration from a series of noisy measurements. Under the assumption that a system's disturbances are zero-mean and gaussian, the Kalman filter can provide the best estimate of a system's state [57]. Typically, Kalman filters for localization use uncorrelated measurements from different sensors at once for a better estimate of the agent's position. This approach is widely known as *sensor fusion*.

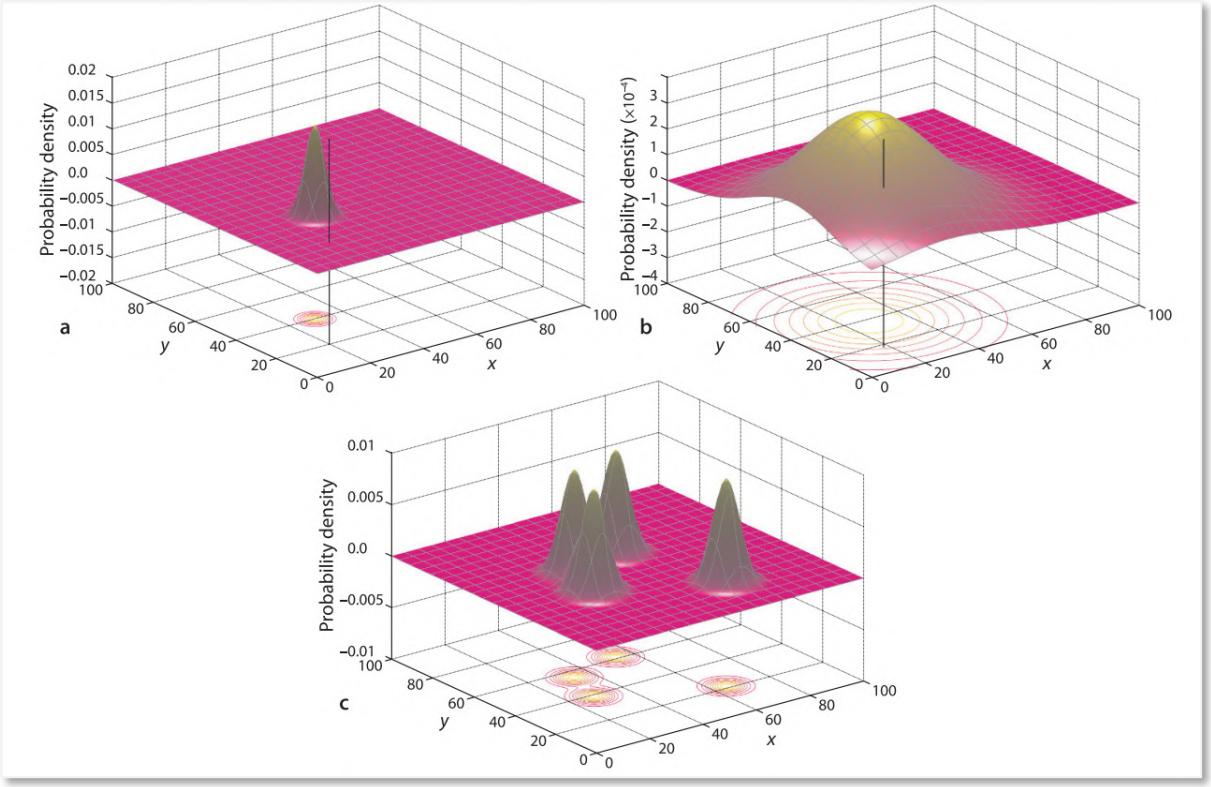


Figure 68: Likelihood of an agents's true position expressed as a PDF
Source: [57]

The localization problem can also be approached with the use of cameras as sensors. Computer vision techniques nowadays allow for the estimation of a camera's pose with respect to some features in the environment. In this work, image input is already being used for object detection, so expanding its use to the location task does not require the setup of additional sensory, and thus the use of a fiducial system is chosen as the path to follow for localization. Computer vision localization approaches, in essence, help obtaining the camera's pose. While the localization problem is formally defined for a mobile entity such as a robot, the relative pose between the robot and an onboard camera is usually known, and thus knowing the pose of the camera equals knowing the pose of the robot. In this work, the implementation stage of the localization problem will only be addressed for a camera due to the unavailability of an autonomous mobile device.

A fiducial marker system is composed by a set of valid markers and an algorithm which performs its detection, and possibly correction, in images [58]. There are many types of fiducial markers, the most spread in consumer technology perhaps being quick response or QR codes. The main features of a fiducial marker are its easy detection via camera and its capability to hold at the very least identity information i.e. a unique token that allows an automated system to know what marker is being detected. It is important to note that detection of fiducial markers is usually not performed under machine learning approaches, but deterministic algorithms that work an input image by applying a series of image operations such as gray-scaling, undistortion, thresholding, contouring... etc. Once a fiducial marker is identified, its pose with respect to the camera can be computed by the information contained in its shape.

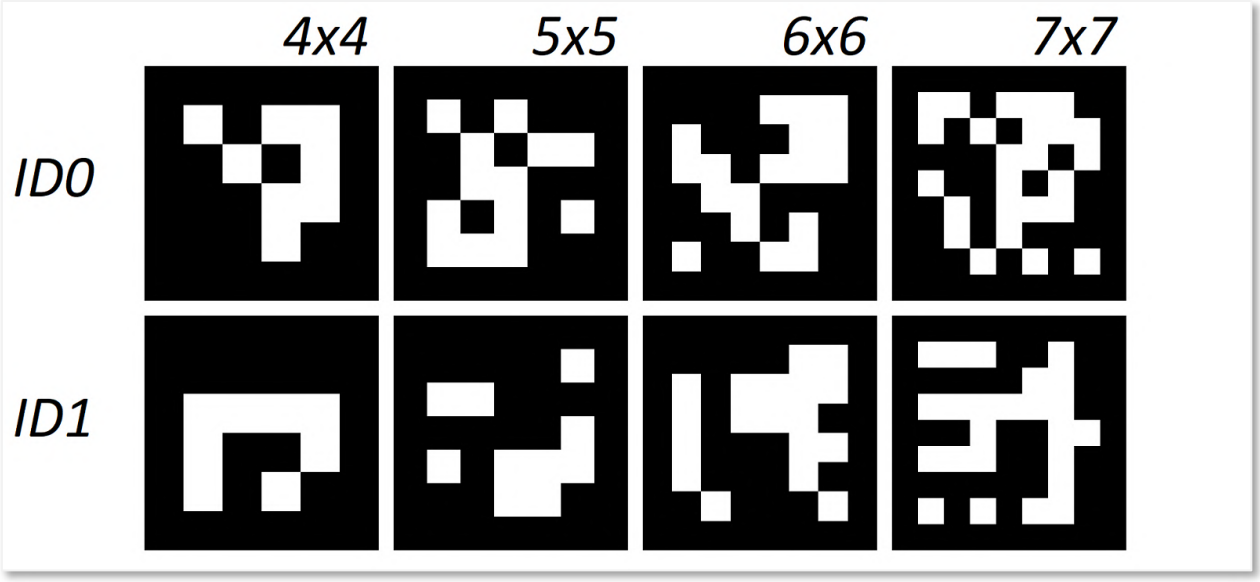


Figure 69: Some examples of ArUco markers 0 & 1 from different dictionaries
Source: Self-made

The fiducial system of choice for camera pose estimation within this work are the *ArUco markers*, introduced at [58]. ArUco markers are square black boxes, whose four corners can be used as significant points to identify its pose, containing a NxN binary matrix encoding its identity. There is a set of available standard ArUco dictionaries, ranging from 4x4 to 7x7 matrix size, and they can be generated either with the official library proposed in [58] or other implementations available online. Their visual simplicity and the availability of a high-level implementation of ArUco within the OpenCV library makes these markers easy to detect and work with implementation-wise. Figure 69 contains some examples of ArUco markers from different libraries generated with the tool at [59].

The detection of ArUco markers is not perfomed only by detecting their 3D world position, but also their pose. By convention, a marker's position is located at its center. Furthermore, its local coordinate frame is placed as follows. In the OpenCV implementation, these axes follow the RGB convention:

- X / red : Aiming to the right side.
- Y / green : Aiming upwards.
- Z / blue : Aiming perpendicular to the ArUco plane and pointing to the viewer.

In a real implementation setting, there must be a coordinate origin, also called *world coordinate frame*, with respect to which the coordinates of all items in the world will be referenced. Towards this work, the coordinate frame depicted by the ArUco marker with $ID0$ will be denoted as the world coordinate frame.

The following sections will briefly describe some of the mathematical building blocks used in robot localization, among other applications, that are of interest to this work, including: basics on pose algebra, formal definition of a Kalman filter for a given state space model and camera pinhole model.

5.2.2 Pose algebra

When addressing the localization problem for a robot or computer vision application, we are not only interested in finding the position but also the orientation of the agent. The combination of position and orientation is called *pose* [57]. The purpose of this section is to do a very brief, practical summary on the essentials of pose algebra.

A *coordinate frame* is a set of 3 axes that are placed on an item, be it moving or still. Figure 70 shows an example system with multiple coordinate frames. Just for a showcase of formal notation, some examples are pointed below:

- {O} : World coordinate frame (absolute reference).
- {R} : Robot coordinate frame.
- ξ_R : Robot pose.
- ${}^{R}\xi_{C}$: Camera pose with respect to the robot coordinate frame.

Relative poses can be composed to relate different elements within this system with the operator $\oplus$. This composition is yet only formal notation. For example, from Figure 70, the following compositions can be stated:

- $\xi_F = \xi_B \oplus {}^{B}\xi_F$
- $\xi_R = \xi_B \ominus {}^{C}\xi_B \ominus {}^{R}\xi_C$

More formal aspects on this can be found at [57], while this work will now focus on an example-led approach to show the numerical methods used to compose different poses.

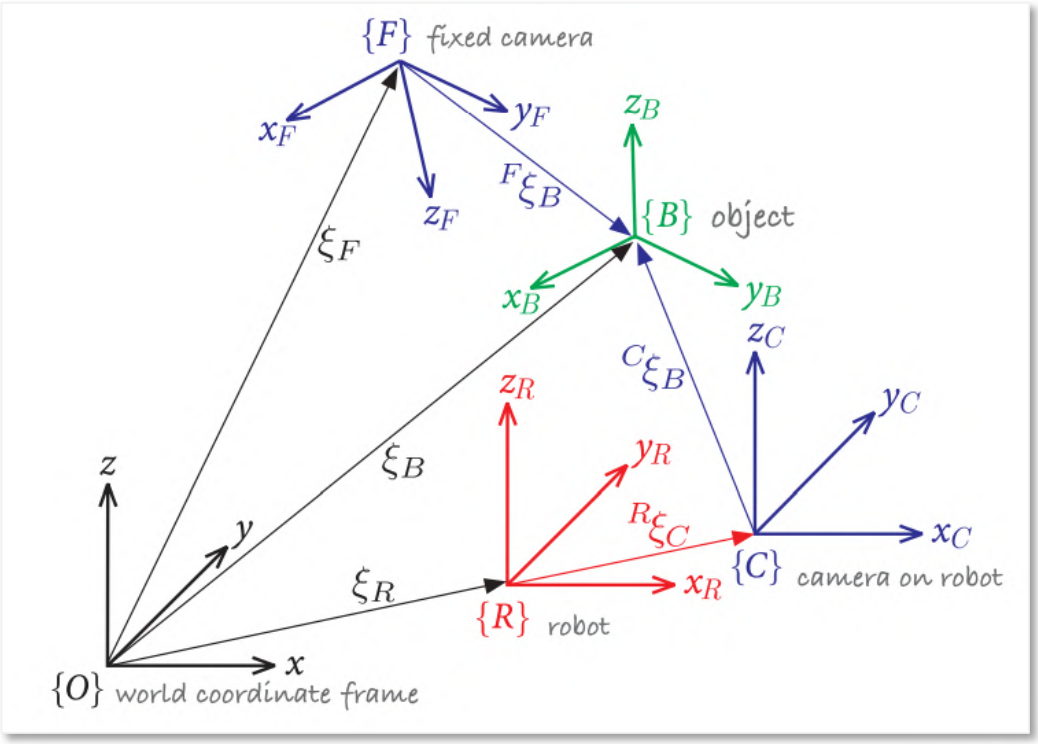


Figure 70: Multiple 3-dimensional coordinate frames and relative poses
Source: [57]

Take the 2D, cartesian system shown in Figure 71.a. Picture the coordinates of the point P with respect to the coordinate frame $\{B\}$ are known, but the coordinates of P with respect to $\{A\}$ are desired. The coordinate frame $\{B\}$ can be replicated by displacing a copy of $\{A\}$ along its axes for x and y, then rotating it θ degrees counter-clockwise. Formally: ${}^{A}\xi_{B} \sim (x, y, \theta)$.

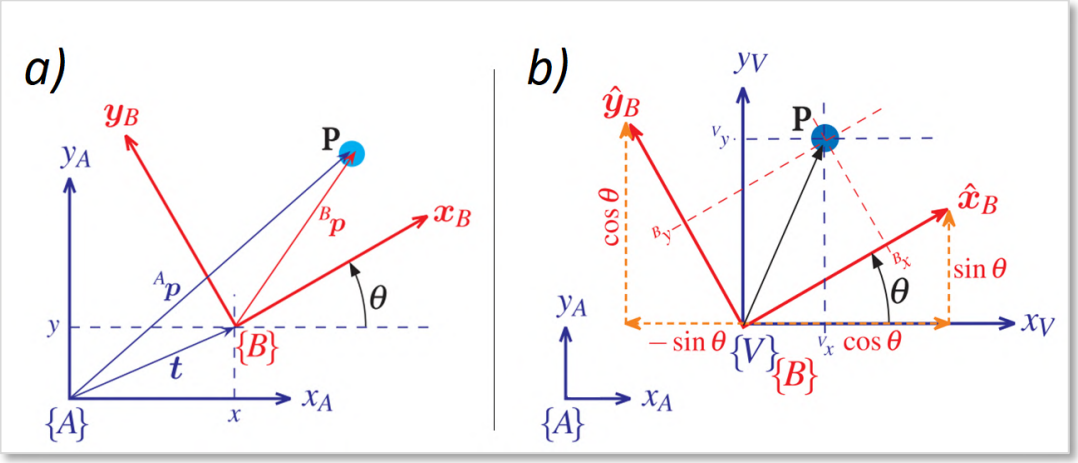


Figure 71: Example coordinate change in the cartesian plane
Source: [57]

To get the coordinates of $P\rfloor_A$, these steps need to be "reversed". First, we compute the coordinates of P with respect to a new coordinate frame $\{V\}$ which has the same origin as $\{B\}$ but axes square to $\{A\}$ (see Figure 71.b). This can easily be done with the following transformation:

$$\begin{pmatrix} {}^{V}x \\ {}^{V}y \end{pmatrix} = \begin{pmatrix} cos\,\theta & -sin\,\theta \\ sin\,\theta & cos\,\theta \end{pmatrix} \begin{pmatrix} {}^{B}x \\ {}^{B}y \end{pmatrix};\ {}^{V}\boldsymbol{R}_{B} := \begin{pmatrix} cos\,\theta & -sin\,\theta \\ sin\,\theta & cos\,\theta \end{pmatrix}$$

Where the *orthonormal rotation matrix* ${}^{V}\boldsymbol{R}_{B}$ is defined. Then, since $\{A\}$ and $\{V\}$ are parallel, we add the displacements in x & y to obtain the coordinates of P with respect to $\{A\}$:

$$\begin{pmatrix} {}^{A}x \\ {}^{A}y \end{pmatrix} = \begin{pmatrix} {}^{V}x \\ {}^{V}y \end{pmatrix} + \begin{pmatrix} x \\ y \end{pmatrix}$$

These two steps can also be performed in a single matricial operation. Because {A} and {V} are parallel, ${}^{A}\boldsymbol{R}_{B} := {}^{V}\boldsymbol{R}_{B}$:

$$\begin{pmatrix} {}^{A}x \\ {}^{A}y \\ 1 \end{pmatrix} = \begin{pmatrix} cos\,\theta & -sin\,\theta & x \\ sin\,\theta & cos\,\theta & y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} {}^{B}x \\ {}^{B}y \\ 1 \end{pmatrix} = \begin{pmatrix} {}^{A}\boldsymbol{R}_{B} & \boldsymbol{t} \\ \boldsymbol{0}_{1x2} & 1 \end{pmatrix} \begin{pmatrix} {}^{B}x \\ {}^{B}y \\ 1 \end{pmatrix};\ {}^{A}\boldsymbol{T}_{B} = \begin{pmatrix} {}^{A}\boldsymbol{R}_{B} & \boldsymbol{t} \\ \boldsymbol{0}_{1x2} & 1 \end{pmatrix} \quad (8)$$

Where the *homogeneous transformation* ${}^{A}\boldsymbol{T}_{B}$ is defined. Note that, from the first part of the Equation (8), it is easy to see that the last row of ${}^{A}\boldsymbol{T}_{B}$ could be omitted, however it is interesting that the homogeneous transformation is a square matrix, so that we can compute its inverse to perform the opposite transformation and do other operations with it.

The very same concept can be applied to the 3D case, though its proof gets much harder, since 3D rotations cannot be worked with as easily as 2D ones. For the purposes of this work, it is enough to note that the 3D homogeneous transformation can be performed by following the Equation (9):

$$\begin{pmatrix} {}^{A}x \\ {}^{A}y \\ {}^{A}z \\ 1 \end{pmatrix} = \begin{pmatrix} {}^{A}\boldsymbol{R}_{B} & \boldsymbol{t} \\ \boldsymbol{0}_{1x3} & 1 \end{pmatrix} \begin{pmatrix} {}^{B}x \\ {}^{B}y \\ {}^{B}z \\ 1 \end{pmatrix}; \quad {}^{A}\boldsymbol{T}_{B\,3D} \coloneqq \begin{pmatrix} {}^{A}\boldsymbol{R}_{B} & \boldsymbol{t} \\ \boldsymbol{0}_{1x3} & 1 \end{pmatrix} \tag{9}$$

For more information on this topic, and a detailed explanation on the contents here reviewed please refer to [57].

5.2.3 State space models and Kalman filters

A *state space representation* [60] is a model of a physical system, consisting of a set of input, output and state variables that are related by first-order differential equations, in continous systems, or difference equations, in the case of discrete ones. State space models use a set of *state variables*, contained in a *state vector*, as a representation for the inner state of the system, so that if both the current state and the input to a system are known, its next state can be computed. The variables that constitute the state vector are the designer's choice and don't need to be measurable or even represent real magnitudes as we know them. Because of this, the same physical system may have infinite different state space representations.

This representation generalizes to *multiple-input-multiple-output MIMO* systems in two matricial expressions: a propagation equation which determines how the physics of the system evolve in time and a measurement equation which relates the current state of the system to a vector of quantities that are measurable by real sensors and may provide feedback. The state space representation of a generic discrete physical system is shown in Equation (10):

$$\begin{cases} \boldsymbol{X}_{k+1} = \boldsymbol{A} * \boldsymbol{X}_{k} + \boldsymbol{B} * \boldsymbol{u}_{k} + \boldsymbol{w}_{k} \\ \boldsymbol{Y}_{k} = \boldsymbol{C} * \boldsymbol{X}_{k} + \boldsymbol{D} * \boldsymbol{u}_{k} + \boldsymbol{v}_{k} \end{cases} \tag{10}$$

In the propagation equation, the future state of the state vector $\boldsymbol{X}_{k+1}$ is computed from its current state $\boldsymbol{X}_{k}$ and input $\boldsymbol{u}_{k}$. In a real system, however, no mathematical model is completely accurate, and thus a term accounting for the system noise is added as the random variable $\boldsymbol{w}_{k}$.

In the measurement equation, the current output of the system $\boldsymbol{Y}_{k}$ is modelled from the current state vector $\boldsymbol{X}_{k}$ and the current input $\boldsymbol{u}_{k}$. The vector $\boldsymbol{Y}_{k}$ is composed by those magnitudes that can be measured by real sensors. A measurement noise random variable $\boldsymbol{v}_{k}$ is added to account for the sensor noise that makes the real measurements mismatch the theoretical model.

The Kalman Filter [61] is a recursive state estimator based on noisy measurements. It admits measurements that might be incomplete, indirect, intermittent or inexact and needs only to know the statistical properties of the process and sensor noise. Since the sensor measures are expected to be very noisy, especially in a localization task, the estimated state of the system is expected to provide more likely information than the sensors themselves.

Towards this application, the Kalman filter fullfills three needs:

- Reducing the random noise on the ArUco measured poses.
- Allowing the estimation of pose from more than one ArUco marker at once.
- Sensor fusion for pose estimation with both inertial sensors and ArUco markers.

The equations that implement the Kalman filter and allow for the computation of the estimated state are typically divided in two sets [61], shown in Equation (11).

$$\underline{\textit{Propagation cycle (time update)}}$$
$$\hat{\boldsymbol{x}}(k+1 \mid k) = \boldsymbol{A}(k) * \hat{\boldsymbol{x}}(k \mid k) + \boldsymbol{B}(k) * \boldsymbol{u}(k) \tag{11}$$
$$\boldsymbol{P}(k+1 \mid k) = \boldsymbol{A}(k) * \boldsymbol{P}(k+1 \mid k) * \boldsymbol{A^T}(k) + \boldsymbol{\Gamma}(\mathrm{k}) * \mathbf{Q}(\mathrm{k}) * \boldsymbol{\Gamma^T}(\mathrm{k})$$

$$\underline{\textit{Upgrade cycle (measurement update)}}$$
$$\boldsymbol{L}(k+1) = \boldsymbol{P}(k+1 \mid k) * \boldsymbol{C^T}(k+1) * [\boldsymbol{C}(k+1) * \boldsymbol{P}(k+1 \mid k) * \boldsymbol{C^T}(k+1) + \boldsymbol{R}(k+1)]^{-1}$$
$$\hat{\boldsymbol{x}}(k+1 \mid k+1) = \hat{\boldsymbol{x}}(k+1 \mid k) + \boldsymbol{L}(k+1) * [\boldsymbol{y}(k+1) - \boldsymbol{C}(k+1) * \hat{\boldsymbol{x}}(k+1 \mid k)]$$
$$\boldsymbol{P}(k+1 \mid k+1) = \boldsymbol{P}(k+1 \mid k) - \boldsymbol{L}(k+1) * \boldsymbol{C}(k+1) * \boldsymbol{P}(k+1 \mid k)$$

These computations take place every measurement's iteration, typically in real-time, in the order they are shown. The index k denotes an instant of time and the index $k+1$ denotes the next one. The index $k+1 \mid k$ denotes a quantity that is estimated for the instant $k+1$ whose computation takes place when only information up to k is available.

In the first place, the propagation cycle simply applies the propagation equation from the state space model of the system, only using the state vector estimate $\hat{\boldsymbol{x}}$ instead of the real value $\boldsymbol{x}$ to obtain the *a priori* estimated $\hat{\boldsymbol{x}}$. Then, the *a priori* estimation error covariance $\boldsymbol{P}$ is computed. In the upgrade cycle, the Kalman gain $\boldsymbol{L}$ is computed from the recently obtained $\boldsymbol{P}$; then the *a priori* estimate $\hat{\boldsymbol{x}}$ is corrected by adding a term $\boldsymbol{L}$-proportional to the estimation error, which is computed from the measurement equation of the state space model swapping $\boldsymbol{x}$ for the *a priori* $\hat{\boldsymbol{x}}$; finally, the estimation error covariance $\boldsymbol{P}$ is too corrected by using the Kalman gain $\boldsymbol{L}$.

The magnitudes involved in this Kalman filter are the following:

- $\boldsymbol{u}, \boldsymbol{y}$: System input and measurable system output. Fed in-line into the filter.
- $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$: Matrices defining the State space system model. Provided as input.
- $\boldsymbol{Q}$: Covariance matrix of the system noise $\boldsymbol{w}_k$. Estimated and provided as input.
- $\boldsymbol{R}$: Covariance matrix of the sensor noise $\boldsymbol{v}_k$. Estimated and provided as input.
- $\boldsymbol{P}, \boldsymbol{L}$: Estimation error covariance and Kalman gain, computed within the filter.
- $\hat{\boldsymbol{x}}$: System state estimate, computed within and main goal of the Kalman filter.

In normal conditions the Kalman gain $\boldsymbol{L}$ and the estimate error covariance $\boldsymbol{P}$ will converge to a specific set of values (time-variant Kalman filter), though for stationary systems these can be computed offline (time-invariant Kalman filter). While the Kalman filter is defined for linear systems, the Extended Kalman filter can be applied to non-linear systems by linearizing the system and the measurement's model around the current estimate. More modern variations of the Kalman filter also exists, among which the *Sequential Kalman filter* [62] is highlighted.

5.2.4 The pinhole camera model

The *pinhole camera model* [63] provides a simplified mathematical representation of a camera device, setting the foundings that allow relating real world positions and picture coordinates or *3D reconstruction*. The pinhole camera model is displayed in Figure 72. This model provides the camera with a coordinate frame in which the z axis follows the line of sight, while x and y are parallel to the picture plane and have the same direction as the ascending pixel coordinates u and v. This representation is used by the functions inside the OpenCV library.

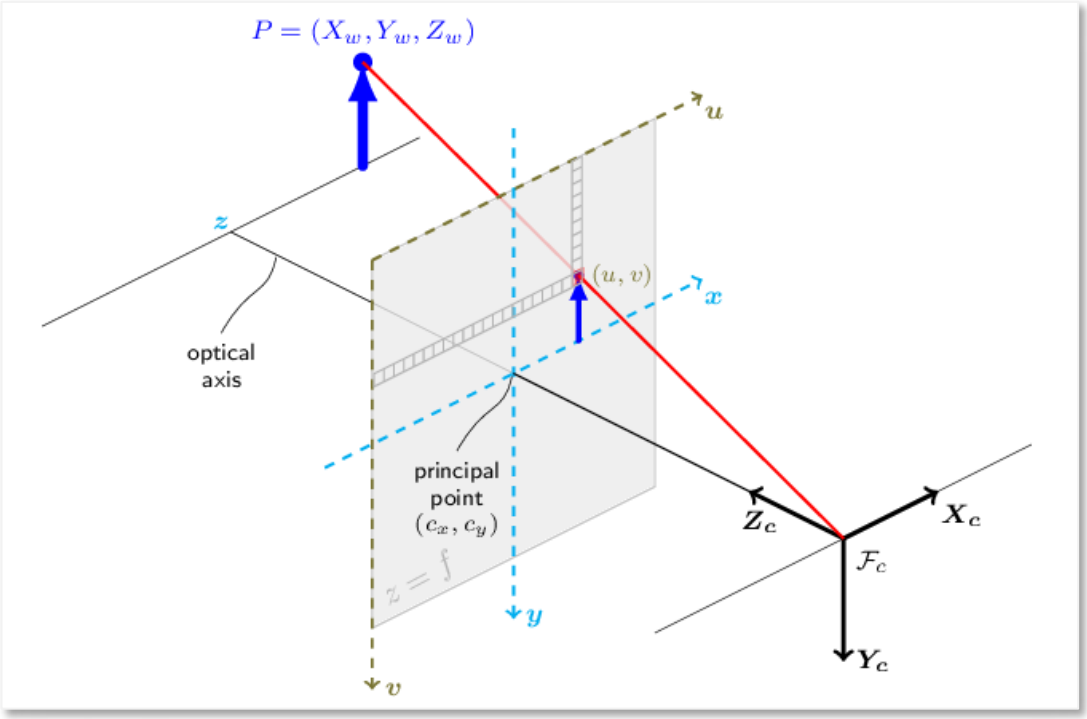


Figure 72: The pinhole camera model
Source: [63]

Consider an ideal pinhole camera capable of taking images free of any distortion. In these conditions, binding a real-world point $\boldsymbol{P}$ with its pixel coordinates $\boldsymbol{p}$ can be done via Equation (12).

$$s * \boldsymbol{p} = \boldsymbol{A} * [\, \boldsymbol{R} \mid \boldsymbol{t} \,] * \boldsymbol{P}_w \tag{12}$$

- $\boldsymbol{P}_w$: Real-world point, referenced to a world coordinate frame.
- $\boldsymbol{p}$: Pixel coordinates of the projection of P_w onto the image.
- $[\boldsymbol{R}|\boldsymbol{t}]$: Transformation matrix defining the relative pose of world and camera frame.
- $\boldsymbol{A}$: Camera intrinsic matrix.
- s : Projective transformation scaling factor.

Which simplifies to the following expression if $\boldsymbol{P}$ is referenced to the camera frame:

$$\boldsymbol{P}_c = [\, \boldsymbol{R} \mid \boldsymbol{t} \,] * \boldsymbol{P}_w \quad \Rightarrow \quad s * \boldsymbol{p} = \boldsymbol{A} * \boldsymbol{P}_c$$

Taking this expression and breaking down $\boldsymbol{A}, \boldsymbol{P}_c$ and $\boldsymbol{p}$ in their components we arrive to Equation (13):

$$s\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} X_c \\ Y_c \\ Z_c \end{pmatrix} \tag{13}$$

This expression provides a powerful bridge between pixels and real locations (with respect to the camera) by a set of parameters contained in the camera intrinsic matrix $\boldsymbol{A}$. These are properties of the camera, usually obtained via a calibration procedure, and are measured in pixel units. The terms shown in this equation are:

- u, v : Pixel coordinates of $\boldsymbol{P}$ within an image, origin on the upper left corner.
- X_c, Y_c, Z_c : Real world position of $\boldsymbol{P}$ with respect to the camera frame.
- f_x, f_y : Focal distance in x & y.
- c_x, c_y : Principal point coordinates, where the camera z axis intersects the image.

These mathematical expressions are representative under the hypothesis of an ideal camera which produces images free of any distortion. However, real cameras take pictures with different types of distortions, generally a predominant radial distortion and a slight tangential one [63]. Figure 73 shows how different distortions alter an example picture. Depending on the author or platform, a different number of parameters may be selected to model these:

- *Tangential distortion*: Occurs when the lens and camera sensor are not parallel, in a way that some areas of the image may appear "closer". Defined by coeffs. (p_1, p_2).
- *Radial distortion*: Which may be positive or negative and is constant for a given lens. Warping increases the farther a pixel is from the principal point. Defined by coeffs. $(k_1, k_2, k_3, k_4, k_5, k_6)$.

The following expressions hold the pertinent distortion corrections to the pinhole model and can be used for relating real world points with pixel coordinates in real images [63]:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} X_c/Z_c \\ Y_c/Z_c \end{pmatrix}$$

$$r^2 = x'^2 + y'^2$$

$$\begin{pmatrix} x'' \\ y'' \end{pmatrix} = \begin{pmatrix} x' * \dfrac{1 + k_1 r^2 + k_2 r^4 + k_3 r^6}{1 + k_4 r^2 + k_5 r^4 + k_5 r^6} + 2p_1 x' y' + p_2\left(r^2 + 2x'^2\right) + s_1 r^2 + s_2 r^4 \\ y' * \dfrac{1 + k_1 r^2 + k_2 r^4 + k_3 r^6}{1 + k_4 r^2 + k_5 r^4 + k_5 r^6} + p_1\left(r^2 + 2x'^2\right) + 2p_2 x' y' + s_1 r^2 + s_2 r^4 \end{pmatrix} \tag{14}$$

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} f_x * x'' + c_x \\ f_y * y'' + c_y \end{pmatrix}$$

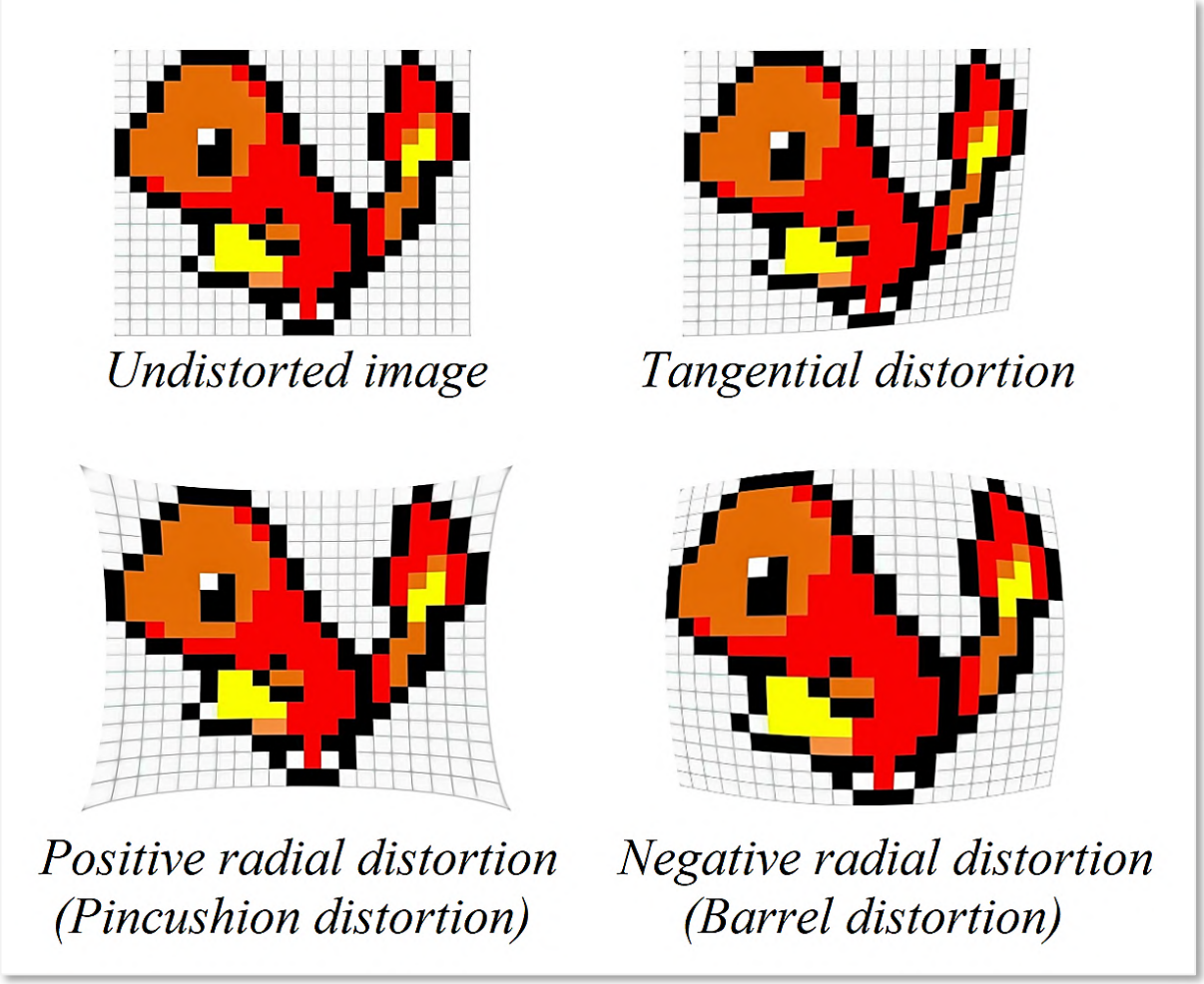


Figure 73: Tangential and radial distortions in images
Source: Self-made

5.2.5 Camera calibration

Camera calibration is a procedure in which a real camera is experimentally tested to infer its camera intrinsic matrix $\boldsymbol{A}$, as well as a set of its distortion parameters up to a certain order, which depends on the specific computational tool used for calibration. This test is performed by taking a series of pictures of a given pattern of known geometry from different points of view, then having an algorithm infer the distortion and camera parameters so that its expected geometry can be reconstructed.

Towards this work, camera calibration is performed by using the ChArUco board displayed in Figure 74. A ChArUco board combines a chessboard pattern with fiducial ArUco markers placed in the white spots. Calibration with these boards is expected to be more accurate and less prone to error than calibration in either a chessboard pattern or single ArUco markers, since there are more points for reference and occlusions and partial views are allowed [64]. Calibration procedure consists on taking a set of pictures of said pattern's printout and feeding them to a high-level function of the OpenCV library that yields the intrinsic camera distortion matrix $\boldsymbol{A}$ and the distortion coefficients $(k_1, k_2, k_3, p_1, p_2)$. Under this approach, the rest of distortion coefficients shown in the distortion-correction equations are assumed to be zero [65].

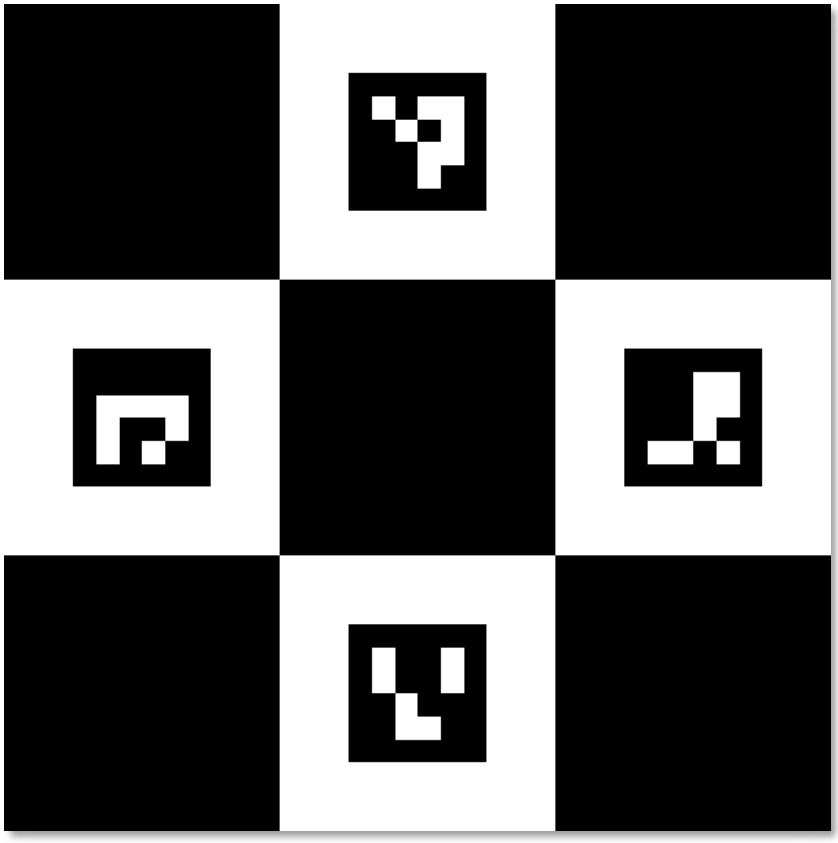


Figure 74: Example ChArUco pattern
Source: Self-made

Because the video feed sent by IP Webcam is of undeterminate characteristics (camera source, resolution, in-line processing, etc.), images straight up taken with the phone camera in a conventional manner are not to be used for calibration. Instead, live feed is sent to the computer, from which certain frames are extracted from this stream at the press of a key. After the capture of a small collection of images, high-level calibration functions from the OpenCV library are run using these as source to obtain the camera parameters. This way, camera parameters are more representative of the images that will be received in deployment.

Implementation-wise, camera parameters are stored in a file that can be loaded by a computer program to recover the inferred intrinsic matrix and distortion coefficients later, so that camera calibration is a process performed only once per camera.

5.3 Camera pose estimation with ArUco markers

5.3.1 Finding the pose of ArUco markers with respect to the camera

The pose of an ArUco marker with respect to the camera can be obtained with high-level OpenCV functions by completing the following steps:

- The marker and its significant points are found within the image as pixel coordinates.
- The pose of said marker is then calculated from its four corners by solving a *Perspective-n-Point* problem [66], which returns the position and rotation vectors of the ArUco marker with respect to the camera.

5.3.2 Finding the pose of the camera with respect to an ArUco marker

At this point, the 3D homogeneous transformation matrix that links the ArUco marker to the camera can be built from the quantities obtained by the procedure depicted in the previous section. For notation, let's define the following coordinate frames:

- $\{C\}$: Placed at the camera, axes matching the pinhole model.
- $\{A\}$: Placed at the ArUco marker, axes matching those of the marker.

From the available translation vector ${}^{C}\boldsymbol{t}_{A}$ and rotation matrix ${}^{C}\boldsymbol{R}_{A}$ (which is obtained by using the *Rodrigues' Rotation formula* [63] on the rotation vector) we can construct the homogeneous transformation ${}^{C}\boldsymbol{T}_{A}$ that relates $\{A\}$ to $\{C\}$. Picture now a point $\boldsymbol{P}$ with arbitary coordinates referenced to $\{A\}$ $\boldsymbol{P}_{A}$. Its coordinates with respect to $\{C\}$ can be computed as:

$$\begin{pmatrix} {}^{C}x \\ {}^{C}y \\ {}^{C}z \\ 1 \end{pmatrix} = {}^{C}\boldsymbol{T}_{A} \underbrace{\begin{pmatrix} {}^{A}x \\ {}^{A}y \\ {}^{A}z \\ 1 \end{pmatrix}}_{\boldsymbol{P_A}} \quad ; \quad {}^{C}\boldsymbol{T}_{A} := \begin{pmatrix} {}^{C}\boldsymbol{R}_{A} & {}^{C}\boldsymbol{t}_{A} \\ \boldsymbol{0}_{1x3} & 1 \end{pmatrix}$$

From here, picture the opposite situation. Imagine a point $\boldsymbol{Q}$ with arbitary coordinates referenced to $\{C\}$ $\boldsymbol{C}_{A}$. Its coordinates with respect to the ArUco marker $\{A\}$ can be obtained as:

$$\begin{pmatrix} {}^{A}x \\ {}^{A}y \\ {}^{A}z \\ 1 \end{pmatrix} = {}^{C}\boldsymbol{T}_{A}{}^{-1} \underbrace{\begin{pmatrix} {}^{C}x \\ {}^{C}y \\ {}^{C}z \\ 1 \end{pmatrix}}_{\boldsymbol{Q_C}}$$

Imagine that, conveniently, point $\boldsymbol{Q}$ were to be placed at the camera's origin. Obtaining its coordinates with respect to $\{A\}$ is then equal to obtaining the camera coordinates. Under this assumption, we can compute the position of the camera with respect to the ArUco marker as:

$$\begin{pmatrix} {}^{A}x \\ {}^{A}y \\ {}^{A}z \\ 1 \end{pmatrix} = {}^{C}\boldsymbol{T}_{A}{}^{-1} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \tag{15}$$

Finally, after obtaining the position, the camera yaw θ is computed by obtaining the Euler angles [57] from the projection matrix ${}^{A}\boldsymbol{T}_{C} \coloneqq {}^{C}\boldsymbol{T}_{A}{}^{-1}$.

5.3.3 Two-dimensional ArUco marker layout

Because the goal is to solve localization problem only horizontally, it should be possible to represent the pose of the camera with respect to a specific ArUco marker by only its xz position and horizontal-plane orientation θ with respect to the marker's coordinate frame. Under this statement, the pose of the camera with respect to a marker A can be stated by the vector $({}^{A}x, {}^{A}z, {}^{A}\theta)$, where the superscript ^{A} denotes this particular marker.

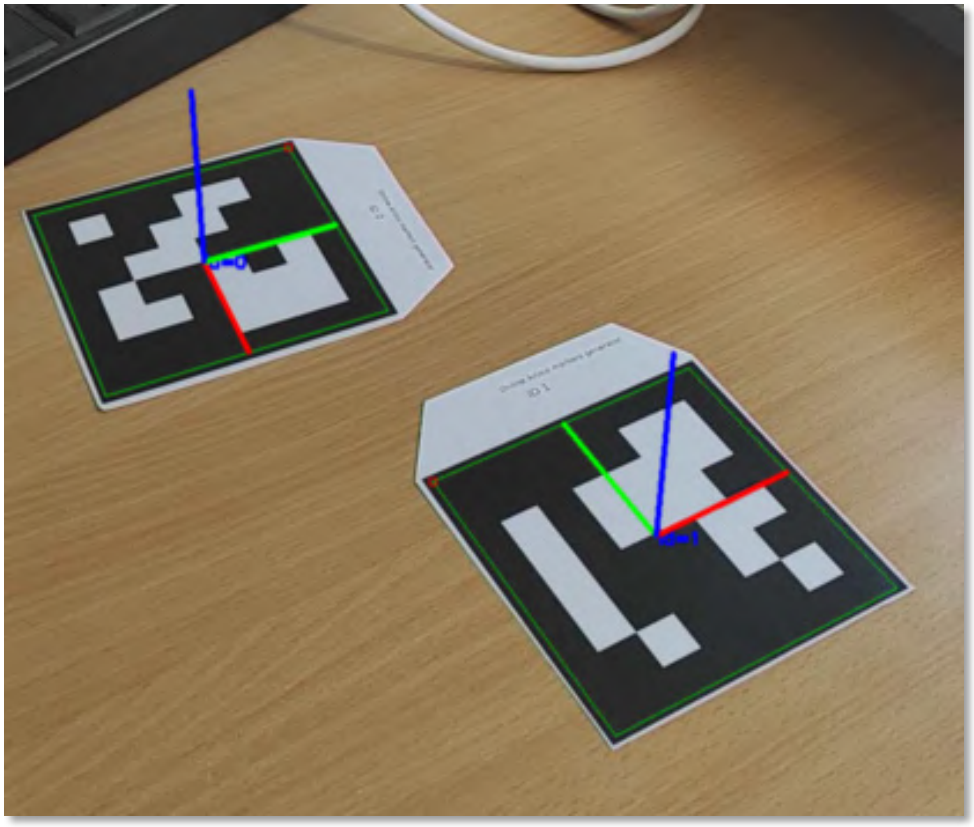


Figure 75: Detection of ArUco markers
Source: Self-made

If markers were to be placed in unconstrained settings, such as in the floor, ceiling or walls without regatd to their axes' orientation, their complete 3D pose should be accounted for, and thus a total of 6 parameters (xyz plus $yaw, pitch, roll$) would be required to completely define the pose of a marker instead of the optimal 3.

Figure 75 shows two ArUco marker frames being identified by the procedure described in the previous sections. Coordinate frames painted on these follow the RGB color convention. In order to simplify experimental fiducial marker layout, we can stablish a placement criterion in which markers are placed in walls, matching axes and orientations as follows:

- R, red : x axis, horizontal.
- G, green : y axis, vertical aiming upwards.
- B, blue : z axis, horizontal aiming towards the marker observer.

Under this placement criterion, y axes can be safely ignored towards horizontal localization, so that coordinates of ArUcos in the environment can be expressed by only their xz position and horizontal-plane orientation θ to a reference coordinate frame. This holds true as long as the y axes of different markers are parallel, even if the fiducial markers are placed at different heights.

Towards building a fiducial marker map for localization and assemblying a localization pipeline, a common reference frame is needed to define the position or pose of all entities in the environment (markers, agents, items). Towards this work, the coordinate frame defined by the ArUco marker with $ID0$ is defined as the world's coordinate frame for all experiments.

5.4 Improving pose estimation with Kalman filters

5.4.1 Disturbance analysis and previous considerations

The method recently described has proven useful for the localization of the camera with respect to a single ArUco marker representing the world's coordinate frame. However, it is observed during real-time testing that the camera pose signal is slightly disturbed by random noise. While sometimes this noise could be argued to be of a negligible order, it is desired to increase the stability of the signal by Kalman filtering.

Initialy, this unstabilities were perceived by numerically displaying measurements of the signals $xz\theta$ on a terminal window. Because conclusions cannot be extracted from such unreliable visualization setting, an experimental test was perfomed to get a profile of the noise and check the characteristics of the measured pose signals, consisting of:

- Resting a camera on a tripod with a clear view of an ArUco marker.
- Initialize pose estimation in real time with the method described in Section 5.3.
- Recording a time series of the $xz\theta$ signals for a duration of one minute.
- Analyze the obtained time-series data, under the following assumptions:
    - Both the camera and marker have stayed perfectly static.
    - All disturbances belong to measurement noise.
    - It is possible to subtract the real pose (constant unknown value) from each measurement, so that its histogram is representative of the noise profile.

An overview of the setup for this experiment is displayed on Figure 76, while the retrieved time-series and histograms for the measured $xz\theta$ signals are shown in Figure 77.

Figure 76: Setup for obtaining the measurement noise profile of fiducial pose estimation
Source: Self-made

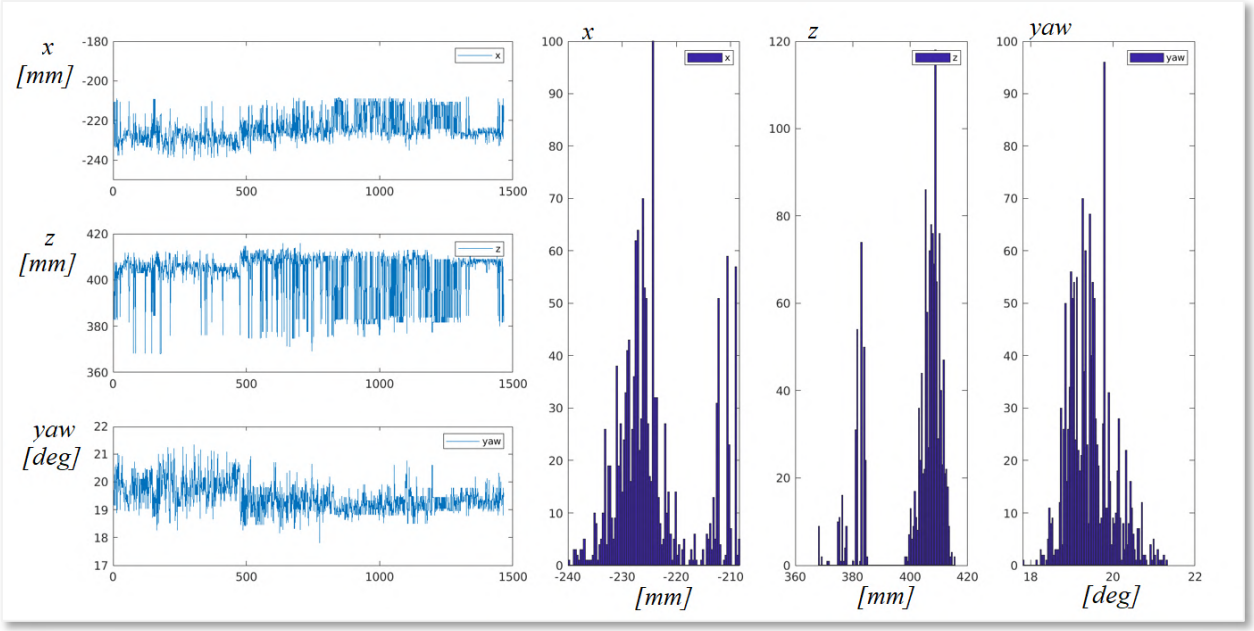


Figure 77: Time series and histogram of a static position measured with ArUco markers
Source: Self-made

From these, it is observed that the noise does not follow a gaussian distribution, but a more complex, discontinuous one. Current perturbations on xz positions are small, on the order of $50\ mm$, while noise in the yaw signal θ is more impactful, within the order of 4°. Perturbations in θ can lead to high inaccuracies when identifying the position of items placed far away from the camera.

The positioning errors are thought to originate from the following sources:

- Pixel artifacts due to compression in the livestream footage.
- Artifacts due to lighting conditions changes: people passing by, ambient light... etc.
- Movement of the ArUco marker: which was printed on paper and fixed with tape to a vertical surface, stilll susceptible to some degree of curvature and play.

These sources of error randomly displace the ArUco marker a few pixels inside the recorded image. It is this discrete displacement within the image that could explain the discontinuous noise profile visible in the histograms.

Formally, it can be proven that the Kalman filter provides the best estimator of a system's state if this is subjected to white noise, however it can be anyway implemented if this condition fails with suboptimal outcome. Despite the sensor noise being blatantly non-gaussian, three reasons still favour the implementation of a Kalman filter in this setting:

- The Author's own interest on the implementation of a Kalman filter for didactical purposes.
- The use of sensor fusion to combine measurements taken from various ArUco markers at once.
- The potential use of sensor fusion to combine visual ArUco positioning with inertial sensory to keep track of pose when no ArUco markers are in sight.

5.4.2 Denoising ArUco marker poses

The first implementation of a Kalman filter is performed for a single ArUco marker, as starting point for laying more complex models. The chosen state space representation of the localization problem is built as follows:

- The single ArUco marker that is being considered defines a world coordinate frame $\{W\}$ with pose $(x, z, \theta) = (0, 0, 0)$.
- The state vector $\boldsymbol{X}$ holds the camera's pose (x, z, θ) and its first derivative $(\dot{x}, \dot{z}, \dot{\theta})$.
- The measurement vector $\boldsymbol{Y}$ holds the ArUco-measured camera pose $(x, z, \theta)_{measured}$.
- The input of the system $\boldsymbol{U}$ is null, and so it its $\boldsymbol{B}$ matrix.
- The change in position of the camera is expected to be random. Furthermore, it is hypothesized that it will follow a gaussian distribution and so it is modelled by the system noise $\boldsymbol{w}_k \sim N(0, \boldsymbol{Q})$. Its covariance matrix $\boldsymbol{Q}$ is estimated by intuition when tuning the filter.
- Though it has been proven otherwise, the sensor noise is blatantly hypothesized to be gaussian so that $\boldsymbol{v}_k \sim N(0, \boldsymbol{R})$. Its covariance matrix $\boldsymbol{R}$ estimated from the recorded data, previously shown, as the covariance of the whole time-series measurement.
- The system is deemed to be real and thus causal, its matrix $\boldsymbol{D}$ null.
- Matrices $\boldsymbol{A}$ and $\boldsymbol{C}$ are defined to match the chosen $\boldsymbol{X}$ & $\boldsymbol{Y}$ in this setting.

A simple representation of this system is shown in Figure 78. The camera holds a pose $\xi_C \sim (x, z, \theta)$ with respect to the world coordinate frame $\{W\}$ defined by an ArUco marker. Note that this is only a example schematic: the camera should be aiming at the ArUco and the axis relative positions might not match other reference frames depicted in this work.

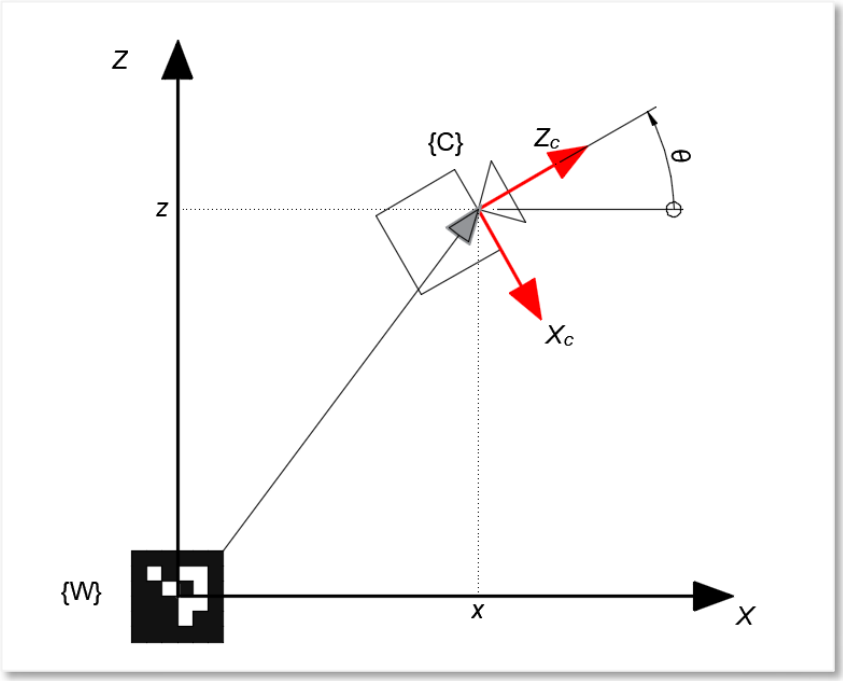


Figure 78: Dummy schematic towards the ArUco localization state space representation
Source: Self-made

The mathematical representation of this state space model is laid out in Equation (16).

$$\begin{cases} \begin{pmatrix} x \\ \dot{x} \\ z \\ \dot{z} \\ \theta \\ \dot{\theta} \end{pmatrix}_{k+1} = \begin{pmatrix} 1 & Ts & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & Ts & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & Ts \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} x \\ \dot{x} \\ z \\ \dot{z} \\ \theta \\ \dot{\theta} \end{pmatrix}_k + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} * (0) + \boldsymbol{w}_k \\ \begin{pmatrix} x \\ z \\ \theta \end{pmatrix}_k = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} * \begin{pmatrix} x \\ \dot{x} \\ z \\ \dot{z} \\ \theta \\ \dot{\theta} \end{pmatrix}_k + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} * (0) + \boldsymbol{v}_k \end{cases} \quad (16)$$

Filter tuning, i.e. the proper definition of $\boldsymbol{Q}$, would generally be meaningful towards the dynamics of the system, since the camera movement is modelled within $\boldsymbol{w}_k$. However, because the system was modelled under a lot of assumptions that are not completely representative it is not expected that a poorly selected $\boldsymbol{Q}$ increase the estimation error significantly at the moment. Thus, current results are deemed admissible yet susceptible to be improved, leaving the Kalman filter tuning for a later stage of implementation.

The implementation of the filter for denoising a pose signal computed from a single ArUco marker is succesfully completed under the stated assumptions. Figure 79 shows a time-series of the yaw θ and the position x, both processed and unfiltered. Despite the bold assumptions that have been made, the signal is seen to be significantly smoothed. Note that, in case the assumption that $\mathrm{E}[\boldsymbol{v}_k] = 0$ fails, the outcome of the Kalman filter might be biased.

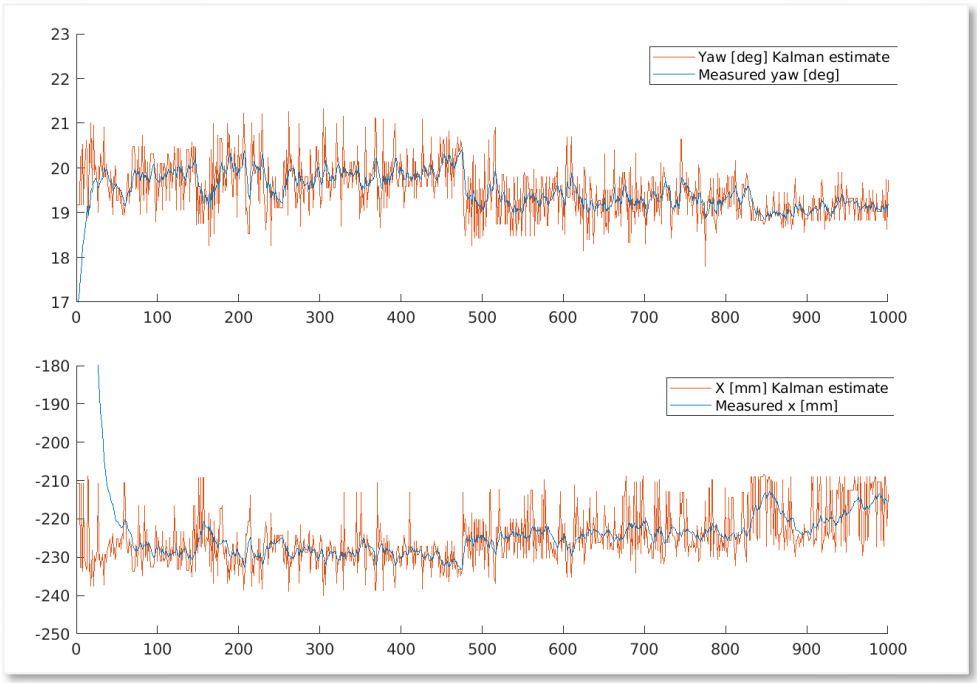


Figure 79: Raw and Kalman filtered pose signals obtained from a single ArUco marker
Source: Self-made

5.4.3 Multi-ArUco pose estimation

Estimating pose from different ArUco marker measurements at the same time requires either a) the implementation of a Sequential Kalman filter [62] or b) the implementation of a Kalman filter in which matrices can change shape every iteration. Provided its standing hypotheses are met, the Sequential Kalman filter achieves the same results being lighter in computation and conceptually easier to implement. The existing implementation of the Kalman filter favoured the latter, however, which only required a few in-line modifications on the program's main loop to adjust for the time-variant state space model.

Firs of all, it is necessary to define all of the ArUco markers' pose, as well as the camera's, to a common world coordinate frame $\{W\}$. For simplicity, $\{W\}$ is set to match the $ID0$ ArUco marker's coordinate frame. Because each ArUco estimates the position of the camera with respect to itself, each measurement is converted via pose algebra before serving it to the measurement vector $\boldsymbol{Y}$ of the Kalman filter. This is, $\boldsymbol{Y}$ is mostly composed of multiple indirect measurements (x_i, z_i, θ_i) of the pose of the camera with respect to the $ID0$ marker $\{W\}$. Thus, the pose of each ArUco marker with respect to $\{W\}$ must be known beforehand and provided as an input to the system.

Towards the Kalman filter implementation, the state vector $\boldsymbol{X}$ and the matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ stay the same as the single-marker model's. The hypothesis that all variation in position comes from the process noise $\boldsymbol{w}_k$ is also kept. Thus, the whole propagation equation remains unchanged.

On the other hand, the following matrices need to be adjusted every iteration:

- The measurement vector $\boldsymbol{Y}$.
- The observation matrix $\boldsymbol{C}$.
- The measurement noise covariance $\boldsymbol{R}$.
- The Kalman filter gain $\boldsymbol{L}$.

The size of $\boldsymbol{Y}$ is determined by the number of measurements that are being taken, or in other words, the number of ArUco markers on sight. If the items in $\boldsymbol{Y}$ are properly arranged, the broadcasting of the matrix $\boldsymbol{C}$ can be easily done. Equation (17) shows the measurement equations of the State Space model for one and two detected markers under a proper $\boldsymbol{Y}$ arrangement. Note that the $\boldsymbol{D}$ matrix has been omitted due to the system being causal.

$$\begin{pmatrix} x_0 \\ z_0 \\ \theta_0 \end{pmatrix}_k = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} * \begin{pmatrix} x \\ \dot{x} \\ z \\ \dot{z} \\ \theta \\ \dot{\theta} \end{pmatrix}_k + w_k \; ; \; w_k \sim N(0, Q_{6x6}) \qquad (17)$$

$$\left(\begin{Bmatrix} x_0 \\ z_0 \\ \theta_0 \end{Bmatrix} \begin{Bmatrix} x_1 \\ z_1 \\ \theta_1 \end{Bmatrix} \right)_k = \left(\begin{Bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{Bmatrix} \begin{Bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{Bmatrix} \right) * \begin{pmatrix} x \\ \dot{x} \\ z \\ \dot{z} \\ \theta \\ \dot{\theta} \end{pmatrix}_k + \boldsymbol{v}_k \; ; \; \boldsymbol{v}_k \sim N\left(0, \begin{pmatrix} \boldsymbol{R}_{3x3} & 0_{3x3} \\ 0_{3x3} & \boldsymbol{R}_{3x3} \end{pmatrix}\right)$$

From these examples, it can be seen that if $\boldsymbol{Y}$ is composed by stacks of poses in the shape $(x, z, \theta)^T$, the broadcasting of $\boldsymbol{C}$ can be done by simply stacking the blocks between brackets, which always hold constant values. Also, the noise covariance matrix needs to be adjusted, having the same square size as there are elements in $\boldsymbol{Y}$. In this case, since the noise is computed from recorded data for a single marker, a constant covariance matrix $\boldsymbol{R}$ is diagonally stacked as shown in the Equation (17).

Finally, the Kalman filter gain $\boldsymbol{L}$ size can be determined from the 2nd equation of the Upgrade cycle of the canonical Linear Kalman filter. This expression is shown in Equation (18), from where it can be seen that its size must be equal to $6xn$, where n is the number of elements in $\boldsymbol{Y}$. For simplicity, the error term is aliased with $\tilde{\boldsymbol{y}}$, its elements carrying the accent $\sim$.

$$\boldsymbol{X}_{k_{6x1}} = \boldsymbol{X}_{k-1_{6x1}} + \boldsymbol{L}_{6xn} * (\boldsymbol{y} - \boldsymbol{C}\boldsymbol{x}_k)_{nx1}$$

$$\tilde{\boldsymbol{y}}_{nx1} \coloneqq (\boldsymbol{y} - \boldsymbol{C}\boldsymbol{x}_k)_{nx1}$$

$$\begin{pmatrix} x \\ \dot{x} \\ z \\ \dot{z} \\ \theta \\ \dot{\theta} \end{pmatrix}_k = \begin{pmatrix} x \\ \dot{x} \\ z \\ \dot{z} \\ \theta \\ \dot{\theta} \end{pmatrix}_{k-1} + \left(\begin{Bmatrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{Bmatrix} \begin{Bmatrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{Bmatrix} \right) * \left(\begin{Bmatrix} \widetilde{x_0} \\ \widetilde{z_0} \\ \widetilde{\theta_0} \end{Bmatrix} \begin{Bmatrix} \widetilde{x_1} \\ \widetilde{z_1} \\ \widetilde{\theta_1} \end{Bmatrix} \right) \tag{18}$$

From this equation, it can be seen that $\boldsymbol{L}$ must grow horizontally 3 columns at a time when a measurement is added. By broadcasting this block the size of $\boldsymbol{L}$ can be fit to accomodate for variable measurement vector sizes. This block, however, is not to be computed *a priori*, but approached by the filter in-line. By following the matricial product, it can be seen that each block of dimension $6x3$ will be the gain of a specific slice of the measurement vector $\boldsymbol{Y}$, this is, each block belongs to a particular ArUco marker's measurements. For the broadcasting of $\boldsymbol{L}$, there are thus two alternatives:

- Assume the $6x3$ block should be equal for every ArUco marker.
- Assume that the Kalman gain slice is a property of each ArUco marker.

Since the latter solution was thought to be the most accurate, its computational cost irrelevant, it was adopted that each ArUco marker would store its own Kalman gain, which will be updated within the filter only when said markers are in sight of the camera and thus, included in the state space model.

By performing these broadcasting steps, the standard Kalman filter can be accomodated to fit a variable number of measurements by modifying its properties in-line. Furthermore, because matrices are recomputed every iteration, adjusting for irregular sample periods requires a trivial recomputation of the fixed, small-size matrix $\boldsymbol{A}$ every loop. This model is experimentally validated and deemed representative, its final accuracy yet dependent on the filter's fine tuning, yet again left for a later stage of development. Figure 80 and Figure 81 show a schematic of the top view plus an overlook on the experimental setup used for validating the model.

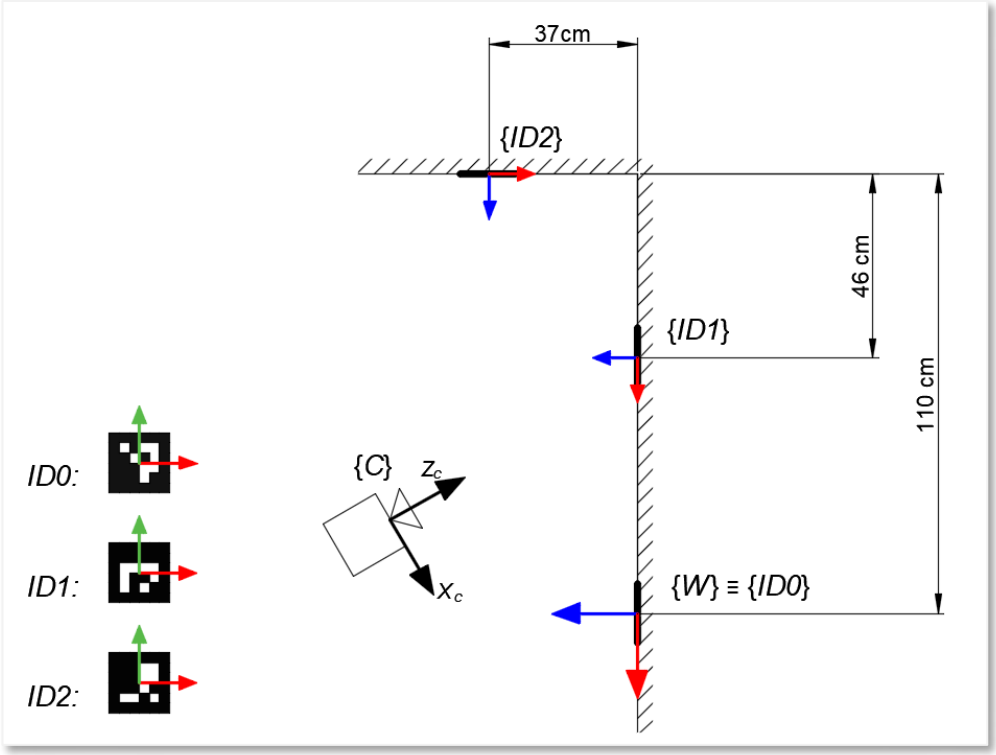


Figure 80: Schematic of the ArUco layout used to test the broadcastable Kalman filter
Source: Self-made

Figure 81: Overview of the ArUco layout used to test the broadcastable Kalman filter
Source: Self-made

Formally, we define the state-space model of the localization problem for an instant in which a number of N ArUco markers are in sight with ids ranging from a to b as:

$$\begin{cases} \begin{pmatrix} x \\ \dot{x} \\ z \\ \dot{z} \\ \theta \\ \dot{\theta} \end{pmatrix}_{k+1} = \begin{pmatrix} 1 & Ts & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & Ts & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & Ts \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}_k * \begin{pmatrix} x \\ \dot{x} \\ z \\ \dot{z} \\ \theta \\ \dot{\theta} \end{pmatrix}_k + \boldsymbol{w}_k \\ \begin{pmatrix} \begin{Bmatrix} x \\ z \\ \theta \end{Bmatrix}_a \\ \vdots \\ \begin{Bmatrix} x \\ z \\ \theta \end{Bmatrix}_b \end{pmatrix}_k = \begin{pmatrix} \begin{Bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{Bmatrix}_a \\ \vdots \\ \begin{Bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{Bmatrix}_b \end{pmatrix}_k * \begin{pmatrix} x \\ \dot{x} \\ z \\ \dot{z} \\ \theta \\ \dot{\theta} \end{pmatrix}_k + \boldsymbol{v}_k \\ \boldsymbol{w}_k \sim N(0, \boldsymbol{Q}) \; ; \quad \boldsymbol{v}_k \sim N\left(0, \begin{pmatrix} \boldsymbol{R}_{3x3} & \vdots & \mathbf{0} \\ \vdots & \ddots & \vdots \\ \mathbf{0} & \cdots & \boldsymbol{R}_{3x3} \end{pmatrix}_{3Nx3N}\right) \end{cases} \tag{19}$$

5.4.4 Addressing the non-linearity of angle predictions

So far, the proposed Kalman filter implementations have suggested the use of the yaw angle as one of its state and measurement variables. It is only when testing 360° ArUco layouts that it is found that this is not a viable solution. The yaw angle is a bounded and looped magnitude, for which two different values might have the same meaning (think 0° and 360°).

In practice, the multi-ArUco Kalman filter is proposing a filtered yaw angle computed as a weighted average between the yaw angles of the camera measured by different, individual ArUco markers. Angles may be represented by following different bounding conventions such as $[0°, 360°)$ or $[-180°, 180°)$, though they always fail around the bounding point. Following the latter, let us imagine an example in which two fiducial markers are proposing that the camera yaw is $175°$ and $-175°$: while reason might say that $180°$ is a good estimate, the linear Kalman filter would propose an estimation of 0°. A visual example of this issue is shown in Figure 82.

In order to overcome this issue, both the state space model and the Kalman filter proposed at Section 5.4.3 can be modified so that the angle is expressed as both its sinus and cosinus, keeping the same broadcastability approach for dealing with multiple markers. While these are bounded magnitudes too, they can be safely operated within a linear Kalman filter since its inner computations will never return a sinus or cosinus value outside of its $[0,1]$ range.

An additional simplification has been adopted in the new state space representation, by removing the first derivative of the state variables, the reasons behind this change being:

- Intuitively defining an angular speed in cosinus and sinus is complex.
- Reducing matrix sizes decreases the computational cost of the filter.
- The filter becomes simpler to tune because less variables are present.

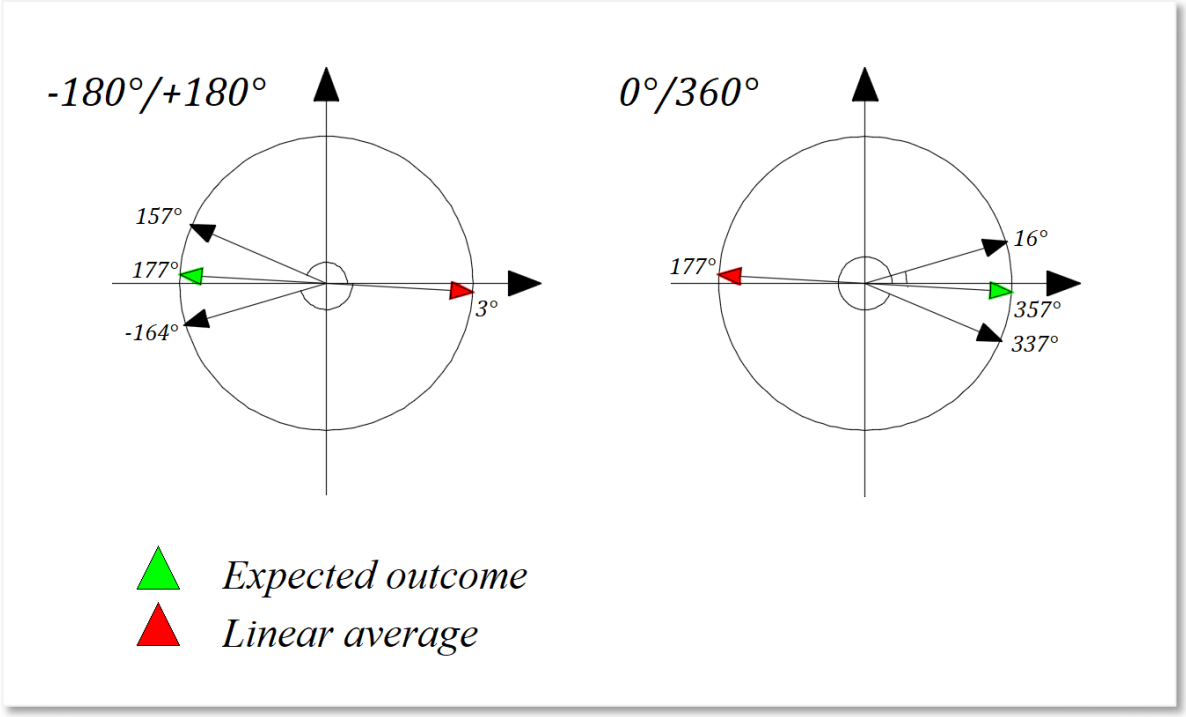


Figure 82: Issues with Kalman weighted average in different angle representations
Source: Self-made

After applying the stated modifications to address yaw issues, the state-space representation of choice for the localization problem of a camera based on ArUco markers, for an instant k in which a number of N ArUco markers are in sight with ids ranging from a to b, is defined as:

$$\begin{cases} \begin{pmatrix} x \\ z \\ \cos(\theta) \\ \sin(\theta) \end{pmatrix}_{k+1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}_k * \begin{pmatrix} x \\ z \\ \cos(\theta) \\ \sin(\theta) \end{pmatrix}_k + \boldsymbol{w}_k \\ \begin{pmatrix} \begin{Bmatrix} x \\ z \\ \cos(\theta) \\ \sin(\theta) \end{Bmatrix}_a \\ \vdots \\ \begin{Bmatrix} x \\ z \\ \cos(\theta) \\ \sin(\theta) \end{Bmatrix}_b \end{pmatrix}_k = \begin{pmatrix} \begin{Bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{Bmatrix}_a \\ \vdots \\ \begin{Bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{Bmatrix}_b \end{pmatrix}_k * \begin{pmatrix} x \\ z \\ \cos(\theta) \\ \sin(\theta) \end{pmatrix}_k + \boldsymbol{v}_k \\ \boldsymbol{w}_k \sim N(0, \boldsymbol{Q}_{4x4}) \; ; \quad \boldsymbol{v}_k \sim N\left(0, \begin{pmatrix} \boldsymbol{R}_{4x4} & \vdots & \boldsymbol{0} \\ \vdots & \ddots & \vdots \\ \boldsymbol{0} & \cdots & \boldsymbol{R}_{4x4} \end{pmatrix}_{4Nx4N}\right) \end{cases} \tag{20}$$

5.5 Item localization against camera coordinate frame

5.5.1 Estimating item position

By following the pinhole camera model described in Section 5.2.4, one can relate real-world points to image ones, provided an image is undistorted. Because we can find the camera matrix and its distortion parameters, any image taken by a camera can be artificially undistorted and then treated as a distortion-free image. These real-world points obtained are, indeed, referred to the camera.

In general, the pinhole model can be used to compute real-coordinates x, y from pixel points u, v, all of these coordinates measured in planes orthogonal to the camera axis. Estimating the z distance of a point to the camera (distance measured parallel to the camera axis) is a more challenging problem that requires external, additional information.

If the interest was not to get 3D coordinates of a single point, but an item, this problem can be solved with the tools at hand. Conveniently, bounding-boxes obtained by object detection pipelines easily provide us with u, v points of interest to find the contours of items. The distance of the item to the camera can be computed by one of the following methods:

- Using the width of the item as a reference for its distance in z.
- Using the area of the item as reference for its distance in z.
- Training a neural network to predict the distance based on the size of detections.

From which the most straightforward, yet less accurate, is the first one. In this case, we are only interested in a 2D localization of items, so only the x camera coordinate is of interest. Furthermore, let us assume that the widths of the items of interest are consistent and known. Figure 83 shows a depiction of this situation.

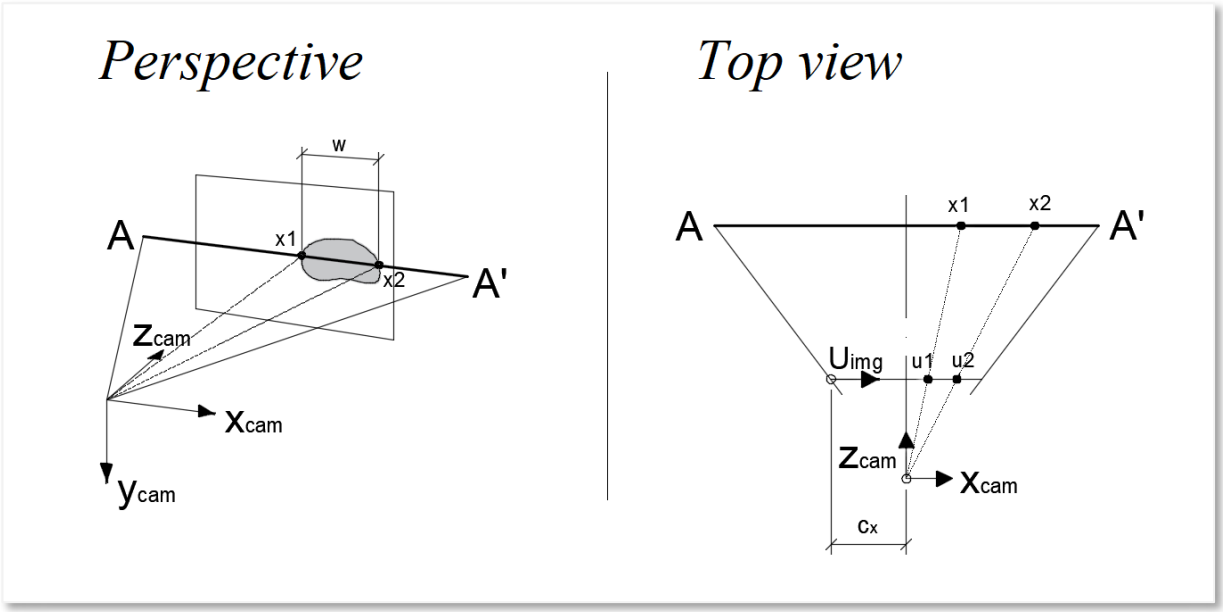


Figure 83: Finding an item's position known its width with the pinhole camera model
Source: Self-made

To compute the horizontal position x of this hypothetical item, we take just the first row of the simplified camera model for the two adjacent points of the item x_1, x_2 (or u_1, u_2 in image coordinates), both at the same distance from the camera z, from which we compute the scaling factor s, known the width of the item $w = x_2 - x_1$:

$$s\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} X_c \\ Y_c \\ Z_c \end{pmatrix} \rightarrow \begin{cases} s * u_1 = x_1 * f_x + c_x * z \\ s * u_2 = x_2 * f_x + c_x * z \end{cases} \quad (21.a)$$

$$s = f_x * \frac{x_2 - x_1}{u_2 - u_1}$$

From the top view in Figure 83, and taking into consideration that the axes x and u have different origins, we can arrive to an additional equation by applying the proportionality rule as:

$$\frac{u_1 - c_x}{u_2 - c_x} = \frac{x_1}{x_2} = \frac{x_1}{x_1 + w}$$

$$x_1 = \frac{\frac{u_1 - c_x}{u_2 - c_x} * (x_2 - x_1)}{1 - \frac{u_1 - c_x}{u_2 - c_x}} \quad (21.b)$$

And taking back that expression to any of the equations obtained from the pinhole model, we can solve for z and the middle point of the item x_{center} as:

$$z = \frac{s * u_1 - x_1 * f_x}{c_x} \quad (21.c)$$

$$x_{center} = x_1 + w/2$$

Arriving to the position of the item with respect to the camera. Optimally, this task would be solved by a convolutional network after the object detector, which could account for both the bounding box size and item orientation (inferred from visual features). For this work, however, width-based positioning is deemed enough due to limitations in time.

5.5.2 In-line object detection

The previous procedure assumes that a live object detector is running concurrently to propose bounding boxes wrapping items. The concerns about this stage are mostly related to implementation issues that will be later discussed in Section 6.

Functionality-wise, it is important to note that the bounding boxes drawn around items might include an unconsistent number of extra pixels in some bboxes. Also, when using video feed, not all frames might be consistent in the bounding box output. A perfectly visible item can sometimes appear undetected when the camera is moving because of a particular blurry movement or an unfortunate momentary beam of light. These phenomena, among others, represent an additional layer of noise characteristic of this particular method.

5.6 Item localization against real-world references

5.6.1 2D mapping as means of localization

As previously stated, the intent of this section is to solve the horizontal localization problem of items and camera. Horizontal localization means that we are only interested in the position of items in the 2D horizontal plane. With the tools presented so far, we accumulate random position errors from different sources:

- **Poor camera calibration**: Which impacts the localization consistency, depending on where the fiducial markers appear in the picture.
- **Camera pose estimation**: Which is a noisy measurement by default, plus some ArUcos can induce missplacements when seen from ambiguous perspectives.
- **Darknet output inconsistencies**: Since different bounding box configurations can appear in two consecutive frames of similar appearance. It can also happen that objects are only predicted intermittently.
- **Item to camera localization inconsistencies**: Which finds its major pitfall in depth estimation (z distance) since the width is estimated from an unstable bounding box.

Leading to very noisy item position placements, so that the instantaneous item localization output is bounded to be a noisy, intermittent magnitude. Towards mapping, it is necessary to express these noisy positions in an intuitive manner.

The tools described so far also fail to identify single instances of the items accurately, since the only means for doing this is position. Imagine we have two backpacks in an environment. The current pipeline can tell where they are (provided the object detector can identify each in a different bounding box) but does not perform tracking to tell which is which.

5.6.2 Likelihood maps, field of view and memory

These conditions suggest the usage of a soft discrete mapping that will not indicate exact item position of each instance of an item, but probability maps for the presence of each category. This is not a new concept in a localization context, as the idea is practically equal to the localization intuition shown in Figure 68. Traditionally, probability density functions have been used for this type of task, though a relaxation of the concept is desirable. PDFs must be computed symbolically (under a solid mathematical probability definition) or estimated from a histogram, both methods undesirable because of their additional computational cost. Also, PDFs must abide additional statistical constraints, such as the area under the curve being equal to 1, which enters in conflict with the idea of sharing a single localization map for many instances of the same category.

This is why we will relax the concept to that of a *likelihood map*, which is the term that we use in this work for essentially defining a custom defined function that, while not abiding to the constraints of a PDF, does share its intuition. A likelihood map will map a value indicating the likelihood that an item is placed at a certain position xz in the world coordinates via a likelihood function $\mathbb{R}^2 \rightarrow \mathbb{R}$, drawing a 2D map of the environment (or world) with the likelihood for the presence of an item at every single point in that world.

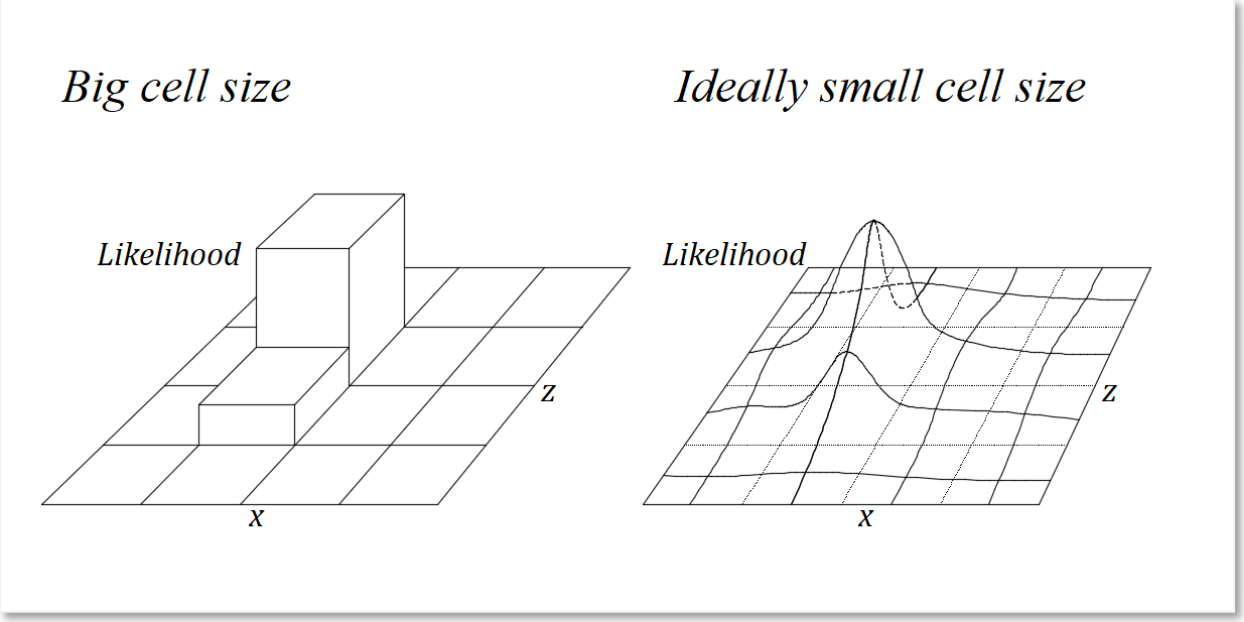


Figure 84: Intuitions behind the likelihood map concept. Discretization and cell size
Source: Self-made

Computation-wise, these likelihood maps will be computed in a discrete manner, by discretizing the 2D space in a number of *cells* with a given *size*, see Figure 84.

- High sizes, of the order of several decimeters, lead to a likelihood grid. Each grid in the 2D space will hold a scalar value. The higher this value is, the more likely that an item is in that cell.
- Small sizes, of the order of several centimeters, lead to an apparently continuous approach of the same concept. Even though this method is more computationally expensive, certain continuity is desired to allow for a smoother likelihood function. Small size cell size will be the standing assumption from now on.

A critical component of the localization system for this approach is the *field of view* of *FoV*, which is the portion of the 2D space (or map) that is visible to the camera. The camera can only perceive items – or absence of items within this field of view. Out of it, the likelihood map must remain frozen, as the camera cannot update that region. Thus, another critical element of this approach is *memory,* or the capability of holding information about a previous point in time. Within the field of view, the localization of items is to be done progressively and not at once, meaning that the camera will stare at an item for some time, having the likelihood increase progressively around the point where that instance is predicted to be, until a saturation value. At the same time, the likelihood of items in the whole field of view must decrease at all times to compensate for items that are no longer at a given position.

Implementation-wise, the likelihood function will be composed of the following elements:

- A field of view mask to decide which region of the map should be frozen.
- An increasing gaussian distribution for item detection.
- A decreasing uniform function for item obliviousness.

For convenience, matrix notation can be used for the definition of a likelihood map. In this matrix, each position i, j denotes a point in discrete 2D space, with an assigned value of likelihood $\boldsymbol{L}(i,j)$. The matrix size nxm of $\boldsymbol{L}$ denotes the overall size of the map. Under this definition and at an instant of time t, the likelihood map for the presence of a single class in the world $\boldsymbol{L}(t)$ is of size nxm. The very same concept can be broadcasted to a matrix with more channels, so that the likelihood map becomes a multi-channel matrix $\boldsymbol{L}_{nxmxc}$, where c is the number of item classes to be localized. For simplicity, further discussion will be made by assuming $c = 1$.

From this standing point, we define the *difference equation for the likelihood function* conditioning the evolution of a complete 2D likelihood map, expressed as the matrix $\boldsymbol{L}_{nxm}$, for the presence of a single class as the following matricial expression:

$$\begin{aligned} \boldsymbol{L}_{nxm}(t) \;=\; sat\,{}_{0}^{1}\,[\;\; & \boldsymbol{L}_{nxm}(t-1) && + \\ & - && \boldsymbol{FoV}_{nxm} \odot \boldsymbol{U}_{nxm} * OR \;+ \\ & + && \boldsymbol{FoV}_{nxm} \odot \sum_{a,b\ in\ dets} \boldsymbol{G}_{nxm}(a,b,s,\sigma) * DR \;\;] \end{aligned} \tag{22}$$

- $sat\,{}_{0}^{1}$: Denoting element-wise saturation with lower & upper bounds 0&1.
- $\odot$: Operator for the Hadamard or element-wise matrix product.
- nxm : Overall size of the likelihood map.
- $\boldsymbol{L}_{nxm}(i,j)$: Likelihood value for the presence of an item at the point i, j.
- $\boldsymbol{FoV}_{nxm}$: Boolean matrix denoting whether a point is within the FoV.
- $\boldsymbol{U}_{nxm}$: Uniform matrix of the same size as the map and filled with 1s.
- $a, b\ in\ dets$: Denoting the set of $i, j = a, b$ points where an item has been found.
- $\boldsymbol{G}_{nxm}(a,b,s,\sigma)$: Zero-mean gaussian kernel with size s and variance σ, padded with 0s so that the size of the matrix $\boldsymbol{G}$ is equal to nxm and the kernel is centered at the position i, j.
- OR : *Obliviousness rate*, factors how fast unseen items disappear.
- DR : *Detection rate*, factors how fast detected items appear in the map.

Figure 85 shows more graphical intuitions on the concept of likelihood maps, FoV and appearance/disappearance. This approach has the following advantages:

- **Dampens false positives/negatives from the object detection model**: In which detections can be location-inconsistent or intermittent in consecutive frames.
- **Dampens positioning accuracy in a smooth manner**: Reducing the impact of outliers due to positioning random noise or object detection inconsistencies.
- **Deals with not being able to track instances**: Allowing for likely location of objects to be displayed even if instances of items cannot be tracked.
- **Deals with moving objects**: Updating localizations in real time and accounting for items that were once discovered but have moved away from its original location.

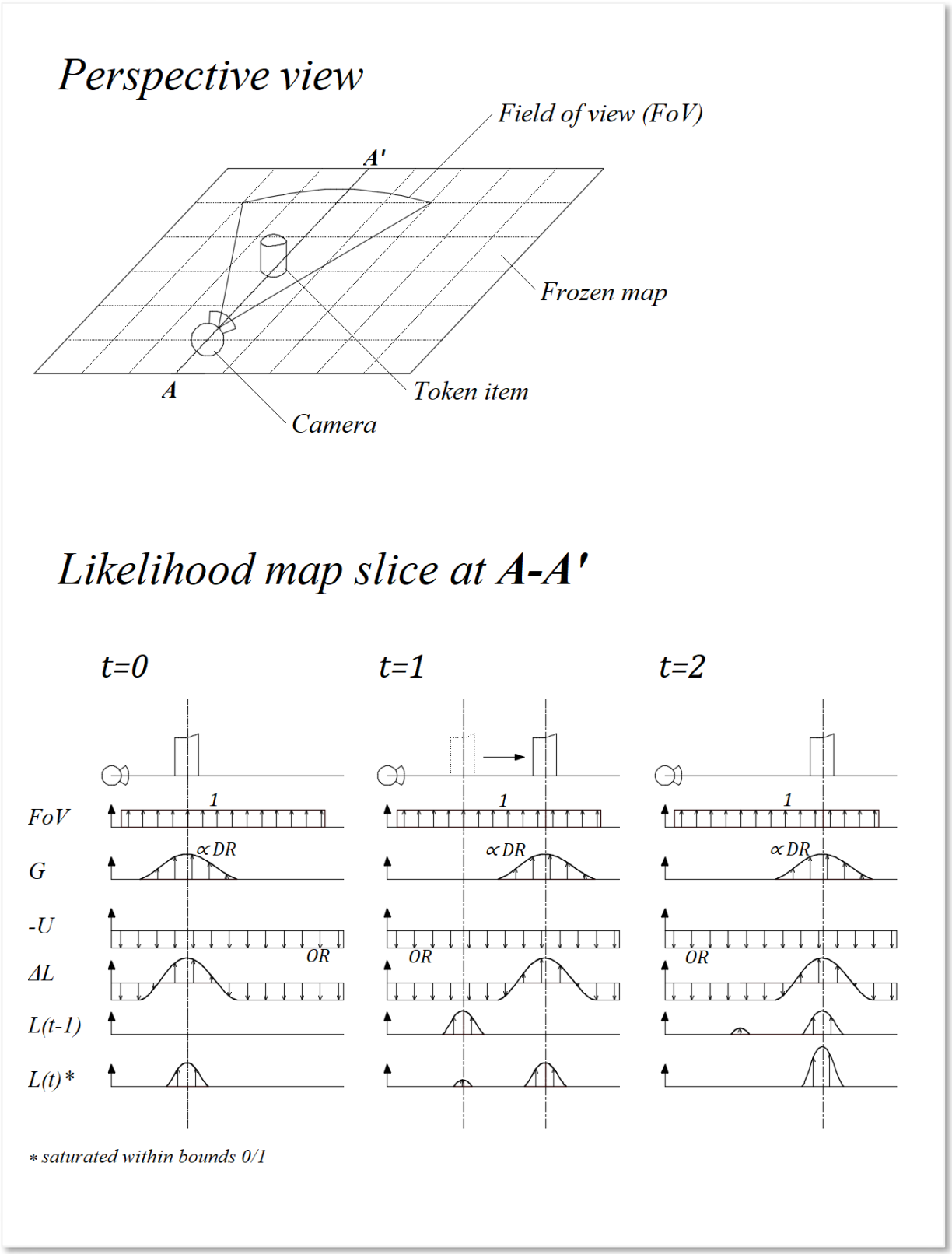


Figure 85: Intuitions behind the likelihood map concept. Appearance and disappearance
Source: Self-made

6 VISION-BASED ITEM LOCALIZATION PIPELINE

6.1 Introduction

6.1.1 Requirements and approach

This section aims to cover the implementation of a perception layer by which an agent (be it a robot, a person carrying a camera, etc.) can perform item search in a mapped environment solely on visual input. The perception layer is built around an object detection model that infers image item positions which are reconstructed to return real 2D coordinates in the horizontal plane. Item localization is to be performed by means of proposing regions on a top-view two-dimensional map of a known environment where the items of interest are likely to be present.

The overall strategy to follow is to concatenate the tasks described in Section 5, which address camera pose estimation and item localization, as well as 3D reconstruction (relaxed to 2D). The generated maps for the presence of items are based in the likelihood map concept depicted in Section 5.6.2, the implementation of which is approached with image-processing techniques that easily interact with the matricial approach adopted so far.

This perception pipeline is case-agnostic and can work with any Darknet-implemented object detector, as well as any set of weights for it, so that any detection model can be seamlessly incorporated to generate presence proposal maps. Towards experimentation and development, the object detector is chosen to be a YOLOv4-tiny pretrained on the MS COCO dataset, allowing for inexpensive testing by using varied household token items.

The implementation of this final tool is made on the Python programming language, by which data is processed to concatenate a series of blocks or modules that perform separate, isolated tasks, most of which run either on Python source code or Python bindings calling highly efficient C-written libraries or the separate deep learning framework Darknet. Computations are meant to take place in a remote computer or server, the agent sending visual information through WiFi. This section's experimental setup closely mirrors the one described in Section 5.1.2.

6.1.2 Computational issues

As stated, this pipeline is mainly code in the Python programming language. Python is a high-level interpreted language and, as such, it is straightforward to design, write and test script applications from simple text files. However, compared to compiled languages such as C, Python is notably slow. The intended solution is meant to perform in real time, meaning that both stability and high iteration speed are desired.

Most stability issues come from the addressing of exceptions throughout the program operation. Numerous tests were performed to identify possible unplanned situations from an application-specific point of view. Some examples of these are: No ArUcos being detected, unexpected ArUco ID detected, overflow on light-weight matrix variables (uint8), etc. The pipeline was designed so that these situations raise manageable exceptions to avoid crashes.

A simpler stability issue is the checking for memory leaks. The outcome software is divided across several modules and attempts to perform various tasks at once, all running inside an infinite loop that continuously feeds in new inputs. Memory leaks take place when new memory slots are occupied by the program, generally because of coding mistakes in statements that attemped to overwrite variables or instances. In this situation, the available RAM fills over time and the program ends up crashing when no memory is available. Checking for memory leaks is straightforward and can be done by normally running the program for a while and monitoring its performance over time with any process manager.

Regarding speed, there are numerous ways to speed up Python code, for example:

- **Lay heavy computations on specific libraries**: such as Numpy or OpenCV, which take computations to C source code, significantly boosting matricial operations and image processing speed. Compared to these structures, inbuilt iterables within Python such as sets and lists are notably slow to process. However, sometimes operations are small enought that the mere conversion from Python to other structures outweights the benefits of performing computations this way.
- **Broadcasting instead of looping**: Specially when working with array data types, such as numpy arrays, big time savings can be obtained by broadcasting such computations in wide arrays instead of looping shallower array operations.
- **Writing pythonic syntax**: Python is a language supposed to be written in a specific style. While this is enforced through the community for readability, some pythonic expressions such as list comprehensions are actually optimized and run faster than performing naive looping.

Further ways to speed up any type of code – not just Python, is recurring to parallelization methods. Concurrency and parallelization are both high-level terms for referring to the fact that two or more tasks are being performed at the same time. In general, computer programs can be classified as CPU-bound or I/O-bound, depending if the limiting factor regarding computational speed is CPU processing or I/O waiting times.

I/O-bound programs can be optimized by *multithreading*, for example, which aims to rearrange the execution of orders within a program so that the I/O waiting times are shared, thus decreasing computation time. CPU-bound problems, on the other hand, have as limiting factor raw computational power, and can only benefit from multiprocessing. *Multiprocessing* consists on generating child processes that can run on different CPU cores, effectively achieving parallelization in the sense that two computational pieces of the same program are actually running at once [67]. Specific tools need to be used to move data between different processes, such as queues or pipes.

6.2 Pipeline overview

6.2.1 Sequential approach

By joining the concepts depicted in Section 5 in a sequential manner, the desired pipeline can be assembled successfully. This implementation is performed in a single looped process in which all variables are shared by all modules, run sequentially as depicted in Figure 86.

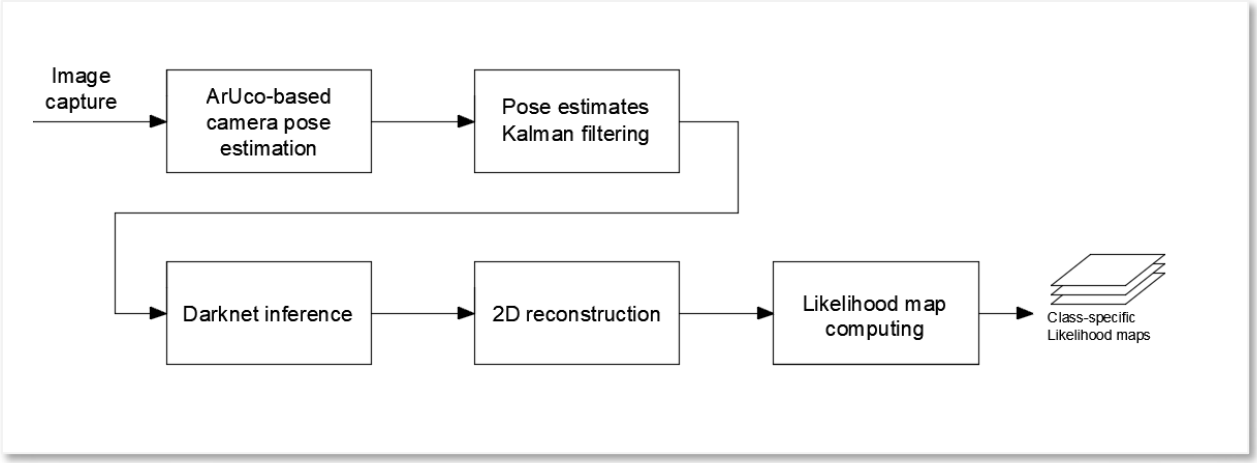


Figure 86: Visual perception layer for item localization. Sequential approach
Source: Self-made

In this approach, image capturing takes place in a remote camera device and is retrieved from a local server. This image input is first used to determine ArUco marker poses with respect to the camera, these poses reversed by pose algebra and then filtered by a Kalman filter. After these steps are performed, inference on the original picture is performed by calling a Darknet convolutional model, then the bounding boxes converted to real 2D coordinates by width-based distance estimation. The camera pose and the position of the items relative to the camera are then composed and smoothed, then likelihood maps are computed to provide estimates of the localization of the items.

While effective, this initial implementation suffered severe performance issues on the hardware described in Figure 43. While almost real-time framerate was achieved (around 10 FPS) the time difference between the recording of a frame and its resolution and mapping on-screen was of about 3 to 5 seconds. This issue was found to be worse when more ArUcos appeared within reach, due to both the ArUco detection system and, especially, the naive Kalman implementation which quickly becomes more computationally expensive the more fiducial markers being filtered.

6.2.2 Parallelized approach

The sequential model depicted in the previous section was seen to clearly belong to a CPU-bound problem. It was the high computational demands of the localization pipeline what was dragging the system behind in time. To improve its performance, the same steps are converted to a multiprocess program that allows splitting the different tasks in separate CPU cores and have them run at the same time.

Each separate task is to be run within its own process, including strictly-sequential tasks such as pose estimation and Kalman filtering. Even if the Kalman filter needs pose estimation, this implementation allows for the pose estimation of the $k + 1$ frame to be performed at the same time the Kalman filtering of the k frame is taking place.

The whole parallelized pipeline is shown in Figure 87, in a more detail than its sequential counterpart so that all computational modules and queues are clearly identified.

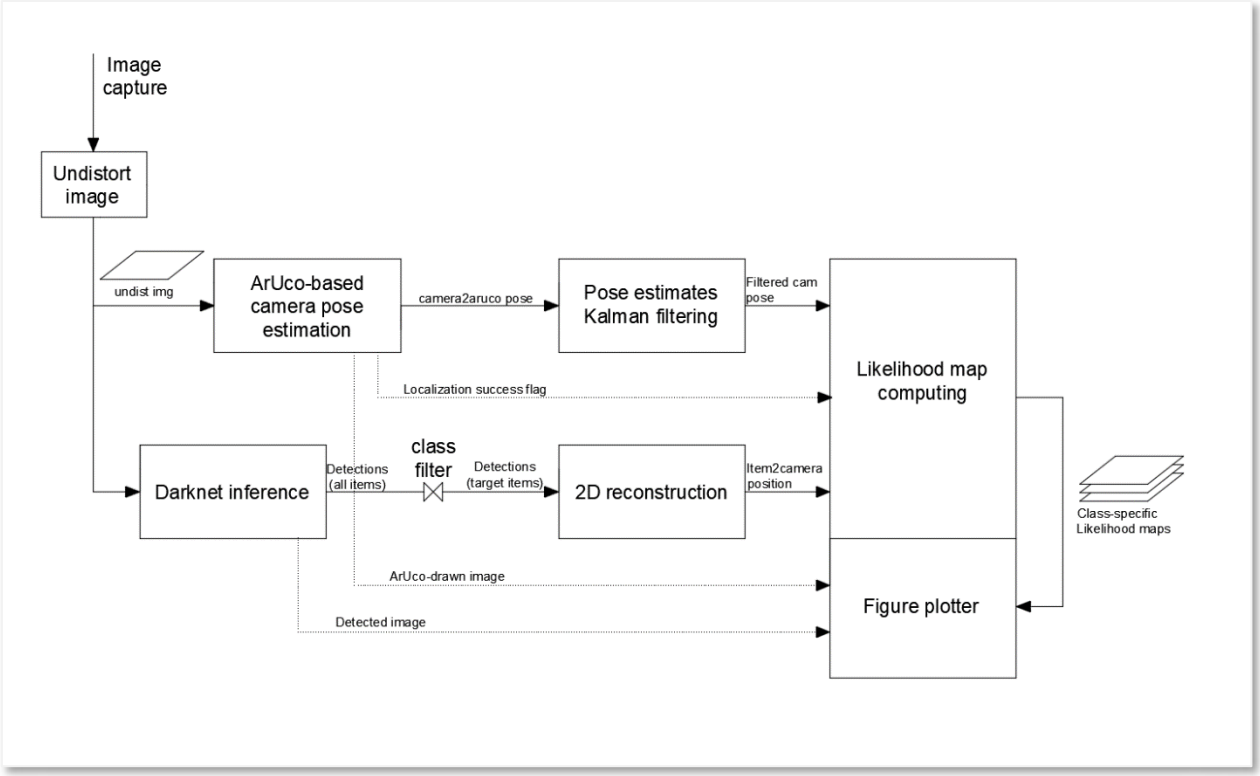


Figure 87: Visual perception layer for item localization. Parallelized approach
Source: Self-made

In the parallelized approach, modules share information with queues. A *queue* is a type of buffer within the multiprocessing Python library which acts as communicator between two or more processes. Queues receive items and store them in a FIFO manner until they are asked for. Queue interaction is a blocking process, meaning that if a statement attempts to retrieve an item from a queue while this is empty, the program will generally pause at that statement until the queue receives some retrievable item, then continue. Queues can be instanced with a maximum size. To guarantee that, after parallelism, the outcomes of parallel detection and pose estimation belong to the same instant of time, queues are set to have a maximum size of 1 element.

The parallel pipeline is split across the following processes:

- Process 0: Boots all other processes then loops image capturing and undistortion.
- Process 1: Boots ArUco pose estimation then loops while inferring input images.
- Process 2: Boots the Kalman filter then loops while real-time filtering.
- Process 3: Boots Darknet then loops inference of input images.
- Process 4: Boots the class filter plus then loops while performing 2D reconstruction.
- Process 5: Boots the likelihood map and loops by updating it in real-time.

Besides the main data queues that link the different modules, three additional auxiliar queues are used for handling exceptions and displaying results.

- A localization success flag prevents the map from updating if agent localization fails.
- Two additional queues move the images with ArUco and object detection results to the final module for plotting, since OpenCV does not support multiprocess display.

6.3 Module-specific implementation aspects

6.3.1 Process 0. Image capture & undistort

The process 0 (*P0*), or main module, holds the program initialization routine. As such, the program is loaded by running the script holding its code. As stated, the image capture is performed from a remote camera device within the local WiFi network, typically a smartphone.

Some preparatory tasks attended in Process 0 are:

- Instancing the frame capture device from a remote IP address.
- Loading the camera calibrated parameters from a local file.
- Defining the ArUco map of the environment, with coordinates and ids of each one.
- Defining the list of items to be detected and their expected width.

Once these tasks are over, this process loops the following actions:

- Load a frame from the video feed.
- Undistort said frame using the camera parameters.
- Feed undistorted images down to processes P1 and P3.

In the parallelized approach, undistortion has been moved to the beginning of the pipeline for tidiness. Undistorted images are required for ArUco pose estimation (P1) and 2D reconstruction (P4), so this step is performed here. Distortion should not affect the efficacy of the image detector, since any state-of-the-art model is trained for numerous items and with strong data augmentation techniques. Because the ArUco detection bult-in functions require the input of a distortion vector, a vector with infinitesimal values is generated as a dummy distortion vector to feed this function with.

6.3.2 Process 1. Camera pose estimation

This process holds the camera pose estimation with respect to multiple ArUco markers. Form these ArUco markers, the axes configuration of the one with $ID0$ is stated to the world coordinate frame, acting as global reference for the pose of the camera.

Initialization tasks in P1:

- Defining the ArUco dictionary and parameters to be used.
- Stating which ArUcos inside the dictionary are actually in the environment.
- Stating the size of ArUco markers.

Looping tasks in P1:

- Get undistorted input frame from P0.
- Attempt to find ArUcos in that frame.
- Compute the pose of the camera with respect to each ArUco.
- Apply a 2D transformation to obtain multiple poses of the camera with respect to a world reference (ArUco with $ID0$), computed from each marker's measurement.
- Send the pose collection to the Kalman filter in P2.

This process can run into several exceptions. First, let's define the concept of ArUco dictionary. An ArUco dictionary is a collection of ArUco markers with a given binary matrix size ($4x4, 5x5, etc.$). Dictionaries are usually generated with a fixed number of output markers, so that, known that we want N markers, these will be generated as differently as possible. For experimentation and development, however, it is more convenient to recur to a big standard dictionary, and place or remove certain physical markers from the environment from a collection that we only print once. This means that, in practice, the ArUco detector function can detect more ArUcos than the ones we already have (and account for) in our experimental setting.

The exceptions that have been accounted for are:

- No ArUco markers are shown:
  - At any time: Fail localization and send in the previous pose estimations.
  - At the first iteration: Fail localization and send fake coordinate detections.
- Unexpected ArUco marker:
  - Filter out any marker whose ID is not stated in the markers dictionary at P0.

When any exception that prevents localization from being made happens, a flag is passed to the mapping module at P5 so that the – unreliable – presence of items is not recorded by freezing the map for the failed iteration. It is interesting to keep the program running so that the camera view for object detections keeps working in real time.

6.3.3 Process 2. Pose Kalman filtering

This stage first imports a hardcoded model of the state space model depicted in Section 5.4.4 and the expandable Kalman filter implementation as defined in 5.4.3. The broadcasting of the filter's inner matrices is performed within the update function inside the filter according the same guidelines as before.

No exceptions are expected to take place at this stage since localization failures are already addressed in P1. In case of localization failure, this process simply performs the same computations on dummy values.

6.3.4 Process 3. Object detection with Darknet

Object detection is carried externally within Darknet. However, default Darknet interface poses several difficulties for live inference. With the commands shown at the tutorial at [50] live inference can be performed but no numerical information extracted in real time. The way of obtaining numerical bounding boxes according to [50] requires to boot up the network every iteration, completely inadmissible towards reaching real time. In this implementation, Darknet is called from the python bindings available at [34] to overcome this issue.

The initialization tasks comprise:

- Importing Darknet with GPU support.
- Initializing the network of choice.

To then indefinitely run inference on the undistorted image taken from P0.

While Darknet models can be imported to OpenCV to perform inference with its high-level API, the available installation of OpenCV did not support GPU acceleration at the time. Importing Darknet properly can be an issue depending on how the instalation is made. Provided the GPU support is set up adequately (CUDA, CUDNN installed), Darknet can be imported from the Python environment by navigating to the directory holding its installation and importing the file *darknet.py* without requiring the installation of a Python-specific darknet library via pip.

6.3.5 Process 4. 2D reconstruction

2D reconstruction is a relaxation from the concept of 3D reconstruction which aims to relate image data to real world coordinates. This module simply attempts to estimate the horizontal position of an item with respect to a camera, based on its width within an undistorted image, according to the procedure described in Section 5.5.

Because not all items in the object detection model may be of interest, especially when using pretrained weights of a problem as big as the MS COCO dataset, a filter is placed at this stage so no instances of undesired items are translated into coordinates.

6.3.6 Process 5. World mapper

The world mapper draws a map of the environment in which the likelihood map for the presence of items is later displayed. So far, the world coordinate frame has been placed at the ArUco marker of $ID0$. Since we approach the drawing of the likelihood maps with image processing, a coordinate change is needed so that the map coordinate frame belongs to the top left corner of the output likelihood map matrix.

This module initializes by:

- Instancing the world map, defining its overall size.
- Defining the items that are to be drawn – particularly how many classes are there.
- Setting values for the discovery and obliviousness rate.
- Defining the coordinates of the world coordinate frame with respect to the map coordinate frame.

Conventionally, the map coordinate frame is set to be square to the world coordinate frame so that the last step can be performed with simple addition. During continuous deployment, this process loops the following tasks:

- Receive filtered camera pose and item positions with respect to camera from P2, P4.
- Rearrange those measurements to make the different coordinate systems compliant.
- Refer all coordinates to the map coordinate frame.
- Compute field of view and instantaneous discovery gaussian kernels.
- Update the current likelihood map according to the Equation (22).
- Receive image output from ArUco detections and Darknet inference.
- Plot these images and the likelihood maps.

As stated, all images are sent to the last module for plotting since the current installation of OpenCV does not support GUI graphics generated by different processes.

Because the likelihood maps are implemented as grayscale image channels, the likelihood concept is remapped to have a value in the range $[0, 255]$ for display.

The field of view is modelled by a set of three points ABC relative to the camera, drawing a isosceles triangle up to a distance L and with a specific aperture angle α. This angle is found first analytically from the camera matrix then reduced to a round value. The value for L is chosen arbitrarily, and is used to model a limit in distance sight in very big environments.

The gaussian kernels placed for localization are defined by three parameters:

- Standard deviation: Which defines the width of the gaussian bell.
- Matrix size: Which states the number of discrete values that will represent the kernel.
- Peak value: The kernel is scaled so its peak a value is 250 for easily tuning the DR.

Namely, the coordinate frames of relevance to this pipeline are:

- The camera frame $\{C\}$: Defined by the pinhole camera model.
- The world frame $\{W\}$: Defined by the axes of the ArUco marker with $ID0$.
- The map frame $\{M\}$: Defined by the top-left corner of the map image.

Towards drawing the likelihood maps, the need for coordinate consistency has to be addressed because the different modules use different coordinate frames, which can be modified to make computations simpler. These frame modifications are:

- Invert the camera's x axis, initially defined by the pinhole camera model.
- Redefine the yaw definition, initially defined at the ArUco pose estimation stage.
- Redefine the map frame as the common reference rather than the world frame.

After these transformations are made, a point P related to the camera can be drawn in the map by simply applying the following homogeneous transformation, C denoting the camera frame and M denoting the map's frame, provided that $\{W\}$ and $\{M\}$ are strictly square:

$$\begin{pmatrix} {}^{M}z_{P} \\ {}^{M}x_{P} \\ 1 \end{pmatrix} = \begin{pmatrix} cos\,\theta & -sin\,\theta & {}^{M}z_{W} + {}^{W}z_{C} \\ sin\,\theta & cos\,\theta & {}^{M}x_{W} + {}^{W}x_{C} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} {}^{C}z_{P} \\ {}^{C}x_{P} \\ 1 \end{pmatrix} \tag{23}$$

- ${}^{M}z_{P}, {}^{M}x_{P}$: Being the coordinates of a point in the output likelihood map,
- ${}^{C}z_{P}, {}^{C}x_{P}$: the horizontal coordinates of the same point with respect to a camera,
- θ : the yaw of the camera with respect to $\{W\}$ and $\{M\}$.
- ${}^{M}z_{W}, {}^{M}x_{W}$: Denoting the horizontal position of $\{M\}$ with respect to $\{M\}$.
- ${}^{W}z_{C}, {}^{W}x_{C}$: Denoting the horizontal position of $\{C\}$ with respect to $\{W\}$.

From which the item positions and the points ABC defining the field of view are converted to map coordinates. Figure 88 visually shows said modifications so that the different coordinate frames are consistent towards this simplified expression.

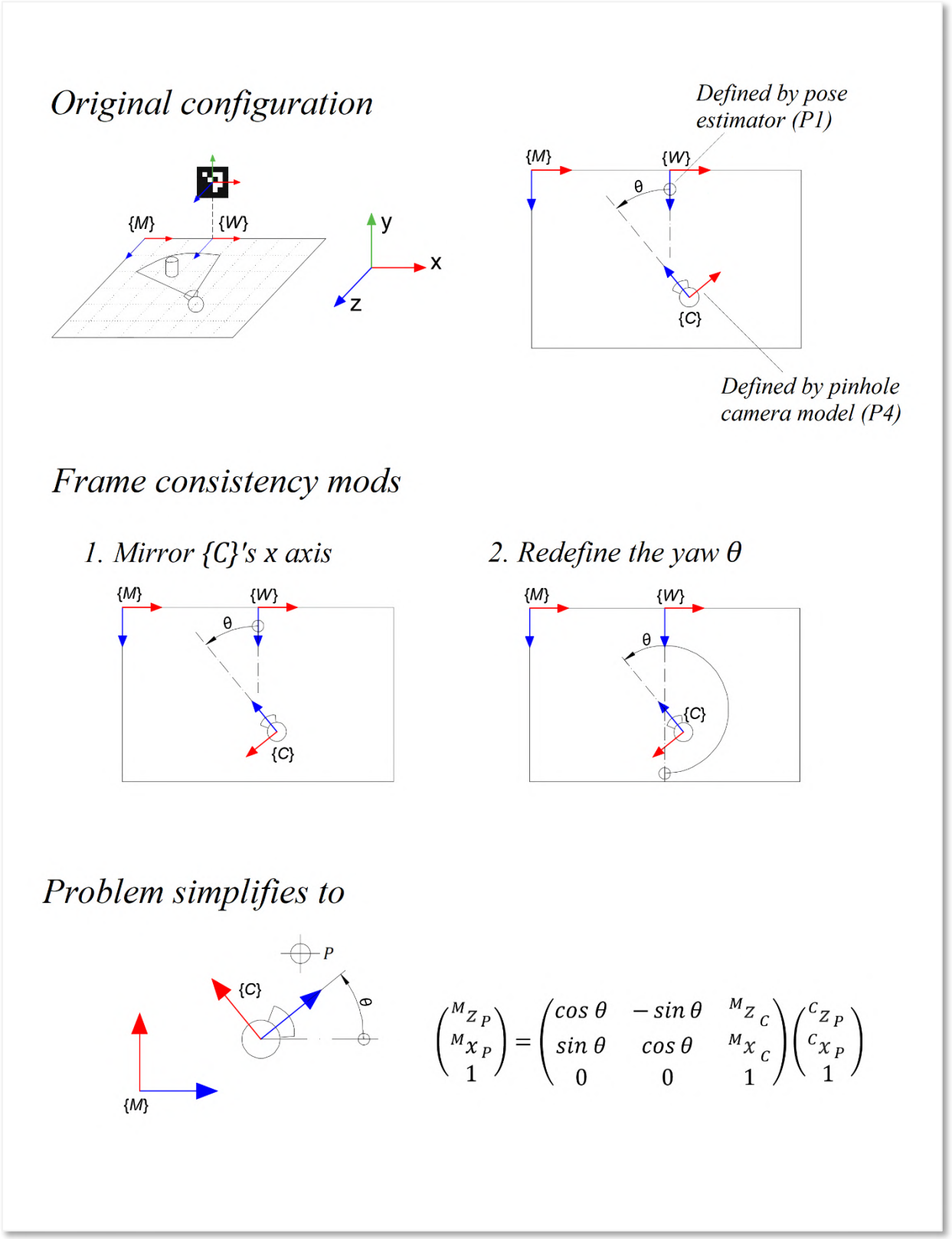


Figure 88: Frame consistency modifications towards straightforward pose composition
Source: Self-made

6.4 Results

6.4.1 Experimental setup

The hardware and software tools used towards the testing and development of this solution mostly mirror those defined in Section 5.1.2. Extra utilities used in this stage are:

- A smartphone charger with a 5m extension.
- An external TV for monitoring when away from the computer's main display.
- A 10m HDMI wire.

For test-supported development and bug fixing, a full layout of ArUcos is placed in an indoors room, with the aims of testing camera localization and item localization for the position of arbitrary token items.

- **House living room:**
  - Used for final tests and bugfixing.
  - Away from the computer's location.
  - Comprises ten ArUco markers allowing for 360° localization.
  - Marker layout visible in Figure 89.

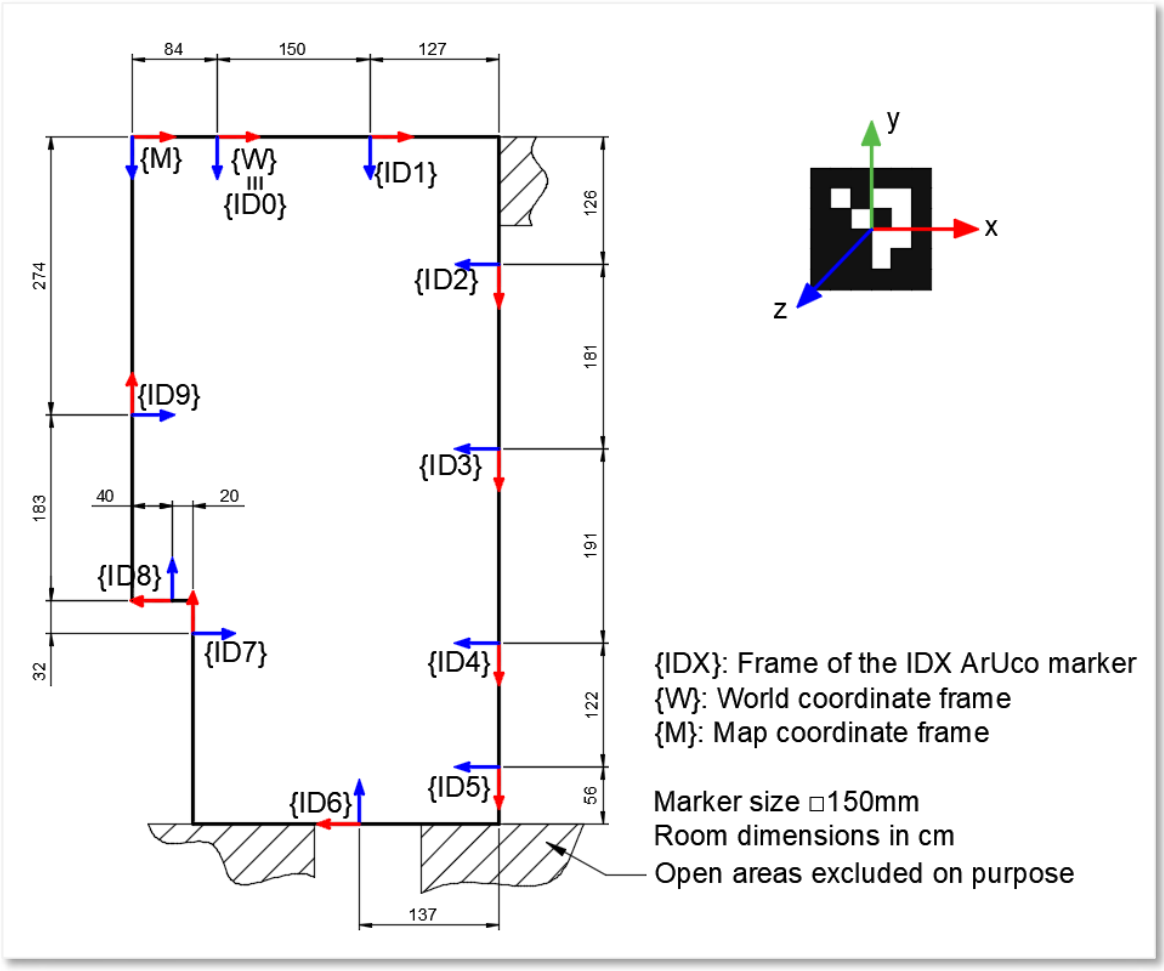


Figure 89: House living room map ArUco layout
Source: Self-made

6.4.2 Live test on House living room map

Live tests were carried out as final testing for the detection of bottles, by using the pretrained MS COCO weights for a YOLOv4-tiny. Among many experimentally supported development and bugfixing, a final scripted experiment was carried out to showcase the performance of the detection pipeline, a video of which can be found under this project's repository at [46], in which the following tasks were performed:

- The camera, mounted on a tripod, and two bottles are placed in the environment:
  - Initially, the camera cannot see both bottles simultaneously.
  - The camera cannot see any of the bottles when the experiment starts.
- The camera starts moving in clockwise direction slowly and steadily.
  - The detection of the bottle 1 takes place while the camera moves past it.
  - The detection of the bottle 2 takes place while the camera moves past it.
- The camera is rested in a position where no bottles are seen.
  - Bottle 2 is manually moved and positioned in sight of the camera.
  - During this movement, noise is recorded around of the FoV.
- The camera is slightly moved around to get rid of the recently generated noise.
- The camera is moved in clockwise direction up to the old location of bottle 2.
  - The camera is stopped when this location is within the FoV.
  - The likelihood value decreases in the spot where bottle 2 is no longer placed.

Figure 90, Figure 91, Figure 92, Figure 93, Figure 94, Figure 95, Figure 96 and Figure 97 show a camera view plus its extracted likelihood map at an different moments in time within this experiment. From this setting, let's focus on the following points:

- Within the likelihood map:
  - The shown likelihood map represents the environment shown in Figure 89.
  - The thick hollow circle represents the position of the camera.
  - The triangle shape represents the camera's field of view at that instant.
  - Two white blobs denote estimates of the locations of both bottles.
- Within the object detector view:
  - Many different items are being detected, due to using an MS COCO model.
  - Most of these are filtered out so that only the bottles are located.

As stated, the very same setup is used for experimentally supported development in the later stages of implementation. Among others, live experiments in this map allowed performing the two following tuning tasks:

- Manual tuning of the Kalman filter so random pose noise is reduced, by redefining the state space model noise covariances so that the inertia of the signals is enough to signifincantly damp the high variance $xz\theta$ measurements.
- Discovering and addressing the issue leading to the linear Kalman filter to output poor yaw angle estimations described at Section 5.4.4, redesigning the state space model and validating it experimentally (requires a $360°$ ArUco layout).

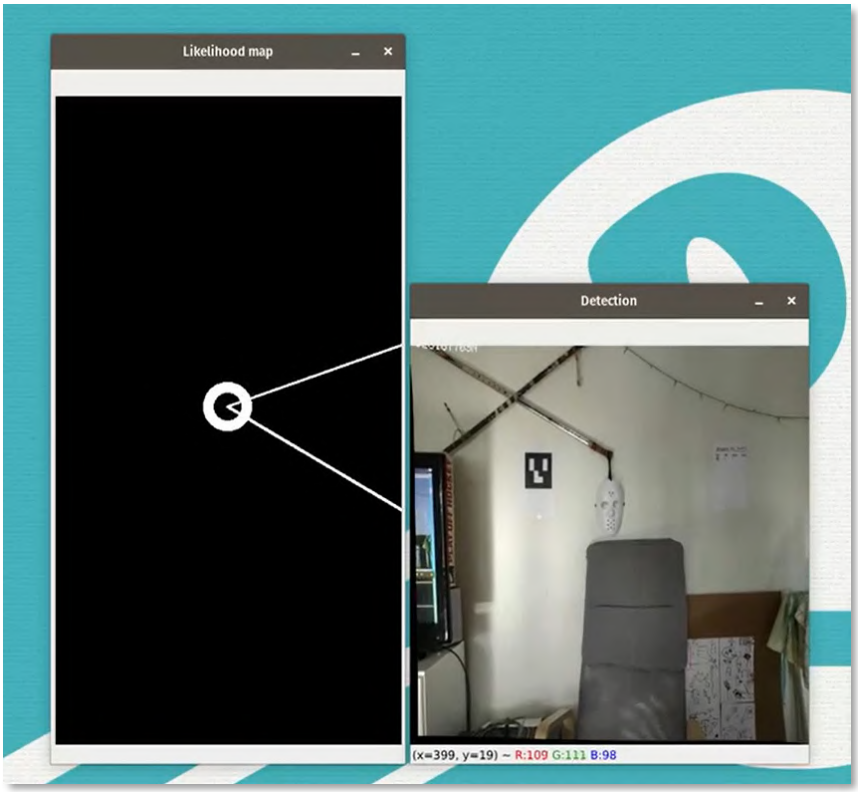


Figure 90: Demonstration: localization of bottles within the house living room map I
Source: Self-made

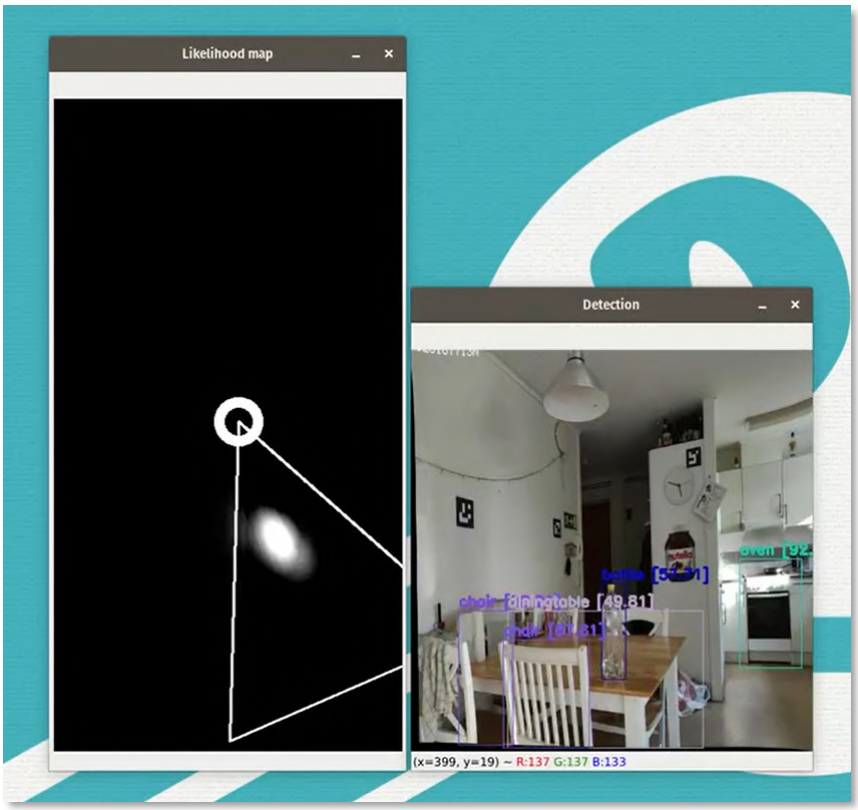


Figure 91: Demonstration: localization of bottles within the house living room map II
Source: Self-made

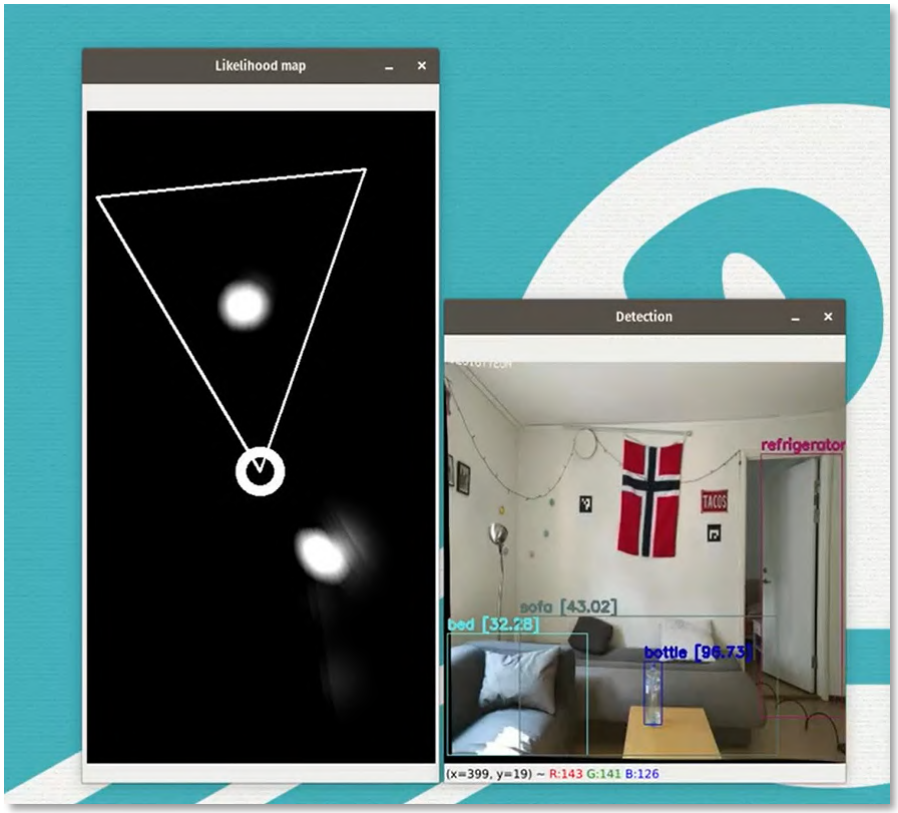


Figure 92: Demonstration: localization of bottles within the house living room map III
Source: Self-made

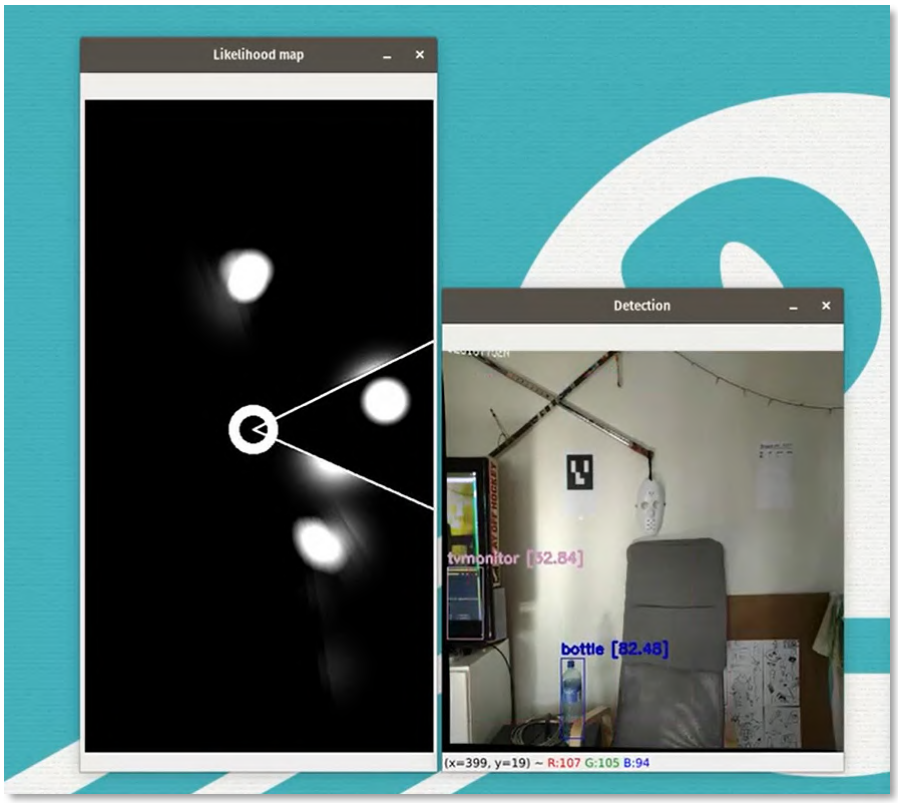


Figure 93: Demonstration: localization of bottles within the house living room map IV
Source: Self-made

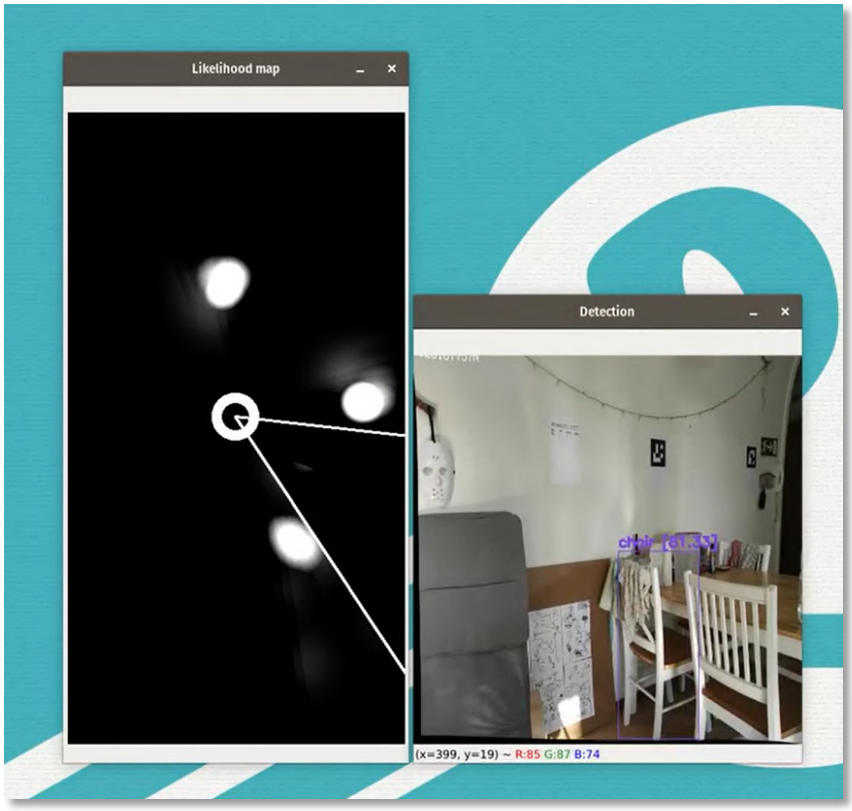


Figure 94: Demonstration: localization of bottles within the house living room map V
Source: Self-made

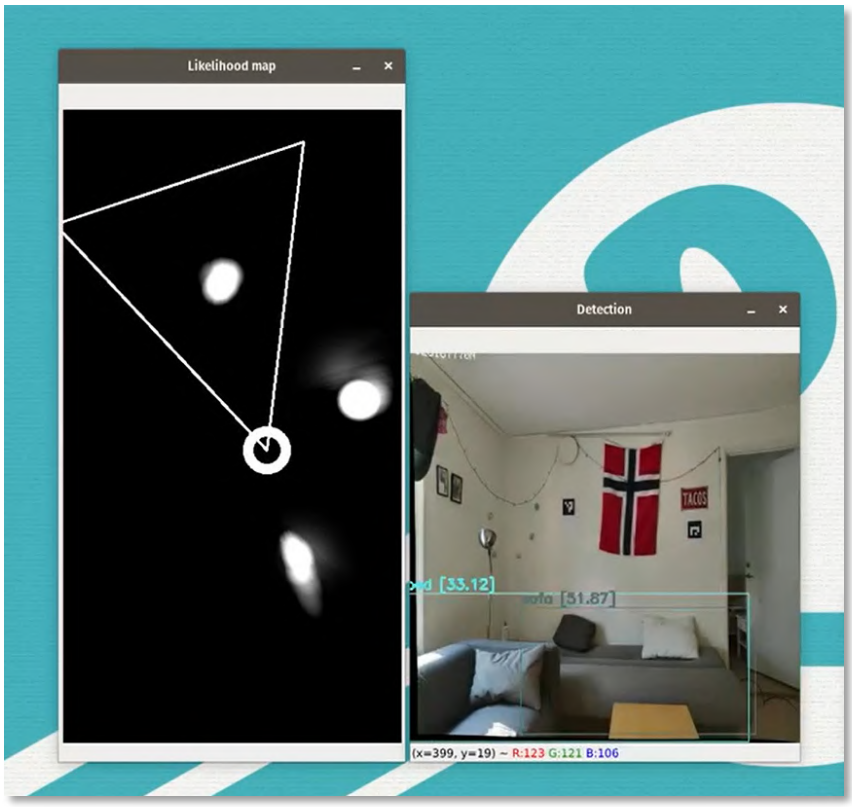


Figure 95: Demonstration: localization of bottles within the house living room map VI
Source: Self-made

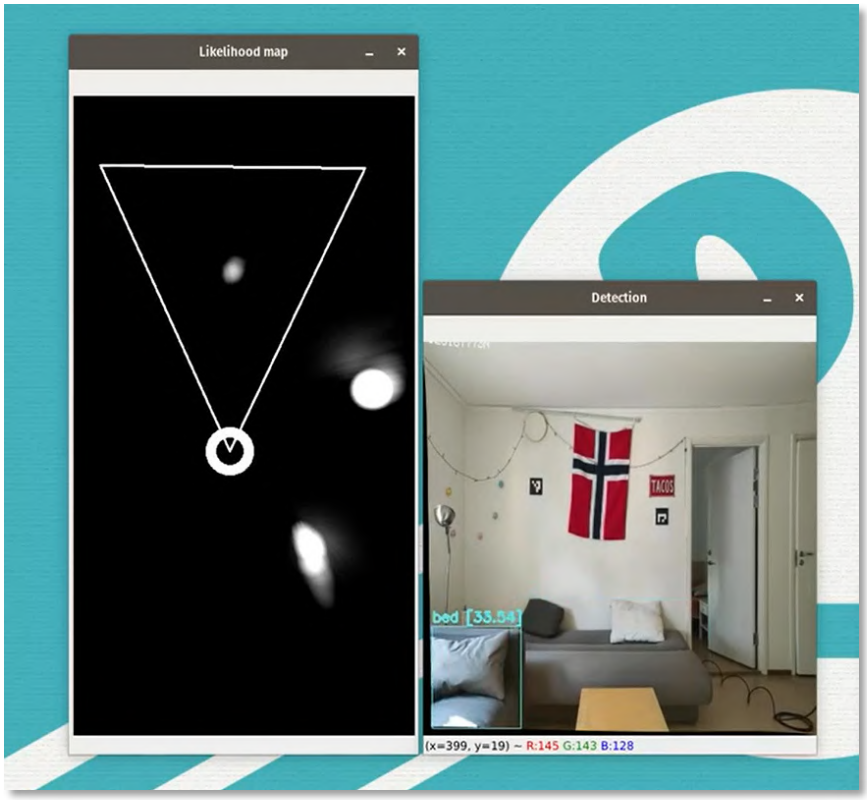


Figure 96: Demonstration: localization of bottles within the house living room map VII
Source: Self-made

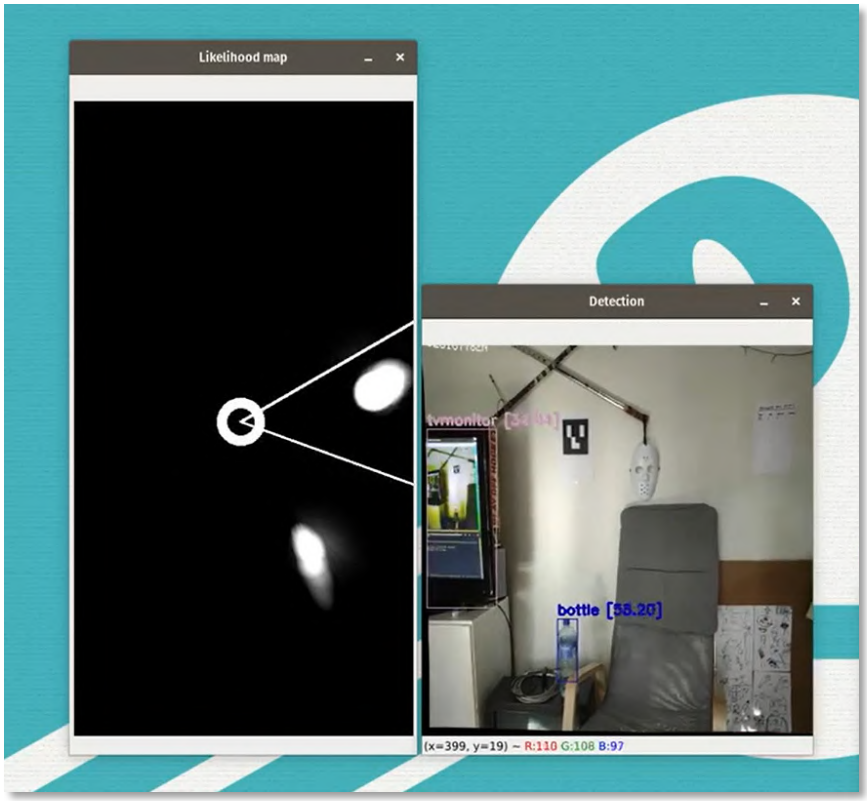


Figure 97: Demonstration: localization of bottles within the house living room map VIII
Source: Self-made

6.4.3 Discussion

The detection pipeline is built in Python in a multiprocess implementation, which is easy to maintain and upgrade. In the available hardware (Figure 43), the pipeline is seen to consistently run at >20 FPS under a stable WiFi network. A reimplementation of the software in a lower-level language might be opportune to increase inference speed, though multiple modifications over the existing one would still allow for performance increase without requiring a different language. Porting the proposed perception layer on any autonomous application is straightforward, and the only required task is to assign the detected item locations to the control and path planning layers.

Agent localization is approached by fiducial markers, returning a significantly noisy pose estimation, specially in orientation, issue addressed by the implementation of a Kalman filter to denoise a variable number of pose readings. The Kalman filter is tuned to prioritize the denoising of the yaw signal, the errors in which are more critical since they propagate more significantly when localizing items far away from the camera. A detailed quantative analysis of localization accuracy in the current experimental setup would not be reliable, main reasons being the imperfect camera calibration and the suboptimal planicity of the fiducial markers, both problems originated from the unavailability of rigid high-quality materials for holding a big-size calibration pattern and the ArUco markers themselves. These make measurement quality highly dependent on the particular orientation of the camera for the given experimental setting. Figure 98 exemplifies a wrong reading of displacement when the camera is rotated yet not displaced because of these issues.

Qualitatively, live testing has shown that the combination of this suboptimal setup with the implemented Kalman filter can identify the position of the camera with an estimated position error in the order of $\pm 50cm$ in a closed environment, based on the visuals of ArUco markers of a square size of $150mm$ at an average distance of $3m$, and by using a camera resolution of $800x600$ pixels. Yaw accuracy is harder to estimate with the available resources, though after proper Kalman filter tuning the yaw measurements are seen to steadily converge to a stable and apparently correct value when holding the camera still. With a clear perspective on ArUco markers, 2D pose estimation measurements become very robust, holding true even if modifying the roll of the camera, as shown in Figure 99.

Opposed to agent localization, item localization is addressed as a pseudo-probabilistic problem, where absolute locations are not explicitly defined but likelihood maps, in which a potential field describes which locations of the 2D space are more likely to contain instances of items of certain categories. Because the map is updated in real time and meant to correct itself at each iteration, any instantaneous random localization error will be corrected over time, thus achieving robust position estimations from many weak item and position readings. This approach fits nicely with the limitations of object detection in real time such as a) detection intermittency b) unstable bounding boxes and c) unability to reliably discern different instances of the same item. Because of this flexibility, object detection accuracy is not a critical factor, and mediocre but fast object detectors can produce reliable localizations. While this work proposes asynchronous computations by sending data over WiFi to a remote powerful computing machine, this is an interesting point towards the design of an on-board light-weight solution.

Figure 98: Missread displacement when modifying the camera's pitch in place
Source: Self-made

Robustness to detection inaccuracies is also of interest because it allows for the use of general models for the detections of some smaller set of items. As an example, the performed tests aimed to identify a single class with an MS COCO YOLOv4-tiny, which is ready to identify 80 different classes, yet only offering an accuracy of 21.7% mAP (see Figure 25), which proves enough for the mapping of the bottles in the room map. Another advantage of this pipeline is that it is completely agnostic to the used detector and weights, meaning that any available Darknet model can replace the one proposed in this work. Platforms other than Darknet can also be used by simply swapping the detection script for that of a different tool, as long as the detected bounding box output format is kept.

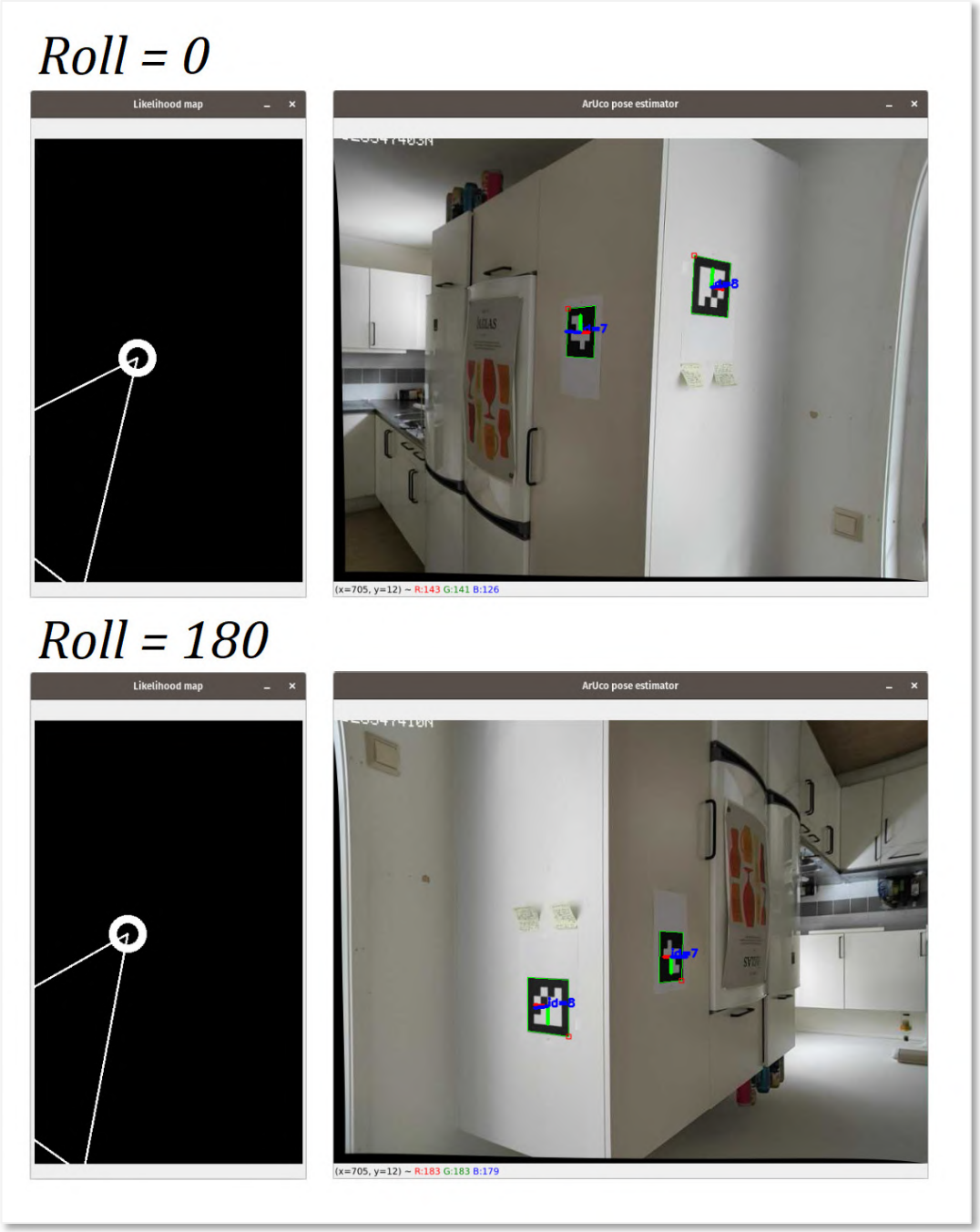


Figure 99: Pose estimation robustness to camera roll variations
Source: Self-made

In its current state, the pipeline is seen to effectively propose regions of reasonable radius that are highly likely to contain the searched items in a robust manner as proven in the tests. The accuracy of these regions is not deemed critical at this point in development, since wide region proposals are deemed fit for coarse item search if the area where an item is found to be is open, so that the target can be clearly identified from image and a large likelihood map blob. Also, high accuracy becomes meaningless if the item moves at a speed high enough to make its accurate positioning irrelevant. An inmediate improvement towards boosting accuracy would be to use a higher input image resolution.

7 CONCLUSIONS AND FUTURE LINES OF WORK

7.1 Conclusions

This work is concluded after performing experimental study and comparison of state-of-the-art object detection models and building a perception layer suitable for autonomous item search around the object detection paradigm. The main outcomes of this project can be listed as:

- Two collections of data towards the DARPA Challenge items:
  - PPU-6: Including newly-generated labelled helmet and rope examples.
  - Unity-6-1000: Including 1000 labelled synthetic pictures of all artifacts.
- A batch of trained object detection models with competitive detection performance.
- An analysis of the previous models towards defining the most competitive detectors for a visual approach to the DARPA SubT Challenge.
- A perception layer for mapping item positions to a 2D map solely on visual input.

The trained models, datasets and source code for the perception layer can be found at this project's repository at [46], among other resources. Data production outcomes are addressed in Section 3.7, while benchmarking results are summarized in Section 4.4. Building blocks towards the perception layer are addressed in Section 5, its implementation showcased in Section 6.4.

It is found that object detection problems comprising few number of classes do not require heavy models to achieve reliable detection accuracies. After numerous experiments, YOLOv4-tiny is found to be the best model towards the DARPA SubT Challenge based on an accuracy-speed tradeoff. An amount of 200 instances per class of training data was found suitable to train the studied models. Though no conclusive results were found for the combination of real plus synthetic data in the studied setting, additional experiments have proven that the produced SynthDet samples are representative for training and suggest that detectors can be produced without the need of handlabelling data at all.

The perception layer is built in a modular, multiprocessing implementation, allowing for visual item search to be performed in real time by using a high-end remote computing machine. In this approach, a camera navigating an environment uses image feed to both localize itself and the pursued items in the environment. The items are drawn in a 2D likelihood map which highlights areas of the known environment where they are more likely to be present. The most relevant strengths of this approach are its robustness to camera pose and object detection noise, its agnosticity to the used object detection method, and its modularity and ease of incorporation to autonomous devices.

The stated visual relaxation of the DARPA SubT Challenge was addressed by the production of case-specific data and the experimental benchmarking of a number of state-of-the-art object detection models, while visual item and agent localization were achieved and combined in an integrated probabilistic real-time perception layer suitable for item search, fullfilling the goals defined at Section 1.2.

7.2 Future lines of work

7.2.1 Further experimentation with synthetic data for object detection training

This work has proven that object detection model can effectively learn features from synthetic examples, even when disregarding specific curriculum-based training methods such as the one proposed in [44]. As stated in Section 4.4.4, models trained exclusively on SynthDet generated data show a high number of false positives because they do not generalize well to realistic background environments. This suggests that a mixture of training examples composed of artificial images plus a set of real pictures with no labels may provide enough information on both targets and backgrounds to create a competitive object detection model. Further benchmarking of models using different configurations of synthetic data would be of interest towards quickly deploying very specific detection models or producing assistive pseudolabelling tools.

7.2.2 Enhancing 3D reconstruction

While robust, the proposed item localization pipeline has several flaws that can significantly improve its performance. Within this area, the proposed upgrades are deemed of interest:

- Implement an item localization approach that can account for item occlusion.
- Expand the likelihood map approach so it can include FoV obstacles such as walls.
- Study an integrated approach based on deep learning for item localization.

7.2.3 Kalman filter for blind pose estimation

Due to its visual nature, the proposed pipeline can only map items when the position of the camera is known, condition only satisfied when ArUco markers are on sight. An expansion of this model can be made by adding complementary sensory such as an inertial unit to keep track of the evolution of the state variables by dead reckoning.

8 REFERENCES

[1] G. Nikolakopoulos, "Robotics Lp2 H20," pp. R7010E, Lulea University of Technology, Nov. 2020.

[2] X. Zhou, Y. Peng, C. Long, F. Ren, and C. Shi, "MoNet3D: Towards Accurate Monocular 3D Object Localization in Real Time," *arXiv*, 2020.

[3] "(No Title)." https://www.subtchallenge.com/resources/SubT_Cave_Artifacts_Specification.pdf (accessed Apr. 14, 2021).

[4] F. Chollet, *Deep Learning with Python, Manning*. 2018.

[5] C. Borngrund, U. Bodin, and F. Sandin, "Machine vision for automation of earth-moving machines: Transfer learning experiments with YOLOv3," Luleå University of Technology, 2019.

[6] B. Widrow and M. A. Lehr, "30 Years of Adaptive Neural Networks: Perceptron, Madaline, and Backpropagation," *Proc. IEEE*, vol. 78, no. 9, pp. 1415–1442, 1990, doi: 10.1109/5.58323.

[7] C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall, "Activation Functions: Comparison of trends in Practice and Research for Deep Learning," pp. 1–20, 2018, [Online]. Available: http://arxiv.org/abs/1811.03378.

[8] M. A. Nielsen, "Neural Networks and Deep Learning." Determination Press, 2015, Accessed: May 26, 2021. [Online]. Available: http://neuralnetworksanddeeplearning.com.

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "[AlexNet] ImageNet Classification with Deep Convolutional Neural Networks," 2012. doi: 10.1201/9781420010749.

[10] "Transfer learning and fine-tuning | TensorFlow Core." https://www.tensorflow.org/tutorials/images/transfer_learning (accessed May 26, 2021).

[11] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "[R-CNN I] Rich feature hierarchies for accurate object detection and semantic segmentation," *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, pp. 580–587, 2014, doi: 10.1109/CVPR.2014.81.

[12] K. Nguyen, N. T. Huynh, P. C. Nguyen, K. D. Nguyen, N. D. Vo, and T. V. Nguyen, "[State of the Art] Detecting objects from space: an evaluation of deep-learning modern approaches," *Electron.*, vol. 9, no. 4, pp. 1–18, 2020, doi: 10.3390/electronics9040583.

[13] H. Zhu, H. Wei, B. Li, X. Yuan, and N. Kehtarnavaz, "A review of video object detection: Datasets, metrics and methods," *Appl. Sci.*, vol. 10, no. 21, pp. 1–24, 2020, doi: 10.3390/app10217834.

[14] T. Y. Lin *et al.*, "Microsoft COCO: Common objects in context," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 8693 LNCS, no. PART 5, pp. 740–755, 2014, doi: 10.1007/978-3-319-10602-1_48.

[15] "The PASCAL Visual Object Classes Homepage." http://host.robots.ox.ac.uk/pascal/VOC/ (accessed May 26, 2021).

[16] "COCO - Common Objects in Context - Detection Evaluation." https://cocodataset.org/#detection-eval (accessed May 26, 2021).

[17] "mAP (mean average precision) calculation for different Datasets (MSCOCO, ImageNet, PascalVOC) - darknet." https://www.gitmemory.com/issue/AlexeyAB/darknet/2746/477459452 (accessed May 26, 2021).

[18] "Breaking Down Mean Average Precision (mAP) | by Ren Jie Tan | Towards Data Science." https://towardsdatascience.com/breaking-down-mean-average-precision-map-ae462f623a52 (accessed May 26, 2021).

[19] "cocodataset/cocoapi: COCO API - Dataset @ http://cocodataset.org/." https://github.com/cocodataset/cocoapi (accessed Apr. 14, 2021).

[20] R. Girshick, "[R-CNN II] Fast R-CNN," *Proc. IEEE Int. Conf. Comput. Vis.*, vol. 2015 Inter, pp. 1440–1448, 2015, doi: 10.1109/ICCV.2015.169.

[21] S. Ren, K. He, R. Girshick, and J. Sun, "[R-CNN III] Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 39, no. 6, pp. 1137–1149, 2017, doi: 10.1109/TPAMI.2016.2577031.

[22] Vaibhaw Singh Chandel, "[web] Selective Search for Object Detection (C++ / Python) | Learn OpenCV," pp. 1–14, 2017, [Online]. Available: https://www.learnopencv.com/selective-search-for-object-detection-cpp-python/.

[23] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "[YOLO I] You only look once: Unified, real-time object detection," *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, vol. 2016-Decem, pp. 779–788, 2016, doi: 10.1109/CVPR.2016.91.

[24] J. Redmon and A. Farhadi, "[YOLO II] YOLO9000: Better, faster, stronger," 2017, doi: 10.1109/CVPR.2017.690.

[25] J. Redmon and A. Farhadi, "[YOLO III] YOLO v3," *Tech Rep.*, pp. 1–6, 2018, [Online]. Available: https://pjreddie.com/media/files/papers/YOLOv3.pdf.

[26] "Darknet: Open Source Neural Networks in C." https://pjreddie.com/darknet/ (accessed May 26, 2021).

[27] W. Liu *et al.*, "[SSD] SSD: Single shot multibox detector," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 9905 LNCS, pp. 21–37, 2016, doi: 10.1007/978-3-319-46448-0_2.

[28] M. Tan, R. Pang, and Q. V. Le, "[EfficientDet] EfficientDet: Scalable and efficient object detection," *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, pp. 10778–10787, 2020, doi: 10.1109/CVPR42600.2020.01079.

[29] M. Tan and Q. V. Le, "[EfficientNet] EfficientNet: Rethinking model scaling for convolutional neural networks," *36th Int. Conf. Mach. Learn. ICML 2019*, vol. 2019-June, pp. 10691–10700, 2019.

[30] G. Ghiasi, T. Y. Lin, and Q. V. Le, "[NAS] NAS-FPN: Learning scalable feature pyramid architecture for object detection," *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, vol. 2019-June, pp. 7029–7038, 2019, doi: 10.1109/CVPR.2019.00720.

[31] X. Li, T. Lai, S. Wang, Q. Chen, C. Yang, and R. Chen, "[FPN] Feature Pyramid Networks for Object Detection," *Proc. - 2019 IEEE Intl Conf Parallel Distrib. Process. with Appl. Big Data Cloud Comput. Sustain. Comput. Commun. Soc. Comput. Networking, ISPA/BDCloud/SustainCom/SocialCom 2019*, pp. 1500–1504, 2019, doi: 10.1109/ISPA-BDCloud-SustainCom-SocialCom48970.2019.00217.

[32] S. Liu, L. Qi, H. Qin, J. Shi, and J. Jia, "[PAN] Path Aggregation Network for Instance Segmentation," *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, pp. 8759–8768, 2018, doi: 10.1109/CVPR.2018.00913.

[33] A. Bochkovskiy, C. Y. Wang, and H. Y. M. Liao, "[YOLO IV.I] YOLOv4: Optimal Speed and Accuracy of Object Detection," *arXiv*, 2020.

[34] "AlexeyAB/darknet: YOLOv4 / Scaled-YOLOv4 / YOLO - Neural Networks for Object Detection (Windows and Linux version of Darknet)." https://github.com/AlexeyAB/darknet (accessed Apr. 14, 2021).

[35] W. Luo, Y. Li, R. Urtasun, and R. Zemel, "Understanding the effective receptive field in deep convolutional neural networks," *Adv. Neural Inf. Process. Syst.*, no. Nips, pp. 4905–4913, 2016.

[36] C. Y. Wang, A. Bochkovskiy, and H. Y. M. Liao, "[YOLO IV.II] Scaled-YOLOv4: Scaling cross stage partial network," *arXiv*, 2020.

[37] C. Y. Wang, H. Y. Mark 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," *IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit. Work.*, vol. 2020-June, pp. 1571–1580, 2020, doi: 10.1109/CVPRW50498.2020.00203.

[38] H. Lee, R. Grosse, R. Ranganath, and A. Y. Ng, "Unsupervised learning of hierarchical representations with convolutional deep belief networks," *Commun. ACM*, vol. 54, no. 10, pp. 95–103, 2011, doi: 10.1145/2001269.2001295.

[39] "Specific format of annotation · Issue #60 · AlexeyAB/Yolo_mark." https://github.com/AlexeyAB/Yolo_mark/issues/60 (accessed May 27, 2021).

[40] "COCO - Common Objects in Context - Format data." https://cocodataset.org/#format-data (accessed May 27, 2021).

[41] D. Dwibedi, I. Misra, and M. Hebert, "[CutPaste&Learn] Cut, Paste and Learn: Surprisingly Easy Synthesis for Instance Detection," *Proc. IEEE Int. Conf. Comput. Vis.*, vol. 2017-Octob, pp. 1310–1319, 2017, doi: 10.1109/ICCV.2017.146.

[42] "Ignition Robotics - Rescue Randy Sitting." https://app.ignitionrobotics.org/OpenRobotics/fuel/models/Rescue Randy Sitting (accessed Apr. 14, 2021).

[43] "Unity-Technologies/SynthDet: SynthDet - An end-to-end object detection pipeline using synthetic data." https://github.com/Unity-Technologies/SynthDet (accessed Apr. 14, 2021).

[44] 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," *Proc. - 2019 Int. Conf. Comput. Vis. Work. ICCVW 2019*, pp. 2787–2796, 2019, doi: 10.1109/ICCVW.2019.00340.

[45] "com.unity.perception/TUTORIAL.md at master · Unity-Technologies/com.unity.perception." https://github.com/Unity-Technologies/com.unity.perception/blob/master/com.unity.perception/Documentation~/Tutorial/TUTORIAL.md (accessed May 27, 2021).

[46] "pabsan-0/sub-t: Object detection in Sub-T environments." https://github.com/pabsan-0/sub-t (accessed Apr. 14, 2021).

[47] S. S. Shivakumar, N. Rodrigues, A. Zhou, I. D. Miller, V. Kumar, and C. J. Taylor, "PST900: RGB-Thermal Calibration, Dataset and Segmentation Network," *Proc. - IEEE Int. Conf. Robot. Autom.*, pp. 9441–9447, 2020, doi: 10.1109/ICRA40945.2020.9196831.

[48] "ShreyasSkandanS/pst900_thermal_rgb: ICRA 2020 | Repository for 'PST900 RGB-Thermal Calibration, Dataset and Segmentation Network' | C++, Python, PyTorch." https://github.com/ShreyasSkandanS/pst900_thermal_rgb (accessed Apr. 14, 2021).

[49] "AlexeyAB/Yolo_mark: GUI for marking bounded boxes of objects in images for training neural network Yolo v3 and v2." https://github.com/AlexeyAB/Yolo_mark (accessed May 27, 2021).

[50] "YOLOv4_Tutorial.ipynb - Colaboratory." https://colab.research.google.com/drive/12QusaaRj_lUwCGDvQNfICpa7kA7_a2dE (accessed May 27, 2021).

[51] "google/automl: Google Brain AutoML." https://github.com/google/automl (accessed Apr. 14, 2021).

[52] "EfficientDet - tutorial.ipynb - Colaboratory." https://colab.research.google.com/github/google/automl/blob/master/efficientdet/tutorial.ipynb (accessed May 27, 2021).

[53] "Welcome To Colaboratory - Colaboratory." https://colab.research.google.com/notebooks/intro.ipynb (accessed Apr. 14, 2021).

[54] "tf-objdetector/yolo_tf_converter.py at master · AlessioTonioni/tf-objdetector."

https://github.com/AlessioTonioni/tf-objdetector/blob/master/yolo_tf_converter.py (accessed May 27, 2021).

[55] "While resizing the images during training(random=1 in .cfg file), does yolo internally changes the grid size? or it uses the initial grid size what we see at the starting of the training. · Issue #728 · pjreddie/darknet." https://github.com/pjreddie/darknet/issues/728#issuecomment-383539370 (accessed May 27, 2021).

[56] "IP Webcam - Apps on Google Play." https://play.google.com/store/apps/details?id=com.pas.webcam&hl=en&gl=US (accessed May 27, 2021).

[57] P. Corke, *Robotics, Vision and Control - Fundamental Algorithms In MATLAB® Second, Completely Revised, Extended And Updated Edition*, vol. 75, no. 1–2. 2017.

[58] S. Garrido-Jurado, R. Muñoz-Salinas, F. J. Madrid-Cuevas, and M. J. Marín-Jiménez, "Automatic generation and detection of highly reliable fiducial markers under occlusion," *Pattern Recognit.*, vol. 47, no. 6, pp. 2280–2292, 2014, doi: 10.1016/j.patcog.2014.01.005.

[59] "Online ArUco markers generator." https://chev.me/arucogen/ (accessed May 27, 2021).

[60] K. Ogata, *Modern Control Engineering Fifth Edition*, vol. 17, no. 3. 2009.

[61] G. Welch and G. Bishop, "An Introduction to the Kalman Filter," *In Pract.*, vol. 7, no. 1, 2006, doi: 10.1.1.117.6808.

[62] A. M. Kettner and M. Paolone, "Sequential Discrete Kalman Filter for Real-Time State Estimation in Power Distribution Systems: Theory and Implementation," *IEEE Trans. Instrum. Meas.*, vol. 66, no. 9, pp. 2358–2370, 2017, doi: 10.1109/TIM.2017.2708278.

[63] "OpenCV: Camera Calibration and 3D Reconstruction." https://docs.opencv.org/3.4/d9/d0c/group__calib3d.html (accessed May 27, 2021).

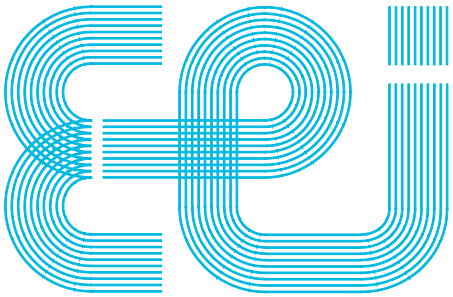
[64] "OpenCV: Calibration with ArUco and ChArUco." https://docs.opencv.org/master/da/d13/tutorial_aruco_calibration.html (accessed May 27, 2021).

[65] "OpenCV: Camera Calibration." https://docs.opencv.org/master/dc/dbb/tutorial_py_calibration.html (accessed May 27, 2021).

[66] T. Ke and S. I. Roumeliotis, "An efficient algebraic solution to the perspective-three-point problem," *Proc. - 30th IEEE Conf. Comput. Vis. Pattern Recognition, CVPR 2017*, vol. 2017-Janua, pp. 4618–4626, 2017, doi: 10.1109/CVPR.2017.491.

[67] "Speed Up Your Python Program With Concurrency – Real Python." https://realpython.com/python-concurrency/ (accessed Jun. 13, 2021).

Vigo, 5th of July of 2021



Escola de Enxeñería Industrial

TRABALLO FIN DE MESTRADO

Convolutional neural networks for object detection in subterranean environments

Master's Degree in Industrial Engineering

Document

BUDGET

Universidade Vigo

CONTENTS

1 SUPPLIES

Item	Unit cost	Cost
Laptop supply High-end gaming laptop with specs: 1000GB HDD, 256GB SSD, 16GB RAM, NVIDIA GTX1060 6GB VRAM, Intel Core i7 8750, inc. basic peripherals such as mouse, mousepad, webcam and shipping.	1,499.95 €	1,499.95 €
DARPA item: helmet supply White medium/large caving helmet Petzl BODEO geared with a Princeton TEC Apex (APX550-BK) headlamp, inc. shipping.	206.29 €	206.29 €
DARPA item: rope supply Coiled Black Diamond 9.9 mm Climbing Rope, held together with a 25.4 mm (1 in) wide black strap, inc. shipping.	135.99 €	135.99 €
Smartphone supply Android 10 smartphone with 5G WiFi hotspot support, screen resolution 1080x2400, size 6.67in, USB C. Inc. high speed charger with 5m long USB wire plus shipping.	290.30 €	290.30 €
Smartphone tripod supply Smartphone-compatible tripod for static holding and 3 degrees of freedom with a standing height of 1 meter, inc. shipping.	22.19 €	22.19 €
Office supplies Various office supplies for both technical work and experimental layout, including pens, paper, printing costs, scissors, adhesive tape and tape measure.	15.00 €	15.00 €
	TOTAL	**2,169.72 €**

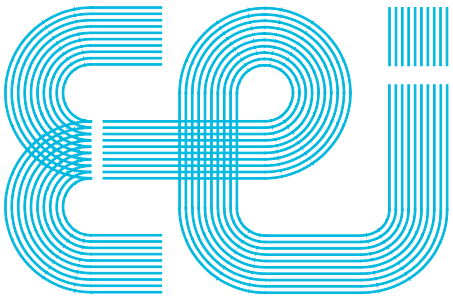
2 ACTIVITY COSTS

Item	Unit cost	Cost
Engineering hours x600 Research engineering hours used to develop different activities towards the present work. Includes deep learning and robotics research, object detector training and evaluation, robot perception layer design, implementation and validation.	25.00 €	15,000.00 €
	Total	**15,000.00 €**

3 SUMMARY

Item	Cost
SUPPLIES	**2,169.72 €**
ACTIVITY COSTS	**15,000.00 €**
TOTAL	**17,169.72 €**

This project's current execution budget reaches the amount of **SEVENTEEN THOUSAND ONE HUNDRED SIXTY-NINE EUROS AND SEVENTY-TWO CENTS (17,169.72 €)**, according to the present documentation.



Escola de Enxeñería Industrial

TRABALLO FIN DE MESTRADO

Convolutional neural networks for object detection in subterranean environments

Master's Degree in Industrial Engineering

Document

APPENDIX I: DETECTOR BENCHMARKING CODE

Universidade Vigo

CONTENTS

```
#!/bin/bash

# This bash script will setup a folder structure to work with YOLOv4 networks
# plus download a current version of AlexeyAB darknet.
# You will NEED TO HARDCODE YOUR HOME USER, since DARKNET WONT EAT ~/* dirs
# Run this script through dos2unix or copy raw text for compatibility
# Find commands for training at the bottom of this script
#
#  ~/
#  └── YOLOv4/
#      ├── darknet/
#      │   └── ...
#      ├── yolo_backup/
#      ├── weights/
#      │   ├── yolov4-tiny.conv.29
#      │   ├── yolov4.conv.137
#      │   ├── yolov4-csp.conv.142
#      │   └── yolov4x-mish.conv.166
#      ├── cfg/
#      │   ├── yolov4-tiny-416-6.cfg
#      │   ├── yolov4-tiny-416-6-test.cfg *
#      │   ├── yolov4-416-6.cfg
#      │   ├── yolov4-416-6-test.cfg *
#      │   ├── yolov4-csp-512-6.cfg
#      │   ├── yolov4-csp-512-6-test.cfg *
#      │   ├── yolov4x-mish-640-6.cfg
#      │   └── yolov4x-mish-640-6-test.cfg *
#      └── PPU-6/
#          ├── train/
#          │   ├── picture001.jpg
#          │   ├── picture001.txt
#          │   └── ...
#          ├── valid/
#          │   ├── picture101.jpg
#          │   ├── picture101.txt
#          │   └── ...
#          ├── test/
#          │   ├── picture201.jpg
#          │   ├── picture201.txt
#          │   └── ...
#          ├── train.txt
#          ├── valid.txt
#          ├── obj.data
#          └── obj.names
#
#  * not automatically downloaded, read script for more info


# Setup folder structure
mkdir ~/YOLOv4/
```

```
mkdir ~/YOLOv4/PPU-6
mkdir ~/YOLOv4/cfg
mkdir ~/YOLOv4/weights
mkdir ~/YOLOv4/yolo_backup

### Cloning Darknet + YOLO weights ### ---------------------------------------
# clone darknet repo
cd ~/YOLOv4/
git clone https://github.com/AlexeyAB/darknet

# change makefile to have GPU and OPENCV enabled
cd ~/YOLOv4/darknet
sed -i 's/OPENCV=0/OPENCV=1/' Makefile
sed -i 's/GPU=0/GPU=1/' Makefile
sed -i 's/CUDNN=0/CUDNN=1/' Makefile
sed -i 's/CUDNN_HALF=0/CUDNN_HALF=1/' Makefile

# verify CUDA
# /usr/local/cuda/bin/nvcc --version

# make darknet
make


# download pretrained weights, all from AlexeyAB's repo
cd ~/YOLOv4/weights
wget https://github.com/AlexeyAB/darknet/releases/download/darknet_yolo_v4_pre/yolov4-tiny.conv.29
wget https://github.com/AlexeyAB/darknet/releases/download/darknet_yolo_v3_optimal/yolov4.conv.137
wget https://github.com/AlexeyAB/darknet/releases/download/darknet_yolo_v4_pre/yolov4-csp.conv.142
wget https://github.com/AlexeyAB/darknet/releases/download/darknet_yolo_v4_pre/yolov4x-mish.conv.166


### Download the modified cfg files ### ----------------------------------------
cd ~/YOLOv4/cfg

# Google drive IDs for the files...
# yolo4csp    1Ichrr6Uu3TRvKuGW2ZqYumGzCu21XhtG
# yolo4tiny   1ZZyaNh_bfiEXVknNqtJ9UHlvEReeQUKb
# yolo4mish   1m4xxAacLuQ7NR7qStu0pfGONBHgzwLcx
# yolo4       1wUWi3q1DlpfmLQpxuBdHx1TveWUYPihA
# Manually generate test cfg by setting batch size and subdivisions to 1

wget --no-check-certificate 'https://docs.google.com/uc?export=download&id=1Ichrr6Uu3TRvKuGW2ZqYumGzCu21XhtG' -O yolov4-csp-512-6.c
wget --no-check-certificate 'https://docs.google.com/uc?export=download&id=1ZZyaNh_bfiEXVknNqtJ9UHlvEReeQUKb' -O yolov4-tiny-416-6.
wget --no-check-certificate 'https://docs.google.com/uc?export=download&id=1m4xxAacLuQ7NR7qStu0pfGONBHgzwLcx' -O yolov4x-mish-640-6
wget --no-check-certificate 'https://docs.google.com/uc?export=download&id=1wUWi3q1DlpfmLQpxuBdHx1TveWUYPihA' -O yolov4-416-6.cfg


### Obtain data assets, model-agnostic within YOLO ### -----------------------------------------
cd ~/YOLOv4/PPU-6

# download data, uncompress & clear the compressed file from disk
wget --load-cookies /tmp/cookies.txt "https://docs.google.com/uc?export=download&confirm=$(wget --quiet --save-cookies /tmp/cookies
unrar x -y PPU-6.rar
rm PPU-6.rar

# Generate obj.data
cat > ~/YOLOv4/PPU-6/obj.data << ENDOFFILE
classes = 6
train = /home/pablo/YOLOv4/PPU-6/train.txt
valid = /home/pablo/YOLOv4/PPU-6/valid.txt
names = /home/pablo/YOLOv4/PPU-6/obj.names
backup = /home/pablo/YOLOv4/yolo_backup
ENDOFFILE

# Generate obj.names
cat > ~/YOLOv4/PPU-6/obj.names << ENDOFFILE

backpack
helmet
```

```
drill
extinguisher
survivor
rope
ENDOFFILE

# Generate & run a python script that will generate the txts
cat > ~/YOLOv4/PPU-6/generate-txts.py << ENDOFFILE
import os
trainpath = '/home/pablo/YOLOv4/PPU-6/train/'
testpath  = '/home/pablo/YOLOv4/PPU-6/test/'
validpath = '/home/pablo/YOLOv4/PPU-6/valid/'
with open('./train.txt', 'w') as file:
    a = os.listdir('./train')
    a = [i for i in a if i[-1]!='t']
    for i in a:
        file.write(trainpath + i +'\n')
with open('./test.txt', 'w') as file:
    a = os.listdir('./test')
    a = [i for i in a if i[-1]!='t']
    for i in a:
        file.write(testpath + i +'\n')
with open('./valid.txt', 'w') as file:
    a = os.listdir('./valid')
    a = [i for i in a if i[-1]!='t']
    for i in a:
        file.write(validpath + i +'\n')
ENDOFFILE

# generate the txt targets and remove the python script
touch ~/YOLOv4/PPU-6/train.txt
touch ~/YOLOv4/PPU-6/test.txt
touch ~/YOLOv4/PPU-6/valid.txt
python3 ~/YOLOv4/PPU-6/generate-txts.py
rm ~/YOLOv4/PPU-6/generate-txts.py



### Lines for actually training each model ### ----------------------------------------
### Retrieve network output from yolo_backup/
### Delete the whole dir to clear space once finished, nothing is written outside of it
#
# LD_LIBRARY_PATH=/usr/local/lib:/usr/lib/cuda-10.2/targets/x86_64-linux/lib/ ./darknet detector train ~/YOLOv4/PPU-6/obj.data ~/YO
#
# Start training - change dir before!
# cd ~/YOLOv4/darknet/
#
# Train yolov4-tiny
# ./darknet detector train ~/YOLOv4/PPU-6/obj.data ~/YOLOv4/yolov4-tiny-416-6.cfg ~/YOLOv4/yolov4-tiny.conv.29 -dont_show -map
#
# Train yolov4 vanilla
# ./darknet detector train ~/YOLOv4/PPU-6/obj.data ~/YOLOv4/yolov4-416-6.cfg ~/YOLOv4/yolov4.conv.137 -dont_show -map
#
# Train yolov4-csp
# ./darknet detector train ~/YOLOv4/PPU-6/obj.data ~/YOLOv4/yolov4-csp-512-6.cfg ~/YOLOv4/yolov4-csp.conv.142 -dont_show -map
#
# Train yolov4x-mish
# ./darknet detector train ~/YOLOv4/PPU-6/obj.data ~/YOLOv4/yolov4x-mish-640-6.cfg ~/YOLOv4/yolov4x-mish.conv.166 -dont_show -map



# THE FOLLOWING ARE FOR MY LOCAL MACHINE, WHICH HAS ISSUES WITH CUDNN VERSIONS
# AND NEEDS TO DEFINE THE PATH VARIABLE BEFORE RUNNING THE COMMAND
#
# Train
# LD_LIBRARY_PATH=/usr/local/lib:/usr/lib/cuda-10.2/targets/x86_64-linux/lib/ ./darknet detector train ~/YOLOv4/PPU-6/obj.data ~/YO
#
# Test multiple files

# LD_LIBRARY_PATH=/usr/local/lib:/usr/lib/cuda-10.2/targets/x86_64-linux/lib/ ./darknet detector test cfg/obj.data cfg/yolov3.cfg y
#
```

```
188 # Infer multilpe files with positional output to text file - USE THIS FOR MAP TOO
189 # LD_LIBRARY_PATH=/usr/local/lib:/usr/lib/cuda-10.2/targets/x86_64-linux/lib/ ./darknet detector test ~/YOLOv4/PPU-6/obj.data ~/YOL
```

```
# Full benchmarking of MAP and FPS for various darknet models

# Generate a resized copy of the test set for benchmarking the nets on their baseline resolution
test-set-resizer.py ~/YOLOv4/PPU-6/test/ 416 512 640

# Temporary cd to darknet dir
pushd ~/YOLOv4/darknet


# Darknet inference (results to text file) on TINY
LD_LIBRARY_PATH=/usr/local/lib:/usr/lib/cuda-10.2/targets/x86_64-linux/lib/ \
    ./darknet detector test \
        ~/YOLOv4/PPU-6/obj.data \
        ~/YOLOv4/cfg/yolov4-tiny-416-6-test.cfg \
        ~/YOLOv4/weights-trained/yolov4-tiny-416-6_best.weights \
        -dont_show -ext_output < $OLDPWD/test-416.txt \
        > $OLDPWD/results-yolov4-tiny-416.txt

# Darknet inference (results to text file) on TINY 3L
LD_LIBRARY_PATH=/usr/local/lib:/usr/lib/cuda-10.2/targets/x86_64-linux/lib/ \
    ./darknet detector test \
        ~/YOLOv4/PPU-6/obj.data \
        ~/YOLOv4/cfg/yolov4-tiny-3l-416-6-test.cfg \
        ~/YOLOv4/weights-trained/yolov4-tiny-3l-416-6_best.weights \
        -dont_show -ext_output < $OLDPWD/test-416.txt \
        > $OLDPWD/results-yolov4-tiny-3l-416.txt

# Darknet inference (results to text file) on YOLOv4
LD_LIBRARY_PATH=/usr/local/lib:/usr/lib/cuda-10.2/targets/x86_64-linux/lib/ \
    ./darknet detector test \
        ~/YOLOv4/PPU-6/obj.data \
        ~/YOLOv4/cfg/yolov4-vanilla-416-6-test.cfg \
        ~/YOLOv4/weights-trained/yolov4-416-6_best.weights \
        -dont_show -ext_output < $OLDPWD/test-416.txt \
        > $OLDPWD/results-yolov4-416.txt

# Darknet inference (results to text file) on YOLOv4 CSP
LD_LIBRARY_PATH=/usr/local/lib:/usr/lib/cuda-10.2/targets/x86_64-linux/lib/ \
    ./darknet detector test \
        ~/YOLOv4/PPU-6/obj.data \
        ~/YOLOv4/cfg/yolov4-csp-512-6-test.cfg \
        ~/YOLOv4/weights-trained/yolov4-csp-512-6_best.weights \
        -dont_show -ext_output < $OLDPWD/test-512.txt \
        > $OLDPWD/results-yolov4-csp-512.txt

# Darknet inference (results to text file) on YOLOv4 MISH
LD_LIBRARY_PATH=/usr/local/lib:/usr/lib/cuda-10.2/targets/x86_64-linux/lib/ \
    ./darknet detector test \
        ~/YOLOv4/PPU-6/obj.data \
```

```
        ~/YOLOv4/cfg/yolov4x-mish-640-6-test.cfg \
        ~/YOLOv4/weights-trained/yolov4x-mish-640-6_best.weights \
        -dont_show -ext_output < $OLDPWD/test-640.txt \
        > $OLDPWD/results-yolov4x-mish-640.txt


# Pop back to previous dir (the one holding this script)
popd


# Convert inference results from darknet to MS COCO
# PPU-6-size.json is loaded only to provide the paths to the images
python3 darknet_2_coco_detections.py \
    ./assets/PPU-6-416.json \
    results-yolov4-tiny-416.txt \
    results-yolov4-tiny-416.json

python3 darknet_2_coco_detections.py \
    ./assets/PPU-6-416.json \
    results-yolov4-tiny-3l-416.txt \
    results-yolov4-tiny-3l-416.json

python3 darknet_2_coco_detections.py \
    ./assets/PPU-6-416.json \
    results-yolov4-416.txt \
    results-yolov4-416.json

python3 darknet_2_coco_detections.py \
    ./assets/PPU-6-512.json \
    results-yolov4-csp-512.txt \
    results-yolov4-csp-512.json

python3 darknet_2_coco_detections.py \
    ./assets/PPU-6-640.json \
    results-yolov4x-mish-640.txt \
    results-yolov4x-mish-640.json


# Convert inference results from darknet to MS COCO
python3 batch-annotate.py \
    results-yolov4-tiny-416.txt \
    results-yolov4-tiny-3l-416.txt \
    results-yolov4-416.txt \
    results-yolov4-csp-512.txt \
    results-yolov4x-mish-640.txt


# Dump new files into dump dir
rm -r ./generated-files-dump
mkdir ./generated-files-dump
mv *.json ./generated-files-dump
mv *.txt ./generated-files-dump
mv resized-* ./generated-files-dump
mkdir ./infer_results
mv results* ./infer_results


# Remove MAP-table if exists so it is overwritten & generate MAP table text file
rm MAP-table.txt

echo MAP TABLE FOR YOLOv4-TINY >> MAP-table.txt
python3 valcoco.py \
    ./assets/PPU-6-416.json \
    ./generated-files-dump/results-yolov4-tiny-416.json \
    >> MAP-table.txt

echo MAP TABLE FOR YOLOv4-TINY-3L >> MAP-table.txt
python3 valcoco.py \
    ./assets/PPU-6-416.json \
```

```
    ./generated-files-dump/results-yolov4-tiny-3l-416.json \
    >> MAP-table.txt

echo MAP TABLE FOR YOLOv4 >> MAP-table.txt
python3 valcoco.py \
    ./assets/PPU-6-416.json \
    ./generated-files-dump/results-yolov4-416.json \
    >> MAP-table.txt

echo MAP TABLE FOR YOLOv4-CSP >> MAP-table.txt
python3 valcoco.py \
    ./assets/PPU-6-512.json \
    ./generated-files-dump/results-yolov4-csp-512.json \
    >> MAP-table.txt

echo MAP TABLE FOR YOLOv4x-MISH >> MAP-table.txt
python3 valcoco.py \
    ./assets/PPU-6-640.json \
    ./generated-files-dump/results-yolov4x-mish-640.json \
    >> MAP-table.txt


# Get the average FPS from the inference logs
python3 fps-retriever.py \
    FPS.txt \
    ./generated-files-dump/results-yolov4-tiny-416.txt \
    ./generated-files-dump/results-yolov4-tiny-3l-416.txt \
    ./generated-files-dump/results-yolov4-416.txt \
    ./generated-files-dump/results-yolov4-csp-512.txt \
    ./generated-files-dump/results-yolov4x-mish-640.txt


# Finish message alert
echo Finished benchmarking successfully!
```

```python
from sys import argv
import os
import shutil
import cv2

'''Takes a YOLO annotated picture folder and makes a copy of it in the current path,
resizing the pictures to a specified square resolution.

args:
    A source directory for extracting image originals
    Resolution 1
    Resolution 2... etc (optional)
'''

# import source test path from arg
test_path = argv[1]
if test_path[-1] != '/':
    test_path = test_path + '/'

# import target_resolutions from arg
target_resolutions = argv[2:]

# create dump folders for each resolution
for resolution in target_resolutions:
    try:
        os.mkdir('./resized-' + resolution)
    except FileExistsError:
        print('Catched FileExistsError! There is already a test dir with that resolution.')

# fetch yolo directory and copy items to the new ones, resizing only images
for file in os.listdir(test_path):
    for resolution in target_resolutions:
        if file[-1] == 't':
            shutil.copy(test_path + file, './resized-' + resolution + '/' + file)
        else:
            pic = cv2.imread(test_path + file)
            pic = cv2.resize(pic, (int(resolution), int(resolution)))
            cv2.imwrite('./resized-' + resolution + '/' + file[:-4] + '.jpg', pic)

for resolution in target_resolutions:
    with open(f'test-{resolution}.txt', 'w') as file:
        for picname in [picname for picname in os.listdir('./resized-' + resolution) if picname[-1]!='t']:
            file.write(os.getcwd() + '/resized-' + resolution + '/' + picname + '\n')
```

```python
from sys import argv
import os
import json

''' Creates a json file in COCO format for the predictions of darknet.

ARGS:
    ground-truth file in COCO json
    detection file in darknet default (see preview below)
    name of the output detection file in COCO json (will overwrite)

Expected predictions on input file are of this shape...:

/home/pablo/YOLOv4/PPU-6/test/25_exyn_bag15_rgb_frame1500305.png: Predicted in 134.026000 milli-seconds.
drill: 99%      (left_x:  406   top_y:  344   width:   42   height:   53)
survivor: 100%  (left_x:  559   top_y:  303   width:  119   height:  268)
'''


def get_path2id_dict(ground_truth_json):
    ''' Reads ground truth json and links image paths to image ids '''
    path2id = {}
    with open(ground_truth_json) as gt:
        for image in json.load(gt)['images']:
            name = image['file_name']
            id = image['id']
            path2id[name] = id
    return path2id


if __name__ == '__main__':
    # get ground truth file and detection file from input arguments
    gt_file       = argv[1]
    dets_file_in  = argv[2]
    dets_file_out = argv[3]

    # build a dictionary that links filenames to image ids defined in the GT json
    path2id = get_path2id_dict(gt_file)

    # build a dictionary to link each item to its coco ID
    names2ids = {
        'backpack': 1,
        'helmet': 2,
        'drill': 3,
        'extinguisher': 4,
        'survivor': 5,
        'rope': 6,
        }

```

```
    # define container for holding the output
    massive_output_list = []

    # read the file holding the darknet detections and scan it
    with open(dets_file_in, 'r') as file:
        for line in file.readlines():

            if '/home/' in line:
                # fetch the image id from the image path that is read
                current_image_path = line.split(':')[0].split('/')[-1]
                current_image_id = path2id[current_image_path]

            if '%' in line:
                # extract category and score by using % as separator
                line_modified = line.replace(':', '%')
                category_id = names2ids[line_modified.split('%')[0]]
                score    = int(line_modified.split('%')[1])/100

                # extract bounding box numbers by doing some complex juggling
                x1, y1, w, h = [int(i) for i in line[:-2].replace('-','').split() if i.isdigit()]

                massive_output_list.append(
                    {
                    'image_id':     current_image_id,
                    'category_id':  category_id,
                    'bbox':         [x1, y1, w, h],
                    'score':        score,
                    }
                )

    # write to output json file AS STRING else it breaks later!
    with open(dets_file_out, 'w') as outfile:
        string_content = json.dumps(massive_output_list)
        outfile.write(string_content)
```

```
import cv2
import colorsys
import os
from sys import argv

'''
This script is meant to read a yolo log file with batch inferred results
and draw predicted bounding boxes on the source pictures.

ARGS:
    Text file 1 *
    Text file 2 (optional)
    ...
    Text file N (optional)

The file needed as input can be obtained by running this command:
./darknet detector test PATH/obj.data PATH/yolov4.cfg PATH/yolov4.weights -dont_show -ext_output < PATH/test.txt > results_yo.txt

Where test.txt is a textfile with the absolute path of each of the images
to be batch-inferred and results_yo.txt is the output that will be generated
by darknet. Inside this file, each predicted image should look something
like this, adjust file scraping to if neccesary for further releases:

"""
Enter Image Path:  Detection layer: 30 - type = 28
 Detection layer: 37 - type = 28
/home/pablo/YOLOv4/PPU-6/test/25_exyn_bag15_rgb_frame1500305.png: Predicted in 104.754000 milli-seconds.
drill: 66%      (left_x:  409   top_y:  342   width:   41   height:   58)
survivor: 99%   (left_x:  562   top_y:  298   width:  120   height:  269)
backpack: 30%   (left_x:  650   top_y:  490   width:   33   height:   61)
Enter Image Path:  Detection layer: 30 - type = 28
 Detection layer: 37 - type = 28
"""

'''

# get text file paths from argv
text_files = argv[1:]


for text_file_path in text_files:

    # define ouput path + init counter 'idx'
    out_dir = text_file_path.split('/')[-1][:-4] + '-infer-results'
    idx = 0

    # Attempt to create the target dir. Hold in a try in case the dir already exists
    try:
        os.mkdir(out_dir)
```

```
    except:
        pass

    # required format params for drawing annotations
    thickness = 1
    font = cv2.FONT_HERSHEY_SIMPLEX
    fontScale = 0.5

    # read the output of the YOLO log to load predictions
    with open(text_file_path, 'r') as file:
        for line in file.readlines():

            # when a new picture is introduced, save previous and load+resize new
            if '/home/pablo/' in line:
                # save current picture to file unless first iteration
                if idx != 0:
                    cv2.imwrite(f'{out_dir + "/" + str(idx)}.jpg', pic)
                    print(f'Saved at {out_dir + "/" + str(idx)}.jpg')

                # load file as cv2 image from text file
                picpath = line.split(':')[0]
                print(picpath)
                pic = cv2.imread(picpath)
                idx += 1

                # resize to small size & store original size to work labels later
                picsize_original = pic.shape[0]
                pic = cv2.resize(pic, (680,680))


            # when a prediction is recorded, add the bbox to the current image
            if '%' in line:
                # read the class + confidence score as title
                title = line.split('%')[0] + '%'

                # use the class initial to define color
                hue = ord(title[0]) - (100.5 - ord(title[0])) * 15
                color = tuple(255 * i  for i in colorsys.hsv_to_rgb(hue/360.0, 1, 1))

                # work out where to draw the bounding box depending on output img size
                scalefactor = 680 / picsize_original
                x1, y1, w, h = [int(abs(int(i))*scalefactor) for i in line[:-2].replace('-','').split() if i.isdigit()]

                # actually overwrite the picture with box + title
                pic = cv2.rectangle(pic, (x1, y1), (x1 + w, y1 + h), color, thickness)
                pic = cv2.putText(pic, title, (x1, y1), font, fontScale, color, thickness, cv2.LINE_AA)
```

pabsan-0 modified folder structure and renamed folders History

1 contributor

16 lines (12 sloc) | 341 Bytes

```python
import sys
from pycocotools.coco import COCO
from pycocotools.cocoeval import COCOeval


if __name__ == '__main__':
    gt = sys.argv[1]
    pred = sys.argv[2]

    cocoGt = COCO(gt)
    cocoDt = cocoGt.loadRes(pred)

    cocoEval = COCOeval(cocoGt, cocoDt, 'bbox')
    cocoEval.evaluate()
    cocoEval.accumulate()
    cocoEval.summarize()
```

```python
from sys import argv
import numpy as np

'''
From a darknet results file, compute the average FPS of the predictions.

args:
    Output file
    Darknet results file 1
    Darknet results file 1 (optional)
    (...)
'''

output_file = argv[1]
with open(output_file, 'w'):
    pass

for darknet_results in argv[2:]:
    milliseconds = []
    with open(darknet_results, 'r') as file:
        for line in file.readlines():
            if 'Predicted in' in line:
                milliseconds.append(float(line.split('in ')[-1].split('milli')[0]))

    with open(output_file, 'a') as file:
        avg_ms = np.mean(milliseconds)
        file.write(f'{darknet_results}: {avg_ms} milli-seconds per image, or {1000/avg_ms} FPS\n')
```

Training on PPU-6 Dataset

```
import os
import sys
import tensorflow.compat.v1 as tf
import cv2
import os
import sys

imsize = 512


# Clone efficientdet repo - DO THIS AT THE BEGGINING AND RESTART RUNTIME WITH THE PROVIDED BUTTON
%cd /content/drive/MyDrive/training-effdet/

if "efficientdet" not in os.getcwd():
    !git clone --depth 1 https://github.com/google/automl
    os.chdir('automl/efficientdet')
    sys.path.append('.')
    !pip install -r requirements.txt
    !pip install -U 'git+https://github.com/cocodataset/cocoapi.git#subdirectory=PythonAPI'
else:
    !git pull
```

Prepare data

```
!mkdir /content/raw-data/
%cd /content/raw-data/

# download data, uncompress & clear the compressed file from disk
!wget --load-cookies /tmp/cookies.txt "https://docs.google.com/uc?export=download&confirm=$(wget --quiet --save-cookies
!unrar x -y PPU-6.rar

# Create one text file per split that point to each of the images in it
for split in ['train', 'valid', 'test']:
    with open(f'/content/raw-data/{split}.txt', 'w') as file:
        dir = os.listdir(f'/content/raw-data/{split}')
        image_list = [image for image in dir if image[-1]!='t']
        for image in image_list:
            file.write(f'/content/raw-data/{split}/' + image +'\n')

```

```
%cd /content/raw-data/
# Resizing images to network size
for split in ['train', 'valid', 'test']:
    with open(f'/content/raw-data/{split}.txt') as file:
        all_pics = file.read().splitlines()
        total = len(all_pics)
        for idx, name in enumerate(all_pics):
            pic = cv2.imread(name)
            pic = cv2.resize(pic, (imsize,imsize))
            cv2.imwrite(name, pic)
            print(f'\rConverting {split} split... {idx+1} / {total} converted. Current size {pic.shape}', end='')
```

```
/content/raw-data
Converting test split... 520 / 520 converted. Current size (512, 512, 3)
```

```
# create a obj.names file to hold the names of the classes

names = '''\
backpack
helmet
drill
extinguisher
survivor
rope'''

```

```
%cd /content/raw-data/
with open('obj.names','w') as file:
    file.write(names)
```

```
/content/raw-data
```

```
# Download yolo2tfrecord. This is a modified file to match some tf sublibraries
# Find source file at https://github.com/AlessioTonioni/tf-objdetector
# CHANGE LINE 115/116 depending on the image extension lenght!! (.jpg vs .jpeg)

%cd /content/raw-data/
!wget --load-cookies /tmp/cookies.txt "https://docs.google.com/uc?export=download&confirm=$(wget --quiet --save-cookies
```

```
/content/raw-data
--2021-03-24 18:07:41--  https://docs.google.com/uc?export=download&confirm=&id=1CE3-DxC_BsAxRdjO4v5-87ury17FhA_K
Resolving docs.google.com (docs.google.com)... 64.233.167.102, 64.233.167.113, 64.233.167.100, ...
Connecting to docs.google.com (docs.google.com)|64.233.167.102|:443... connected.
HTTP request sent, awaiting response... 302 Moved Temporarily
Location: https://doc-0k-58-docs.googleusercontent.com/docs/securesc/eg3dk9k94umm6vhs8asdfvano5dgmg5d/3qnd44uusm28lr
--2021-03-24 18:07:41--  https://doc-0k-58-docs.googleusercontent.com/docs/securesc/eg3dk9k94umm6vhs8asdfvano5dgmg5c
Resolving doc-0k-58-docs.googleusercontent.com (doc-0k-58-docs.googleusercontent.com)... 74.125.140.132, 2a00:1450:4
Connecting to doc-0k-58-docs.googleusercontent.com (doc-0k-58-docs.googleusercontent.com)|74.125.140.132|:443... cor
HTTP request sent, awaiting response... 302 Found
Location: https://docs.google.com/nonceSigner?nonce=tlo6dhknou04a&continue=https://doc-0k-58-docs.googleusercontent.
--2021-03-24 18:07:42--  https://docs.google.com/nonceSigner?nonce=tlo6dhknou04a&continue=https://doc-0k-58-docs.goo
Connecting to docs.google.com (docs.google.com)|64.233.167.102|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://doc-0k-58-docs.googleusercontent.com/docs/securesc/eg3dk9k94umm6vhs8asdfvano5dgmg5d/3qnd44uusm28lr
--2021-03-24 18:07:42--  https://doc-0k-58-docs.googleusercontent.com/docs/securesc/eg3dk9k94umm6vhs8asdfvano5dgmg5c
Connecting to doc-0k-58-docs.googleusercontent.com (doc-0k-58-docs.googleusercontent.com)|74.125.140.132|:443... cor
HTTP request sent, awaiting response... 200 OK
Length: 5547 (5.4K) [text/x-python]
Saving to: ‘yolo2tfrecord_fixes.py’

yolo2tfrecord_fixes 100%[===================>]   5.42K  --.-KB/s    in 0s

2021-03-24 18:07:42 (46.6 MB/s) - ‘yolo2tfrecord_fixes.py’ saved [5547/5547]
```

```
# Generate the tfrecords for each split & save them to drive

%cd /content/raw-data/

!python /content/raw-data/yolo2tfrecord_fixes.py -t /content/raw-data/train.txt -o /content/PPU-6-{imsize}-train.tfrec
!python /content/raw-data/yolo2tfrecord_fixes.py -t /content/raw-data/valid.txt -o /content/PPU-6-{imsize}-valid.tfrec
!python /content/raw-data/yolo2tfrecord_fixes.py -t /content/raw-data/test.txt -o /content/PPU-6-{imsize}-test.tfrecor

!cp /content/PPU-6-{imsize}-train.tfrecord /content/drive/MyDrive/TFM/training-assets
!cp /content/PPU-6-{imsize}-valid.tfrecord /content/drive/MyDrive/TFM/training-assets
!cp /content/PPU-6-{imsize}-test.tfrecord /content/drive/MyDrive/TFM/training-assets
```

```
/content/raw-data
2021-03-24 18:07:50.672682: I tensorflow/stream_executor/platform/default/dso_loader.cc:49] Successfully opened dyna
911/912
TFRecord saved, creating label_name.pbtxt
Conversion Done
2021-03-24 18:07:55.437159: I tensorflow/stream_executor/platform/default/dso_loader.cc:49] Successfully opened dyna
274/275
TFRecord saved, creating label_name.pbtxt
Conversion Done
2021-03-24 18:07:58.671988: I tensorflow/stream_executor/platform/default/dso_loader.cc:49] Successfully opened dyna
519/520
TFRecord saved, creating label_name.pbtxt
Conversion Done
```

```
# check that the tfrecord has the appropiate info
for i, example in enumerate(tf.python_io.tf_record_iterator("/content/drive/MyDrive/TFM/training-assets/PPU-6-512-trai
    pass
print(str(i+1) + ' items in train tfrecord')

for i, example in enumerate(tf.python_io.tf_record_iterator("/content/drive/MyDrive/TFM/training-assets/PPU-6-512-vali
    pass
```

```
print(str(i+1) + ' items in valid tfrecord')

for i, example in enumerate(tf.python_io.tf_record_iterator("/content/drive/MyDrive/TFM/training-assets/PPU-6-512-test
    pass
print(str(i+1) + ' items in test tfrecord')
```

```
912 items in train tfrecord
275 items in valid tfrecord
520 items in test tfrecord
```

```
# Create a yaml file with some net properties

with open('/content/raw-data/obj.names','r') as file:
    names = [str(i) + ': ' + name.replace('\n',', ') for i, name in enumerate(file.readlines())]

filecontent = f'''\
num_classes: {len(names)}
var_freeze_expr: '(efficientnet|fpn_cells|resample_p6)'
label_map: {"{" + "".join(names) + "}"}
moving_average_decay: 0'''

with open('/content/PPU-6-config.yaml','w') as file:
    file.write(filecontent)

!cp /content/PPU-6-config.yaml /content/drive/MyDrive/TFM/training-assets
```

Start network

```
train_pattern = f'PPU-6-{imsize}-train.tfrecord'
train_images_per_epoch = 912 // 64 * 64             # round to batchsize

valid_pattern = f'PPU-6-{imsize}-valid.tfrecord'
valid_images_per_epoch = 275 // 8 * 8               # round to batchsize

print('Train images_per_epoch = {}'.format(train_images_per_epoch))
print('Valid images_per_epoch = {}'.format(valid_images_per_epoch))
```

```
Train images_per_epoch = 896
Valid images_per_epoch = 272
```

```
# Train the efficientdet
%cd /content/drive/MyDrive/training-effdet/automl/efficientdet

# Touch the logfile to ensure it exists & same for model dir
!touch pablos-logfile.txt
!mkdir /content/drive/MyDrive/training-effdet/PPU-6-512-effdet-d0

# Download the model if neccessary
MODEL = 'efficientdet-d0'
if MODEL not in os.listdir():
    !wget https://storage.googleapis.com/cloud-tpu-checkpoints/efficientdet/coco/{MODEL}.tar.gz
    !tar xf {MODEL}.tar.gz

# Train!
!python main.py \
    --mode=train_and_eval \
    --train_file_pattern=/content/drive/MyDrive/TFM/training-assets/{train_pattern} \
    --val_file_pattern=/content/drive/MyDrive/TFM/training-assets/{valid_pattern} \
    --model_name={MODEL} \
    --model_dir=/content/drive/MyDrive/training-effdet/PPU-6-512-effdet-d0 \
    --ckpt={MODEL} \
    --train_batch_size=64 \
    --eval_batch_size=8 \
    --eval_samples={valid_images_per_epoch} \
    --num_examples_per_epoch={train_images_per_epoch} \
    --num_epochs=99999999  \
    --hparams=/content/drive/MyDrive/TFM/training-assets/PPU-6-config.yaml \
    --run_epoch_in_child_process=True \
    >> pablos-logfile-effdet-d0-train.txt
```

```
# Resume training
%cd /content/drive/MyDrive/training-effdet/automl/efficientdet

MODEL = 'efficientdet-d0'
if MODEL not in os.listdir():
    !wget https://storage.googleapis.com/cloud-tpu-checkpoints/efficientdet/coco/{MODEL}.tar.gz
    !tar xf {MODEL}.tar.gz

!python main.py \
    --mode=train_and_eval \
    --train_file_pattern=/content/drive/MyDrive/TFM/training-assets/{train_pattern} \
    --val_file_pattern=/content/drive/MyDrive/TFM/training-assets/{valid_pattern} \
    --model_name={MODEL} \
    --model_dir=/content/drive/MyDrive/training-effdet/PPU-6-512-effdet-d0 \
    --ckpt=/content/drive/MyDrive/training-effdet/PPU-6-512-effdet-d0 \
    --train_batch_size=64 \
    --eval_batch_size=8 \
    --eval_samples={valid_images_per_epoch} \
    --num_examples_per_epoch={train_images_per_epoch} \
    --num_epochs=99999999  \
    --hparams=/content/drive/MyDrive/TFM/training-assets/PPU-6-config.yaml \
    --run_epoch_in_child_process=True \
    >> pablos-logfile-effdet-d0-train.txt
```

Testing

```
# IF YOU CAME STRAIGHT HERE RUN THIS...

%cd /content/drive/MyDrive/training-effdet/automl/efficientdet
MODEL = 'efficientdet-d0'
if MODEL not in os.listdir():
    !wget https://storage.googleapis.com/cloud-tpu-checkpoints/efficientdet/coco/{MODEL}.tar.gz
    !tar xf {MODEL}.tar.gz
```

```
/content/drive/MyDrive/training-effdet/automl/efficientdet
```

```
# This code box to test on default backup dir, else skip to next one

%cd /content/drive/MyDrive/training-effdet/automl/efficientdet

!python main.py --mode=eval \
      --model_name=efficientdet-d0 \
      --model_dir=/content/drive/MyDrive/training-effdet/backup_dir/backup  \
      --val_file_pattern=/content/drive/MyDrive/training-effdet/automl/efficientdet/tfrecord/PPU-6-train.tfrecord \
      --hparams=/content/drive/MyDrive/TFM/training-assets/PPU-6-config.yaml \
      --eval_batch_size=1 \
      --eval_samples=500 \
```

```
# Testing checkpoint on remote folder for test split
# 1. Take files away to some folder
# 2. Modify */checkpoint with the path of the model without extension in both rows
# 3. Run this and hope for the best

%cd /content/drive/MyDrive/training-effdet/automl/efficientdet

!python main.py --mode=eval \
      --model_name=efficientdet-d0 \
      --model_dir=/content/drive/MyDrive/ED-D0-backup_temp  \
      --val_file_pattern=/content/drive/MyDrive/TFM/training-assets/PPU-6-512-test.tfrecord \
      --hparams=/content/drive/MyDrive/TFM/training-assets/PPU-6-config.yaml \
      --eval_batch_size=1 \
      --eval_samples=520 \
      >> /content/logfile.txt
```

```
!cp /content/logfile.txt /content/drive/MyDrive/TFM/effdet-d0-map.txt
```

Inference on pictures

```
# copy all test images to a folder where no text files exist
import os

!mkdir /content/test-notxt/

piclist = [pic for pic in os.listdir('/content/raw-data/test') if pic[-1]!='t']
for idx, pic in enumerate(piclist):
    img = cv2.imread(f'/content/raw-data/test/{pic}')
    img = cv2.resize(img, (imsize,imsize))
    cv2.imwrite(f'/content/test-notxt/{pic}', img)
    print(f'\r Resizing & copying {split} split... {idx+1} converted. Current size {img.shape}', end='')

```

```
mkdir: cannot create directory ‘/content/test-notxt/’: File exists
 Resizing & copying test split... 520 converted. Current size (512, 512, 3)
```

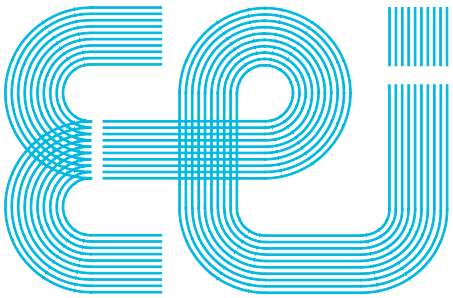
```
# Inference of pictures
%cd /content/drive/MyDrive/training-effdet/automl/efficientdet



# First export a saved model.
saved_model_dir = '/content/savedmodel'
!rm -rf {saved_model_dir}

ckpt_path = '/content/gdrive/MyDrive/TFM/efficientDet/models/efficientdet-d0-finetune-allnighter'

!python model_inspect.py --runmode=saved_model \
    --model_name=efficientdet-d0 \
    --ckpt_path=/content/drive/MyDrive/ED-D0-backup_temp \
    --saved_model_dir={saved_model_dir} \
    --hparams=/content/drive/MyDrive/TFM/training-assets/PPU-6-config.yaml



# Then run saved_model_infer to do inference.
# Notably: batch_size, image_size must be the same as when it is exported.
serve_image_out = '/content/serve_image_out'
!mkdir {serve_image_out}

!python model_inspect.py --runmode=saved_model_infer \
    --saved_model_dir={saved_model_dir} \
    --model_name=efficientdet-d0 \
    --input_image='/content/test-notxt/*'  \
    --output_image_dir={serve_image_out} \
    --min_score_thresh=0  \
    --max_boxes_to_draw=10 \
    --hparams=/content/drive/MyDrive/TFM/training-assets/PPU-6-config.yaml

```

```
!zip -q -r /content/drive/MyDrive/TFM/effdet-d0-infer.zip /content/serve_image_out
```

```
!zip -r /content/effdet-D0-minitest.zip /content/serve_image_out
```

remove stuff from colab once finished training

```
# # # !rm -r /content/drive/MyDrive/training-effdet/automl
# # # !rm -r /content/drive/MyDrive/training-effdet/
```



Escola de Enxeñería Industrial

TRABALLO FIN DE MESTRADO

Convolutional neural networks for object detection in subterranean environments

Master's Degree in Industrial Engineering

Document

APPENDIX II: PERCEPTION LAYER SOURCE CODE

Universidade Vigo

CONTENTS

pabsan-0 Update pabloWorldMap.py ... 5 minutes ago History

..	
README.md	28 minutes ago
RedmiNote9Pro.npz	23 hours ago
data1.txt	23 hours ago
main-multiprocessing.py	16 minutes ago
main.py	23 hours ago
pabloCameraPoseAruco.py	23 hours ago
pabloDarknetInference.py	10 minutes ago
pabloItem2CameraPosition.py	23 hours ago
pabloKalman.py	23 hours ago
pabloKalman_yawEuclidean.py	11 minutes ago
pabloWorldMap.py	5 minutes ago
requirements.txt	33 minutes ago

README.md

Final implementation of object localizer

This directory contains source code plus assets for running the proposed perception layer for item search. For this, make sure all libraries and requisites are set up, hardcode the image source inside `main-multiprocessing.py` and then run the script normally from a terminal window.

In this directory

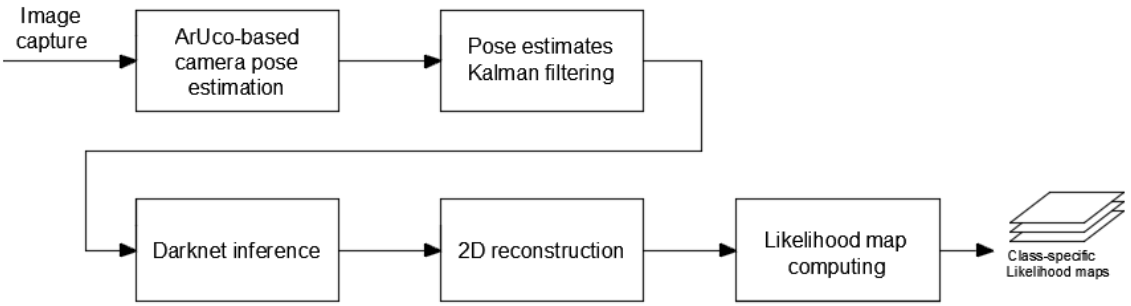
- RedmiNote9Pro.npz: File containing camera calibration parameters. Use yours.
- data1.txt: (deprecated) Experimental data for estimating the sensor noise matrix towards Kalman filters.
- main-multiprocessing.py: Main script calling the rest of the program. Holds some config under main.
- main.py: (deprecated) Old version of main-multiprocessing, running the same source but sequentially instead of in parallel. You can use this, though it is slower and is a few commits behind.
- pabloCameraPoseAruco.py: Custom python module implementing the Camera pose detection from ArUcos.
- pabloDarknetInference.py: Custom python module implementing darknet inline inference.
- pabloItem2CameraPosition.py: Custom python module implementing width-based item localization from bboxes.
- pabloKalman.py: (deprecated) Custom python module implementing the camera pose Kalman filter, buggy.
- pabloKalman_yawEuclidean.py: Custom python module implementing the camera pose Kalman filter, debugged.
- pabloWorldMap.py: Custom python module implementing likelihood map drawing to localize items in the environment.
- requirements.txt: List of required python dependencies.

To use the perception layer

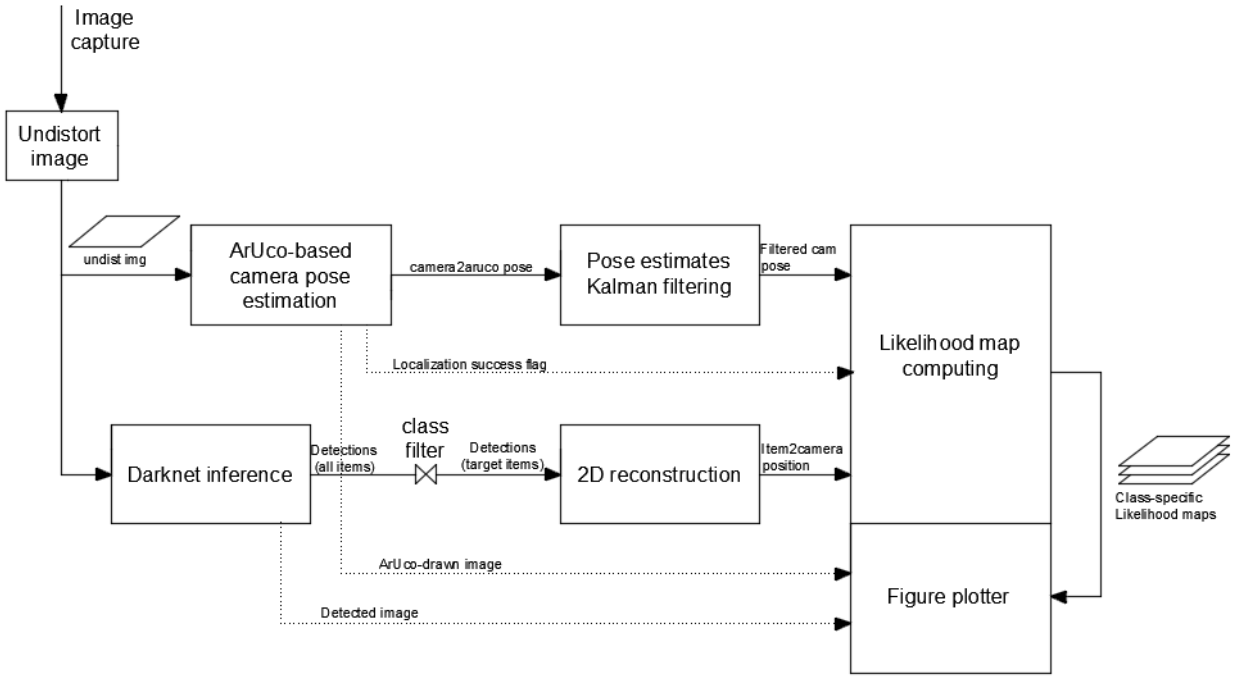
- Make sure the following is properly installed
  - Darknet (with GPU support)
  - The python modules listed at `requirements.txt`
- Calibrate your camera with the tools at ../camera-calibration
- Clone the percepion-layer-final directory (this directory)
  - Drop your camera's calibration file in the cloned dir
  - Check the mains of the file `main-multiprocessing.py` and hardcode-adjust:
    - Feed IP source
    - Camera calibration file
    - Darknet installation path
- Boot your camera live stream to a local IP
- Run `main-multiprocessing.py` to start live-finding objects

Media

Sequential perception layer overview



Multiprocessing perception layer overview



```
################################################################################
#### main-multiprocessing.py                                                ####
#### Pablo Santana - CNNs for object detection in subterranean environments ####
################################################################################
# p0 (main)         p1                   p2                      p5
# ┌─────────────┐   ┌──────────────────┐   ┌──────────────────┐    ┌──────────────┐
# │ImageCapture│ q1 │Estimate Camera│ q3 │Kalman-Filter    │q4  │Draw Worldmap │
# │&& Undistort├──->│Pose From ArUco├──->│Camera Pose      ├──┬>│with cam FoV &│
# │            ||   │markers        │    │                 │  | │item positions│
# └─────────────┘|   └──────────────────┘   └──────────────────┘  | └──────────────┘
#  └YOU'RE HERE│   p3                   p4                      |
#              |   ┌──────────────────┐   ┌──────────────────┐   |
#              |q2 │Object detection│q5 │Estimate item    │q6|
#              └──>│with Darknet    ├──->│position w.r to  ├──┘
#                  │(CNN)           |   │the camera       |
#                  └──────────────────┘   └──────────────────┘
#
# Run with ``$ python3 main-multiprocessing.py`
# Make sure all submodules and assets are available to the main process.

import numpy as np
import matplotlib.pyplot as plt
from numpy.linalg import inv
import cv2
import cv2.aruco as aruco
import math
import os
import time
import pprint
import psutil
import ctypes
import sys

import pabloCameraPoseAruco
import pabloKalman_yawEuclidean as pabloKalman
# import pabloKalman as pabloKalman

import pabloDarknetInference
import pabloItem2CameraPosition
import pabloWorldMap

from pabloCameraPoseAruco import ArucoMarker
from pabloKalman_yawEuclidean import KalmanFilter
# from pabloKalman import KalmanFilter
from pabloWorldMap import worldMap

from multiprocessing import Process, Queue

"""
```

```
This program runs live darknet inference from images taken from a capture device
(an IP camera or a webcam) and, from the detections of specified items, builds
a likelihood map of a 2D environment in which the areas where items are found
are highlighted in white. Requires a standalone installation of darknet with
GPU support.
"""

def trackingMessage(id, parentName, outputMetricName, outputMetric, now):
    """
    Used for displaying compact messages of time and output to track the
    performance of each parallel process. Mainly a debugging tool.
    """
    if timeAllModules and not showAllModulesOutput:
        # Display elapsed time per module processing
        str_out = f'{parentName} took {time.time()-now} seconds'

    elif showAllModulesOutput:
        # Format result for pretty printing
        pretty_result_string = pprint.pformat(outputMetric)

        # Print all in a single message to avoid process print overlapping
        str_out = f'''\n\n{id * 100}
        \r\r\r>>> Module {parentName} says: ({time.time()})')
        \r\r\r>>  Elapsed time: {time.time()-now} seconds.
        \r\r\r>>  {outputMetricName} value: \n{pretty_result_string}
        '''
    else:
        str_out = ''
    return str_out



def parallelCameraPoseAruco(markerDict, cameraMatrix, qframeUndist_2aruco, qCamera2ArUcoPose, qPlotland_1, qLocalizationSuccess):
    """
    Get camera pose from ArUco markers. Returns dict with each marker data.
    """
    # Aruco dict. Selecting subset avoids running into unexpected IDs!
    # This should be working but isnt!! cant generate small size dict from 250x250
    num_markers = len(markerDict)                       # Bold hypotheses
    aruco_dict = aruco.Dictionary_get(aruco.DICT_4X4_250)
    aruco_dict = cv2.aruco_Dictionary.create_from(num_markers, 4, aruco_dict)
    print('LOOK HERE, we are using this # markers ',num_markers)

    # Required aruco params (default, braindead copypaste)
    parameters =  aruco.DetectorParameters_create()

    # Infer which markers are susceptible to be found in the world
    env_markers = [markerDict[i].id for i in markerDict.keys()]


    # Hardcoded marker size and fake distortion coefs (for usage on undist image)
    fakeDist = np.array([[1e-5,1e-5,1e-5,1e-5,1e-5]])
    size_of_marker = 150

    # Dummy fake detection to cover if the first iteration fails to find an ArUco
    camera2aruco_pose = {}
    camera2aruco_pose[0] = {
                        'id':0,
                        'pose2aruco': np.array([0,0,0]),
                        'pose2world': np.array([0,0,0]),
                        }

    while 1:
        # time tracker
        now = time.time()

        # Get an undistorted frame from the input queue
        frameUndist = qframeUndist_2aruco.get()

```

```
        # Try to estimate camera pose
        try:
            camera2aruco_pose, frameOutArUco = pabloCameraPoseAruco.main(
                frameUndist,
                aruco_dict, parameters,
                env_markers, size_of_marker,
                cameraMatrix, fakeDist,
                markerDict,
                debug=False
                )
            LocalizationSuccess = 1
        except pabloCameraPoseAruco.CorruptedArUcoDetections:
            # Either no arucos in sight or an unexpected one: return previous pose
            frameOutArUco = frameUndist
            LocalizationSuccess = 0

        # Send estimation through output queue
        qCamera2ArUcoPose.put(camera2aruco_pose)

        # Send image and success flag through output aux queue
        qPlotland_1.put(frameOutArUco)
        qLocalizationSuccess.put(LocalizationSuccess)

        # Time and output tracking (enable/disable) at MAIN
        print(trackingMessage('1', 'CameraPoseAruco', 'camera2aruco_pose', camera2aruco_pose, now), end='')



def parallelKalman(markerDict, qCamera2ArUcoPose, qCameraPoseFiltered):
    """
    KalmanFilters a live Pose input from multiple ArUcos at once
    """
    # Import model matrices (hardcoded)
    A, B, C, _, Rw, Rv = pabloKalman.modelImport(dataSample='data1.txt')

    # Instance the handmade Kalman filter
    KF = KalmanFilter(A, B, C, [], Rw, Rv)

    while 1:
        # time tracker
        now = time.time()

        # Get unfiltered pose dict from input queue
        camera2aruco_pose = qCamera2ArUcoPose.get()

        # Kalman-filter the current measurement
        cameraPoseFiltered = pabloKalman.main(
            KF,
            camera2aruco_pose,
            markerDict
            )

        # Drop filtered pose dict to output queue
        qCameraPoseFiltered.put(cameraPoseFiltered)

        # Time and output tracking (enable/disable) at MAIN
        print(trackingMessage('2', 'KalmanFilter', 'cameraPoseFiltered', cameraPoseFiltered, now), end='')



def parallelDarknetInference(qframeUndist_2darknet, qItemDetections, qPlotland_2):
    """
    Performs darknet inference on image via python bindings
    """
    # Change dirs for darknet importing
    sys.path.append(os.getcwd())
    os.chdir('/home/pablo/YOLOv4/darknet')
    sys.path.insert(1, '/home/pablo/YOLOv4/darknet')

```

```
    # Initialize the network
    network, class_names, class_colors = pabloDarknetInference.networkBoot()

    while 1:
        # time tracker
        now = time.time()

        # Get undistorted frame from input queue
        frameUndist = qframeUndist_2darknet.get()

        # Image detection with darknet!
        frameDetected, itemDetections = pabloDarknetInference.imageDetection(
            frameUndist,
            network,
            class_names,
            class_colors,
            thresh=0.3
            )

        # Drop detected + input frames plus detection output to output queue
        qItemDetections.put([frameDetected, frameUndist, itemDetections])

        # Send image through output aux queue
        qPlotland_2.put(frameDetected)

        # Time and output tracking (enable/disable) at MAIN
        print(trackingMessage('3','DarknetInference', 'itemDetections', itemDetections, now), end='')



def parallelItem2CameraPosition(itemListAndSize, cameraMatrix, qItemDetections, qItem2CameraPosition):
    """
    Using an item's width and given bounding boxes, infer its position wrt camera
    """
    while 1:
        # time tracker
        now = time.time()

        # Get detected + input frames plus detection output input queue
        frameDetected, frameUndist, itemDetections = qItemDetections.get()

        # Get the horizontal position of the item with respect to the camera
        item2CameraPosition = pabloItem2CameraPosition.getItemPosition(
            frameDetected,
            frameUndist,
            itemDetections,
            itemListAndSize,
            cameraMatrix
            )

        # Drop item position to output queue
        qItem2CameraPosition.put(item2CameraPosition)

        # Time and output tracking (enable/disable) at MAIN
        print(trackingMessage('4','item2CameraPosition', 'item2CameraPosition', item2CameraPosition, now), end='')



def parallelWorldMap(itemListAndSize, qCameraPoseFiltered, qItem2CameraPosition, qPlotland_1, qPlotland_2, qLocalizationSuccess):
    """
    Given camera Pose and object positions with respect to it, draw a world Map
    in which a likelihood map for the presence of an item at a given location is
    shown.
    """

    # Instance the worldMap class
    map = worldMap(
        numItems=len(itemListAndSize),
        mapsize_XZ_cm=[84+277,676]
```

```
        )

    # Small-time configuration of dissapearance and appearance rates
    map.disappearanceRate = 0.0002
    map.appearanceRate    = 0.0005

    # adjust worlFrame Coordinates wr to map origin
    map.world2map_XZ_cm = [84, 0]

    while 1:
        # time tracker
        now = time.time()

        # Get item and camera poses from input queues
        cameraPoseFiltered = qCameraPoseFiltered.get()
        item2CameraPosition = qItem2CameraPosition.get()

        # Get data from auxiliary queues
        frameArUco = qPlotland_1.get()
        frameDetected = qPlotland_2.get()
        LocalizationSuccess = qLocalizationSuccess.get()

        # if localization has been successful do map detections else dont
        if LocalizationSuccess == 1:
            # Fix coordinate frame consistency issues. Units to mm-deg
            item2CameraPositionCompliant, cameraPoseFilteredCompliant \
                = pabloWorldMap.cameraCoordinateComply(
                    item2CameraPosition,
                    cameraPoseFiltered
                    )

            # Update the map with detections!
            map.update(
                item2CameraPositionCompliant,
                cameraPoseFilteredCompliant
                )



            # Show if localization is okay else freeze
            cv2.imshow('Field of view', map.fovMask)
            cv2.imshow('Instantaneous discovery', np.array(map.discoveryMask, np.uint8))
            #cv2.imshow('Likelihood map', map.map)
            cv2.imshow('Likelihood map', map.map_fov)

        # Always show
        cv2.imshow('ArUco pose estimator', frameArUco)
        cv2.imshow('Detection', frameDetected)
        cv2.waitKey(1)

        if timeAllModules:
            print(f'\tMapping took {time.time()-now} seconds')



if __name__ == '__main__':

    # Do you want process time's being shown? ATT. LOTS OF PRINTOUTS
    timeAllModules, showAllModulesOutput = False, True

    # Video feed source
    cap = cv2.VideoCapture('https://192.168.0.102:8085/video')

    # Load camera calibration matrices
    with np.load('RedmiNote9Pro.npz') as X:
        cameraMatrix, distCoeffs = [X[i] for i in('cameraMatrix', 'distCoeffs')]
    fakeDist = np.array([[1e-5,1e-5,1e-5,1e-5,1e-5]])

    # Kalman Rv, preimported here to assign Rv to markerDict, again on Kalman module
```

```
    #_,_,_,_,_, Rv = pabloKalman.modelImport(dataSample='data1.txt')
    # Use this to solve non-euclidean angle issue
    Rv = np.eye(4)

    # Marker coordinates in the world, inc Kalman Noise matrix
    # angle measured in negative y!
    markerDict_BedroomCorner = {
        0: ArucoMarker(0, [    0,    0,   0], Rv),
        1: ArucoMarker(1, [ -640,    0,   0], Rv),
        2: ArucoMarker(2, [-1100,  370,  -90], Rv),
        }

    markerDict_LivingRoom = {
        0: ArucoMarker(0, [    0,    0,   0], Rv),
        1: ArucoMarker(1, [ 1500,    0,   0], Rv),
        2: ArucoMarker(2, [ 2770, 1260,  90], Rv),
        3: ArucoMarker(3, [ 2770, 3070,  90], Rv),
        4: ArucoMarker(4, [ 2770, 4980,  90], Rv),
        5: ArucoMarker(5, [ 2770, 6200,  90], Rv),
        6: ArucoMarker(6, [ 1400, 6760, 180], Rv),
        7: ArucoMarker(7, [ -240, 4890, -90], Rv),
        8: ArucoMarker(8, [ -440, 4570, 180], Rv),
        9: ArucoMarker(9, [ -840, 2740, -90], Rv),
        }

    # Select world. REMEMBER TO CHANGE MAP SIZE
    markerDict = markerDict_LivingRoom


    # list of items of interest and their estimate width
    itemListAndSize = {'bottle': 100
            }

    # Main data queues
    q1 = qframeUndist_2aruco     = Queue(maxsize=1)
    q2 = qframeUndist_2darknet   = Queue(maxsize=1)
    q3 = qCamera2ArUcoPose       = Queue(maxsize=1)
    q4 = qCameraPoseFiltered     = Queue(maxsize=1)
    q5 = qItemDetections         = Queue(maxsize=1)
    q6 = qItem2CameraPosition    = Queue(maxsize=1)

    # Auxiliar data bypassing
    # Move detected frame to be plotted (cv2 cant handle multiprocess imshow)
    q7 = qPlotland_1             = Queue(maxsize=1)
    q8 = qPlotland_2             = Queue(maxsize=1)
    #
    # Successful location frame to not print detections if cameraPose ambiguous
    q9 = qLocalizationSuccess    = Queue(maxsize=1)


    # Defining the processes. Placeholders to be able to check on them from mains
    p1 = Process(target=parallelCameraPoseAruco,
                 args=(markerDict,
                       cameraMatrix,
                       qframeUndist_2aruco,
                       qCamera2ArUcoPose,
                       qPlotland_1,
                       qLocalizationSuccess,))

    p2 = Process(target=parallelKalman,
                 args=(markerDict,
                       qCamera2ArUcoPose,
                       qCameraPoseFiltered,))

    p3 = Process(target=parallelDarknetInference,
                 args=(qframeUndist_2darknet,
                       qItemDetections,
                       qPlotland_2,))

```

```
    p4 = Process(target=parallelItem2CameraPosition,
                 args=(itemListAndSize,
                       cameraMatrix,
                       qItemDetections,
                       qItem2CameraPosition,))

    p5 = Process(target=parallelWorldMap,
                 args=(itemListAndSize,
                       qCameraPoseFiltered,
                       qItem2CameraPosition,
                       qPlotland_1,
                       qPlotland_2,
                       qLocalizationSuccess,))

    # Start all the processes
    for p in [p1,p2,p3,p4,p5]:
        p.start()

    # For FPS tracking (if option selected under __main__)
    now = time.time()

    while 1:
        # Read a frame from the capture object correct image distortions
        ret, frameIn = cap.read()
        frameUndist = cv2.undistort(frameIn, cameraMatrix, distCoeffs, None)

        # This allows real-time frames without clogging the queues
        # Once these queues hold values the rest of the pipeline starts working
        if qframeUndist_2aruco.empty() & qframeUndist_2darknet.empty():
            qframeUndist_2aruco.put(frameUndist)
            qframeUndist_2darknet.put(frameUndist)

            # Just for time tracking (if option selected under __main__)
            if timeAllModules:
                print(f'Doing {1/(time.time() - now)} FPS!')
                now = time.time()

            # Make sure that the CUDNN issue solution is cleanly shown on exception
            # This will allow the error-solution message to be seen easily.
            if not p3.is_alive():
                print('Run the previous command on this terminal. \
                      Press ^C (ctrl+C) to exit and ignore the error message that will pop up.')
                quit()
```

```
##############################################################################
#    main-multiprocessing.py / pabloCameraPoseAruco                          #
#    Pablo Santana - CNNs for object detection in subterranean environments  #
##############################################################################
# p0 (main)         p1                   p2                      p5
# ┌─────────────┐    ┌─────────────────┐    ┌──────────────────┐    ┌──────────────┐
# │ImageCapture│ q1 │Estimate Camera│ q3 │Kalman-Filter    │q4  │Draw Worldmap │
# │&& Undistort├┬──>│Pose From ArUco├──>│Camera Pose      ├──┬>│with cam FoV &│
# │            ││   │markers        │    │                 │  │ │item positions│
# └─────────────┘│   └─────────────────┘    └──────────────────┘  │ └──────────────┘
#                │   p3   └YOU'RE HERE   p4                     │
#                │   ┌─────────────────┐    ┌──────────────────┐  │
#                │q2 │Object detection│q5 │Estimate item    │q6│
#                └──>│with Darknet    ├──>│position w.r to  ├──┘
#                    │(CNN)           │    │the camera       │
#                    └─────────────────┘    └──────────────────┘
#
#

import numpy as np
import cv2
import cv2.aruco as aruco
import math
import os
import pprint


class CorruptedArUcoDetections(Exception):
    def __init__(self, message):
        self.message = message


class ArucoMarker(object):
    '''
    Implements a marker object to store its pose and kalman gain.

    ARGS:
        id:   ID of this ARUCO marker.
        pose: Provide as list [x, z, yaw]. Pose of this ARUCO marker.
        Rv:   Measurement noise covariance matrix of the camera seeing this ID.
    '''

    def __init__(self, id, pose, Rv=[]):
        # Store marker ID, pose
        self.id = id
        self.pose = pose

        # Store Kalman gain and Measurement Noise Covariance
        #self.L = np.zeros([6, 3], dtype=np.float32)
        # IF USING COS-SIN KALMAN FILTER
```

```
        self.L = np.zeros([4, 4], dtype=np.float32)
        self.R = Rv

        # Compute and store the rotation matrix of this marker
        x = pose[0]
        z = pose[1]
        yaw = np.deg2rad(pose[2])
        self.rot = np.array([[np.cos(yaw), -np.sin(yaw),   x],\
                             [np.sin(yaw),  np.cos(yaw),   z],\
                             [          0,            0,   1]])

    def transform2D_toWorld(self, pose_camera2marker):
        '''
        Applies homogeneous 2D transform to project pose of camera with
        respect to a marker and obtain the pose of the camera with respect to
        the origin coordinate frame.
        '''

        # Convert input [x, z, yaw] to [x, z, 1] for 2D conversion
        B = pose_camera2marker * np.array([1,1,0]) + np.array([0,0,1])

        # Apply homogeneous transformation matrix to get [x, z] w.r to origin
        A = self.rot @ B

        # Add angle camera_/marker + marker_/origin
        yaw = pose_camera2marker[2] + self.pose[2]

        # Recover yaw angle and return array in form [x, z, yaw]
        # returning   X & Z Term     +       angle
        pose_camera2world = A * np.array([1,1,0]) + yaw * np.array([0,0,1])

        return pose_camera2world




def detectMarkers(frame, aruco_dict, parameters, env_markers, size_of_marker, mtx, dist):
    ''' Use to compute the translation and rotation vectors of ArUcos in an
    image, including their 2D projections (corners)
    '''
    # Detect ArUcos from grayscaled picture
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    corners, ids, rejectedImgPoints = aruco.detectMarkers(gray, aruco_dict, parameters=parameters)

    # Filter out markers that are not desired to be detected.
    # this is a consequence of poor dictionary choice, but I already got...
    # ... printed arucos for dict 250x250 and i only wanna use the first 10
    try:
        corners = [cor for i,cor in enumerate(corners) if int(ids[i][0]) in env_markers]
        ids = np.array([i for i in ids if int(i[0]) in env_markers])
    except:
        pass

    # If some markers are found in the picture
    if corners != []:
        '''if not all([i in env_markers for i in ids]):
            # If a marker that is not expected appears (noise) raise exception
            # this situation should not happen if aruco_dict is properly defined
            raise CorruptedArUcoDetections('Unexpected ArUco marker spotted.')'''

        # get the rotation and traslation vectors CAMERA -> ARUCO
        rvecs,tvecs,trash = aruco.estimatePoseSingleMarkers(corners, size_of_marker , mtx, dist)
        return corners, ids, rvecs, tvecs

    else:
        raise CorruptedArUcoDetections('No ArUcos being detected.')


def drawDetectedMarkers(frame, corners, ids, rvecs, tvecs, mtx, dist):
```

```
    ''' Draw aruco markers and 3D axes on picture
    '''
    frame = aruco.drawDetectedMarkers(frame.copy(), corners, ids)
    for i, j in zip(rvecs, tvecs):
        frame = aruco.drawAxis(frame, mtx, dist, i, j, 50)
    return frame


def getCamera2MarkerPose(ids, rvecs, tvecs, markers):
    ''' Gets the camera pose with respect each marker in any set of detected IDs.
    '''
    # Define placeholder for detection data
    pose_camera2marker = {}

    for idx, markerID in enumerate(ids):
        # get the tvexs and rvecs of this particular marker
        tvec = tvecs[idx][0]
        rvec = rvecs[idx][0]

        # Get rotation matrix from object coordinates to camera coordinates
        rot = cv2.Rodrigues(rvec)[0]

        # Assemble translation matrix and use it to compute camera|markerID
        T = np.eye(4)
        T[0:3, 0:3] = rot
        T[0:3, 3]   = tvec
        tvec_camera_from_camera = np.array([0,0,0,1])
        x, y, z, _ = np.matmul(np.linalg.inv(T), tvec_camera_from_camera)

        eul   = -cv2.decomposeProjectionMatrix(T[:3,:])[6]
        yaw   =  eul[1,0]
        # pitch = (eul[0,0]+(90))*math.cos(eul[1,0])
        # roll  = (-(90)-eul[0,0])*math.sin(eul[1,0]) + eul[2,0]

        # Add each marker's pose to the detection dictionary
        # transform2D to relate the position wr this aruco to the world frame
        pose_camera2marker[idx] =  {#'raw': [int(markerID), x, y, z, yaw, roll, pitch],
                            'id': int(markerID),
                            'pose2aruco': np.array([x, z, yaw]),
                            'pose2world': markers[int(markerID)].transform2D_toWorld(np.array([x, z, yaw]))
                            }

    return pose_camera2marker



# CALL FROM LOOP
def main(frame, aruco_dict, parameters, env_markers, size_of_marker, mtx, dist, markers, debug=False):
    # placeholder in case of corner break later
    camera2aruco_pose = None

    # get the arucos spatial information, if error-free then compute camera poses
    corners, ids, rvecs, tvecs = detectMarkers(frame, aruco_dict, parameters, env_markers, size_of_marker, mtx, dist)
    camera2aruco_pose = getCamera2MarkerPose(ids, rvecs, tvecs, markers)

    # draw and output a picture with the detected markers
    frameOut = drawDetectedMarkers(frame, corners, ids, rvecs, tvecs, mtx, dist)

    # terminal + video display for tracking and debugging
    if debug != False:
        cv2.imshow('', frame)
        cv2.waitKey(1)
        pprint.pprint(camera2aruco_pose, width=1)

    return camera2aruco_pose, frameOut
```

```python

# STANDALONE TEST
def standalone():
    # Video feed source
    cap = cv2.VideoCapture('https://192.168.0.103:8085/video')

    # Load camera calibration matrices
    with np.load('RedmiNote9Pro.npz') as X:
        mtx, dist = [X[i] for i in('cameraMatrix', 'distCoeffs')]

    # Defining all the aruco-related stuff
    # marker map (implementation supports adding kalman Rv, but omitted here)
    markers = {
        0: ArucoMarker(0, [    0,    0,   0]),
        1: ArucoMarker(1, [ -640,    0,   0]),
        2: ArucoMarker(2, [-1100,  370,  -90]),
        }
    env_markers = [markers[i].id for i in markers.keys()]
    size_of_marker = 150
    aruco_dict = aruco.Dictionary_get(aruco.DICT_4X4_250)
    parameters =  aruco.DetectorParameters_create()

    while 1:
        ret, frame = cap.read()
        corners, ids, rvecs, tvecs = detectMarkers(frame, aruco_dict, parameters, env_markers, size_of_marker, mtx, dist)

        if corners != None:
            camera2aruco_pose = getCamera2MarkerPose(ids, rvecs, tvecs, markers)

        cv2.imshow('', drawDetectedMarkers(frame, corners, ids, rvecs, tvecs, mtx, dist))
        cv2.waitKey(1)

        os.system('clear')
        pprint.pprint(camera2aruco_pose, width=1)

if __name__ == '__main__':
    standalone()
```

```
##############################################################################
#    main-multiprocessing.py / pabloKalman                                   #
#    Pablo Santana - CNNs for object detection in subterranean environments  #
##############################################################################
# p0 (main)         p1                   p2                      p5
# ┌─────────────┐    ┌─────────────────┐    ┌──────────────────┐    ┌──────────────┐
# │ImageCapture│ q1 │Estimate Camera│ q3 │Kalman-Filter    │q4  │Draw Worldmap │
# │&& Undistort├┬──>│Pose From ArUco├──>│Camera Pose      ├┬──>│with cam FoV &│
# │            ││   │markers        │    │yaw EUCLIDEAN    │  | │item positions│
# └─────────────┘|   └─────────────────┘    └──────────────────┘  | └──────────────┘
#                |   p3                   p4    └YOU'RE HERE   |
#                |   ┌─────────────────┐    ┌──────────────────┐  |
#                |q2 │Object detection│q5 │Estimate item    │q6|
#                └──>│with Darknet    ├──>│position w.r to  ├──┘
#                    │(CNN)           |   |the camera       |
#                    └─────────────────┘    └──────────────────┘
#
#

import numpy as np
from numpy.linalg import inv
import os

def angdiff(b1, b2):
        r = (b2 - b1) % np.pi
        if r >= np.pi:
                r -= 2 * np.pi
        return r


def modelImport(sampleTime=0.0386, dataSample=[]):
    # System model

    Ts = sampleTime

    A = np.eye(4)

    B = np.zeros([4,1]);

    C = np.eye(4)

    # lower -> more inertia
    Rw = np.array([\
        [0.05,    0,    0,    0],
        [0,    0.05,    0,    0],
        [0,       0,  0.1,    0],
        [0,       0,    0,  0.1]])

    if dataSample != []:
        data = np.loadtxt(dataSample)
```

```
        x   = data[:,1]
        z   = data[:,2]
        yaw = data[:,3]
        y = np.array([x, z, yaw], dtype=np.float32)
        Rv = np.cov(y)
    else:
        Rv = np.eye(3)

    Rv = np.array([\
        [10,    0,    0,    0],
        [0,    10,    0,    0],
        [0,    0,   10,    0],
        [0,    0,    0,   10]]) * 1

    return A, B, C, None, Rw, Rv


class KalmanFilter(object):
    '''
    Implements a Linear Kalman Filter that can be used online to filter a signal.
    Initialize with system model matrices.

    ARGS:
        A: From the propagation equation for State Space models.
        B: From the propagation equation for State Space models.
        C: From the measurement equation for State Space models.
        D: From the measurement equation for State Space models. NOT USED.
        Q: Process noise covariance matrix.
        R: Measurement noise covariance matrix.
    '''

    def __init__(self, A, B, C, D, Q, R):
        # System model: propagation
        self.A = A
        self.B = B
        self.Q = Q

        # System model: measure
        self.C = C
        self.D = D
        self.R = R

        # Get state and measure vector dimensions from matrices
        n_elements_x = A.shape[0]
        n_elements_y = C.shape[0]

        # Kalman gains
        self.L = np.zeros([n_elements_x, n_elements_y], dtype=np.float32)
        self.P = np.zeros([n_elements_x, n_elements_x], dtype=np.float32)

        # Q coefficient
        self.LL = np.eye(n_elements_x)

        # Initial state vector
        self.xhat = np.zeros([n_elements_x], dtype=np.float32)

        # store blueprints of scalable matrices
        self.C_blueprint = C

    def propagate(self, u):
        # Compute next state with propagation equation
        self.xhat = self.A @ self.xhat + self.B  @ u

        # Compute P
        self.P = self.A @ self.P @ self.A.T + self.LL @ self.Q @ self.LL.T

    def update(self, y):
        # Compute the Kalman gain
        self.L = self.P @ self.C.T @ inv(self.C @ self.P @ self.C.T + self.R)
```

```
        # Correct xhat from weighted sum of current xhat + error * kalman gain
        self.xhat = self.xhat + self.L @ (y - self.C @ self.xhat)

        # Adjust P
        self.P = self.P - self.L @ self.C @ self.P

    def filter_step(self, u, y, ret_xhat=False):
        self.propagate(u)
        self.update(y)
        if ret_xhat == True:
            return self.xhat
        else:
            return self.C @ self.xhat



# CALL FROM LOOP
def main(KF, camera2aruco_pose, markers):
    # KALMAN FILTER ONLINE MODIFICATIONS

    # infer which markers are being seen
    markers_on_sight = [camera2aruco_pose[i]['id'] for i in camera2aruco_pose.keys()]

    # Assemble new kalman filter matrices: Kalman Gain and C matrix
    KF.L = np.hstack([markers[i].L for i in markers_on_sight])
    KF.C = np.vstack([KF.C_blueprint for i in markers_on_sight])

    # Assemble new kalman filter matrix: Measurement noise covariance matrix
    canvas =  np.zeros([len(markers_on_sight)*4, len(markers_on_sight)*4])

    for idx, id in enumerate(markers_on_sight):
        canvas[0+4*idx:4+4*idx, 0+4*idx:4+4*idx] = markers[id].R
    KF.R = canvas


    ''' DOESNT WORK - KEPT FOR REFERENCE
    # Linearize angles with respect to current average
    # Compute average from all measurements
    angle_list = [np.deg2rad(camera2aruco_pose[i]['pose2world'][2]) for i,j in enumerate(markers_on_sight)]
    cos_avg = np.average([np.cos(i) for i in angle_list])
    sin_avg = np.average([np.sin(i) for i in angle_list])
    angle_avg = np.arctan2(sin_avg, cos_avg)
    #
    # compute the angle mean and subtract it from each measurement
    for i, j in enumerate(markers_on_sight):
        camera2aruco_pose[i]['pose2world'][2] = np.rad2deg(angle_list[i] - angle_avg)
    '''

    # convert angle to cos-sin and pack it up into camera2aruco_pose dict structure
    for i, j in enumerate(markers_on_sight):
        camera2aruco_pose[i]['pose2world'] = np.append(camera2aruco_pose[i]['pose2world'], 0)
        camera2aruco_pose[i]['pose2world'][3] = np.sin(np.deg2rad(camera2aruco_pose[i]['pose2world'][2]))
        camera2aruco_pose[i]['pose2world'][2] = np.cos(np.deg2rad(camera2aruco_pose[i]['pose2world'][2]))


    # assemble inputs and measurements
    u = np.array([0])
    y = np.hstack([camera2aruco_pose[i]['pose2world'] for i,j in enumerate(markers_on_sight)])

    # Perform a filtering step
    xhat = KF.filter_step(u, y, ret_xhat=True)

    ''' DOESNT WORK - KEPT FOR REFERENCE
    # undo the linear transformation of the angle and convert to +180/-180 space
    xhat[4] = np.mod(xhat[4] + np.rad2deg(angle_avg) - 180, 360) - 180
    '''

    # Retrieve the updated kalman gain of each marker to storage
```

```
188     for idx, id in enumerate(markers_on_sight):
189         markers[id].L = KF.L[:, 0+idx*3 :3+3*idx]
190 
191     return xhat[0], xhat[1], np.rad2deg(np.arctan2(xhat[3], xhat[2]))
192 
193 
194 if __name__ == '__main__':
195     print('You cannot run this script standalone. Go to main.py and call from there.')
```

118 lines (97 sloc) | 4.92 KB

```
##############################################################################
#    main-multiprocessing.py / pabloDarknetInference                         #
#    Pablo Santana - CNNs for object detection in subterranean environments  #
##############################################################################
# p0 (main)         p1                   p2                     p5
# ┌─────────────┐   ┌──────────────────┐ ┌──────────────────┐   ┌──────────────┐
# │ImageCapture│ q1 │Estimate Camera│ q3 │Kalman-Filter    │q4  │Draw Worldmap │
# │&& Undistort├──>│Pose From ArUco├──>│Camera Pose      ├──┐>│with cam FoV &│
# │            ||  │markers        │    │                 │  | │item positions│
# └─────────────┘|  └──────────────────┘ └──────────────────┘  | └──────────────┘
#                |  p3                   p4                   |
#                |  ┌──────────────────┐ ┌──────────────────┐  |
#                |q2 │Object detection│q5 │Estimate item    │q6|
#                └──>│with Darknet    ├──>│position w.r to  ├──┘
#                    │(CNN)           |   │the camera       |
#                    └──────────────────┘ └──────────────────┘
#                        └YOU'RE HERE

import cv2
import os
import sys

'''
Detection shape:

[('bottle',
  '83.45',
  (181.35324096679688,
   214.51708984375,
   41.65458679199219,
   144.4390869140625))]
'''

def mindYourDarknet(func):
    ''' Decorator handling the proper importing of darknet.
    '''
    def inner_function(*args, **kwargs):
        try:
            return func(*args, **kwargs)

        except ModuleNotFoundError:
            # If path hasn't been changed yet (darknet invisible to python)
            sys.path.append(os.getcwd())
            os.chdir('/home/pablo/YOLOv4/darknet')
            sys.path.insert(1, '/home/pablo/YOLOv4/darknet')
            return func(*args, **kwargs)

        except OSError:
            # Deals with the LD library path for CUDNN, run once per terminal execution
```

```
            print('''>>> Catched CUDNN error! Run this bash command and try again:\n
            \rexport LD_LIBRARY_PATH=/usr/local/lib:/usr/lib/cuda-10.2/targets/x86_64-linux/lib
            ''')
            quit()

    return inner_function


@mindYourDarknet
def networkBoot():
    ''' Boot a network model
    '''
    import darknet

    # load a nnet params
    config_file = '/home/pablo/YOLOv4/darknet/cfg/yolov4-tiny.cfg'
    data_file = '/home/pablo/YOLOv4/darknet/cfg/coco.data'
    weights = '/home/pablo/YOLOv4/darknet/cfg/yolov4-tiny.weights'

    # initialize object detection neural network
    network, class_names, colors = darknet.load_network(config_file, data_file, weights, batch_size=1)
    return network, class_names, colors



@mindYourDarknet
def imageDetection(image, network, class_names, class_colors, thresh=0.1):
    '''
    Modified version of the function from darknet_images.py that accepts
    a live frame instead of a file path for loading an image, conveting it to
    C format and performing Inference.
    '''
    import darknet

    # Darknet doesn't accept numpy images.
    # Create one with image we reuse for each detect
    width = darknet.network_width(network)
    height = darknet.network_height(network)
    darknet_image = darknet.make_image(width, height, 3)

    # image = cv2.imread(image_path)
    image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    image_resized = cv2.resize(image_rgb, (width, height),
                               interpolation=cv2.INTER_LINEAR)

    darknet.copy_image_from_bytes(darknet_image, image_resized.tobytes())
    detections = darknet.detect_image(network, class_names, darknet_image, thresh=thresh)
    darknet.free_image(darknet_image)
    image = darknet.draw_boxes(detections, image_resized, class_colors)
    return cv2.cvtColor(image, cv2.COLOR_BGR2RGB), detections



''' DEPRECATED BUT STORED TOWARDS CLEANER IMPORT
def prepareImportDarknet():
    # VERY NASTY way of importing the darknet bindings without installing extra packages
    sys.path.append(os.getcwd())
    os.chdir('/home/pablo/YOLOv4/darknet')
    sys.path.insert(1, '/home/pablo/YOLOv4/darknet')
    try:
        import darknet

    except OSError:
        # deals with the LD library path for CUDNN, run once per terminal execution
        print(""">>> Catched CUDNN error! Run this bash command and try again:\n
        \rexport LD_LIBRARY_PATH=/usr/local/lib:/usr/lib/cuda-10.2/targets/x86_64-linux/lib
        """)
        quit()
'''
```

58 lines (49 sloc) | 3.27 KB

```
##############################################################################
#    main-multiprocessing.py / pabloItem2CameraPosition                      #
#    Pablo Santana - CNNs for object detection in subterranean environments  #
##############################################################################
# p0 (main)         p1                   p2                      p5
# ┌─────────────┐    ┌──────────────────┐    ┌──────────────────┐    ┌──────────────┐
# │ImageCapture│ q1 │Estimate Camera│ q3 │Kalman-Filter    │q4  │Draw Worldmap │
# │&& Undistort├──->│Pose From ArUco├──->│Camera Pose      ├──┬>│with cam FoV &│
# │            ||   │markers        │    │                 │  | │item positions│
# └─────────────┘|   └──────────────────┘    └──────────────────┘  | └──────────────┘
#                |   p3                   p4                     |
#                |   ┌──────────────────┐    ┌──────────────────┐  |
#                |q2 │Object detection│q5 │Estimate item    │q6|
#                └──>│with Darknet    ├──->│position w.r to  ├──┘
#                    │(CNN)           |    │the camera       |
#                    └──────────────────┘    └──────────────────┘
#                                              └YOU'RE HERE

import numpy as np
import cv2

def filterDetections(detections, itemListAndSize):
    ''' Filter the detections: only items of interest + horizontal coordinates
    '''
    filteredComplete = [i for i in detections if i[0] in itemListAndSize]
    filteredPosition = [(i[0], i[2][0], i[2][2]) for i in detections if i[0] in itemListAndSize]
    return filteredComplete, filteredPosition


def getItemPosition(frameDetected, originalImage, itemDetections, itemListAndSize, mtx):
    ''' Computes the horizontal position of an item's bbox w.r. to the camera.
    '''
    # filter detections so only items of interest are fed forward
    _, filteredPosition = filterDetections(itemDetections, itemListAndSize)

    # Adjust the scaling factor, input_width=800 but darknet_width=416 (dets)
    rescaling_factor = originalImage.shape[1] / frameDetected.shape[1]

    item2CameraPosition = []
    for (bbox_class, bbox_hpos, bbox_width) in filteredPosition:
        # compute the pixel position in full-size image
        u1 = (bbox_hpos - bbox_width /2) * rescaling_factor
        u2 = (bbox_hpos + bbox_width /2) * rescaling_factor
        dx = itemListAndSize[bbox_class]

        # Perform math to get results (see README)
        fx = mtx[0,0]
        cx = mtx[0,2]
        s = fx * (dx) / (u2 - u1)
```

```
50          A = (u1 - cx)/(u2 - cx)
51          x1 = (A * dx) / (1 - A)
52          x2 = x1 + dx
53          z = (fx * x1 - s * u1) / cx
54
55          # Store as part of many possible positions
56          item2CameraPosition.append([bbox_class, z, (x1+x2)/2])
57
58      return item2CameraPosition
```

```
################################################################################
#    main-multiprocessing.py / pabloWorldMap                                   #
#    Pablo Santana - CNNs for object detection in subterranean environments    #
################################################################################
# p0 (main)         p1                   p2                      p5
# ┌─────────────┐    ┌──────────────────┐    ┌──────────────────┐    ┌──────────────┐
# │ImageCapture│ q1 │Estimate Camera│ q3 │Kalman-Filter    │q4  │Draw Worldmap │
# │&& Undistort├──->│Pose From ArUco├──->│Camera Pose      ├──┐>│with cam FoV &│
# │            ||   │markers        │    │                 │  | │item positions│
# └─────────────┘|   └──────────────────┘    └──────────────────┘  | └──────────────┘
#                |   p3                   p4                    |    └YOU'RE HERE
#                |   ┌──────────────────┐    ┌──────────────────┐  |
#                |q2 │Object detection│q5 │Estimate item    │q6│
#                └──>│with Darknet    ├──->│position w.r to  ├──┘
#                    │(CNN)           │    │the camera       │
#                    └──────────────────┘    └──────────────────┘
#
#

import numpy as np
from numpy.linalg import inv
import cv2


def rotationMatrix(theta, unit='rad'):
    if unit != 'rad':
        theta = np.deg2rad(theta)
    return np.array([[ np.cos(theta), -np.sin(theta)],
                     [ np.sin(theta), np.cos(theta)]])


def translationMatrix(dx, dy, theta):
    return np.array([[ np.cos(theta), -np.sin(theta), dx],
                     [ np.sin(theta),  np.cos(theta), dy],
                     [             0,              0,  1]])


def cameraCoordinateComply(item2CameraPosition, cameraPoseFiltered):
    ''' AD-HOC COMPENSATION! DO NOT USE BLINDLY.
    Applies a set of modifications to each of the results so far so that
    the compound system coordinate frames are consistent.
    '''
    # Invert XZ directions so +X==Left & +Z==Front
    item2CameraPosition = [[bbox_class, -z, -x] for (bbox_class, z, x) in item2CameraPosition]

    # Redefine yaw as the horiz-plane clock-wise angle between Zcam & Zworld
    cameraPoseFiltered = [cameraPoseFiltered[0],
                          cameraPoseFiltered[1],
                          180 - cameraPoseFiltered[2]]

```

```
    return item2CameraPosition, cameraPoseFiltered


def relu(x):
    return np.max(x, 0)



def imagePatch(img, img_overlay, x, y):
    ''' Overlap a smaller image on top of a bigger one
    '''
    # compensate for patch radious so that x,y tell the center
    h, w, _ = img_overlay.shape
    y -= h//2
    x -= w//2

    # Image ranges
    y1, y2 = max(0, y), min(img.shape[0], y + img_overlay.shape[0])
    x1, x2 = max(0, x), min(img.shape[1], x + img_overlay.shape[1])

    # Overlay ranges
    y1o, y2o = max(0, -y), min(img_overlay.shape[0], img.shape[0] - y)
    x1o, x2o = max(0, -x), min(img_overlay.shape[1], img.shape[1] - x)

    # Exit if nothing to do
    if y1 >= y2 or x1 >= x2 or y1o >= y2o or x1o >= x2o:
        return img

    # Add overlay within the determined ranges
    img[y1:y2, x1:x2] = img[y1:y2, x1:x2] + img_overlay[y1o:y2o, x1o:x2o]

    return np.clip(img, 0, 255)



class worldMap(object):
    ''' Implements a world map class in which item detections are to be drawn
    '''

    def __init__(self, mapsize_XZ_cm=[320,320], numItems=1):
        # unpack map size (default is for bedroom)
        self.sizeZcm = mapsize_XZ_cm[0]
        self.sizeXcm = mapsize_XZ_cm[1]

        # Map template
        self.map = np.zeros((self.sizeXcm, self.sizeZcm, numItems), np.float32)

        # Coordinates of the World Frame with respect to the Map Frame
        self.world2map_XZ_cm = [110, 0]


        ### Init FoV triangle
        #
        # distance of view & angle aperture
        self.fov_dist = 300
        self.fov_ang  = np.deg2rad(25)
        #
        # define ficticious ABC with respect to the camera frame for FoV
        # These represent a triangular FoV in the horizontal plane
        self.a2camera = np.array([self.fov_dist,  self.fov_dist*np.tan(self.fov_ang), 1])
        self.b2camera = np.array([self.fov_dist, -self.fov_dist*np.tan(self.fov_ang), 1])
        self.c2camera = np.array([0,0, 1])
        #
        # Concatenate in matrix for quicker computations
        self.abc2camera = np.c_[self.a2camera, self.b2camera, self.c2camera]

        '''For debugging - remember that tripod yaw is not coaxial with camera Y
        self.a2camera = np.array([40,  0, 1])
        self.b2camera = np.array([ 30, 30, 1])
```

```
        self.c2camera = np.array([ 0,  0, 1])'''


        ### Init for detections

        # GAUSSIAN KERNEL
        # Produce 2D gaussian kernel with top value close to white and shape (X,X,1)
        # Imshowing this kernel will not be representative, loot at detectionMask!
        ksize, sigma = [200, 20]
        kernel = cv2.getGaussianKernel(ksize=ksize,sigma=sigma)
        kernel = kernel @ kernel.T
        kernel *= 250/np.max(kernel)
        self.gaussianKernel2D = np.array([kernel], np.uint8).reshape(ksize,ksize,1)

        # Speed at which items appear / disappear
        self.disappearanceRate = 0.003
        self.appearanceRate = 0.005



    def getFoV_demo2world(self, cameraPose):
        # unpack coordinates for readability, mm-deg -> cm-rad
        x_cam   = 0.1 * cameraPose[0]
        z_cam   = 0.1 * cameraPose[1]
        yaw_cam = np.deg2rad(cameraPose[2])

        a2world = translationMatrix(z_cam, x_cam, yaw_cam) @ self.a2camera
        b2world = translationMatrix(z_cam, x_cam, yaw_cam) @ self.b2camera
        c2world = translationMatrix(z_cam, x_cam, yaw_cam) @ self.c2camera

        return a2world, b2world, c2world


    def getPoints_demo2world(self, item2CameraPositionCompliant, cameraPose):
        # Might be many points - parallelizable matrix implementation
        # unpack coordinates for readability, mm-deg -> cm-rad
        x_cam   = 0.1 * cameraPose[0]
        z_cam   = 0.1 * cameraPose[1]
        yaw_cam = np.deg2rad(cameraPose[2])

        # Remove class string and create a matrix in which each column is a position vector
        # we add a residual 1 to match matrix dimensions...
        # ...but we add as a 10 then divide the matrix by 10 to convert mm->cm
        p2camera = [i[1:] + [10] for i in item2CameraPositionCompliant]
        p2camera = np.array(p2camera).T/10

        # compute the translation matrix (common for all) and broadcast it along one dimension
        tm = translationMatrix(z_cam, x_cam, yaw_cam)
        # BROADCASTING NOT REQUIRED BUT LEFT COMMENTED FOR REFERENCE
        ##tm = np.repeat(tm[np.newaxis,:,:], len(item2CameraPositionCompliant), axis=0)

        # simultaneously transform the points in p2camera with the translation matrix
        # output vector contains a matrix in which each column is a point
        return tm @ p2camera



    def getFoV_2map(self, cameraPose):
        ''' Convert the coordinates of the ABC points that represent the FoV
        to map absolute coordinates.
        '''

        # unpack coordinates for readability, mm-deg -> cm-rad
        # Because WorldFrame and MapFrame are square to each other we add coords
        x_cam2map   = 0.1 * cameraPose[0]  + self.world2map_XZ_cm[0]
        z_cam2map   = 0.1 * cameraPose[1]  + self.world2map_XZ_cm[1]
        yaw_cam2map = np.deg2rad(cameraPose[2])

        # Apply homogeneous transformation matrix to the three points at once
```

```
        abc2map = translationMatrix(z_cam2map, x_cam2map, yaw_cam2map) @ self.abc2camera

        # Remove the tailing [1]s and swap coordinate place so that...
        # ...cv2 reads 3 different 2D points the way we want to. (ad-hoc)
        # output has the shape [[x1,z1], [x2,z2], [x3,z3]]
        abc2map = abc2map[[1,0],:].T

        return abc2map.astype(int)


    def getPoints_2map(self, item2CameraPosition, cameraPose):
        ''' Convert the coordinates of arbitrary points from camera
        to map absolute coordinates.
        '''

        # Might be many points - parallelizable matrix implementation
        # unpack coordinates for readability, mm-deg -> cm-rad
        x_cam2world  = 0.1 * cameraPose[0]
        z_cam2world   = 0.1 * cameraPose[1]
        yaw_cam2world = np.deg2rad(cameraPose[2])

        # Because WorldFrame and MapFrame are square to each other
        x_cam2map = x_cam2world + self.world2map_XZ_cm[0]
        z_cam2map = z_cam2world + self.world2map_XZ_cm[1]
        yaw_cam2map = yaw_cam2world

        # Remove class string and create matrix in which each column is a position vector
        # we append a residual [1] to match matrix dimensions later...
        # ...but we add as a [10] then divide the matrix by 10 to convert mm->cm
        p2camera = [i[1:] + [10] for i in item2CameraPosition]
        p2camera = np.array(p2camera).T / 10

        # simultaneously transform the points in p2camera with the translation matrix
        # output vector contains a matrix in which each column is a point
        p2map = translationMatrix(z_cam2map, x_cam2map, yaw_cam2map) @ p2camera

        # Remove the tailing [1]s and swap coordinate place so that...
        # ...cv2 reads 3 different 2D points the way we want to. (ad-hoc)
        p2map = p2map[[1,0],:].T

        return p2map.astype(int)


    def update(self, item2CameraPosition, cameraPose):
        ''' Update existing likelihood map with any detection / missdetection
        '''

        # Get FoV mask (region of visible points)
        self.fovMask = np.zeros((self.sizeXcm, self.sizeZcm, 1), np.float32)
        fovTriangle = self.getFoV_2map(cameraPose)
        cv2.drawContours(self.fovMask, [fovTriangle], 0, 255, -1)


        # Get discovery mask (regions where items are thought to be NOW)
        self.discoveryMask = np.zeros((self.sizeXcm, self.sizeZcm, 1), np.uint16)
        if item2CameraPosition != []:
            item2MapPosition = self.getPoints_2map(item2CameraPosition, cameraPose)
            for (x, y) in item2MapPosition:
                self.discoveryMask = imagePatch(self.discoveryMask, self.gaussianKernel2D, x, y)


        # Update likelihood map. Uniform decrease + gaussian increase. Clip for grayscale
        self.map = np.clip(
            self.map - self.fovMask * self.disappearanceRate
                     + self.discoveryMask * self.appearanceRate,
            0, 255)

        # this to show fov over detections
        self.map_fov = self.map.copy()
```

```
257         cv2.drawContours(self.map_fov, [fovTriangle], 0, 255, 2)
258         cv2.circle(self.map_fov, tuple(fovTriangle[2]), 20, 255, 9)
259 
260         '''
261         # moved to main-multiprocessing.py
262         cv2.imshow('Field of view', self.fovMask)
263         cv2.imshow('Instantaneous discovery', np.array(self.discoveryMask, np.uint8))
264         cv2.imshow('Likelihood map', self.map)
265         cv2.waitKey(1)
266         '''
267 
268 
269 
270         #print(xz_item2world[0])
271         #self.map = cv2.circle(self.map, (int(i) for i in xz_item2world[0]), 20, (255), -1)
272 
273         # convert continuous to discrete coordinates
274         # x_i = np.digitize(x, self.x_bins)
275         # z_i = np.digitize(z, self.z_bins)
276 
277         # self.map[z_i][x_i] += 0.5
278         # decrease probability for false negatives that are recognized now
279         #self.map = self.map - self.mask_removal
280         # bound values
281         #self.map = np.clip(self.map, 0, 2)
```